El problema de la búsqueda

Algoritmo de búsqueda

- Un **algoritmo de búsqueda** es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

- Por ejemplo, ubicar el registro correspondiente a cierta persona en una base de datos, o el mejor movimiento en una partida de ajedrez.

- La variante más simple del problema es la búsqueda de un número en una lista.

Búsqueda secuencial

Búsqueda secuencial

- Consiste en buscar el elemento comparándolo secuencialmente (de ahí su nombre) con cada elemento del arreglo o conjunto de datos hasta que se encuentre, o hasta que se llegue al final del arreglo.
- La existencia se puede asegurar desde el momento que el elemento es localizado, pero no podemos asegurar la no existencia hasta no haber analizado todos los elementos del arreglo.
- Se utiliza cuando el contenido del arreglo no se encuentra o no puede ser ordenado.

- La búsqueda lineal comprueba secuencialmente cada elemento de la lista hasta que encuentra un elemento que coincide con el valor de objetivo.

- Si el algoritmo llega al fin de la lista sin encontrar el objetivo, la búsqueda termina insatisfactoriamente.

- Es la menos eficiente en tiempo, ya que realiza como máximo ***n*** comparaciones, donde ***n*** es la longitud del arreglo.

Código 1
Búsqueda secuencial

```
int busquedaSecuencial(int  arreglo[], int n, int dato)
{
    int i;
    for(i=0; i<n; i++)
    {
        if(dato == arreglo[i])
            return i;
    }
    return -1;
}
```

Código 2
Búsqueda secuencial

```
int busquedaSecuencial(int arr[], int len, int v)
{
    int i = 0;
    while( i<len && arr[i]!=v )
    {
        i = i + 1;
    }
    return  i<len?i:-1;
}
```

Búsqueda binaria

Búsqueda binaria

- El algoritmo de la búsqueda binaria puede plantearse como una función recursiva.
- Esta función recibirá el arreglo ***arr***, el valor ***v*** que se quiere buscar y dos índices, ***i*** y ***j***, que delimitarán las posiciones de inicio y fin.
- Independientemente de qué valores contengan los índices ***i*** y ***j***, siempre vamos a comparar a ***v*** con el valor que se encuentre en la posición promedio del arreglo **k = (i+j)/2**

3	4	6	8	9	12	14	18	21	25	29	30	36	38	41	47
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

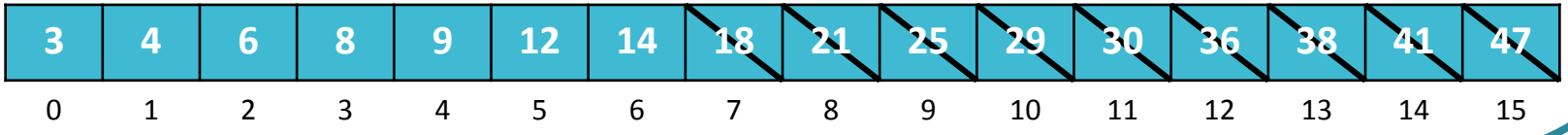
- Luego, descartaremos la mitad del arreglo anterior a ***k*** o la mitad posterior a esta posición dependiendo de que ***v*** sea mayor o menor que ***arr[k]***.
- Imaginemos el siguiente arreglo ***arr*** ordenado ascendentemente.
- Las posiciones de inicio y fin son i =0 y j = 15, esta última coincide con la longitud del arreglo.

3	4	6	8	9	12	14	18	21	25	29	30	36	38	41	47
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Para determinar si ***arr*** contiene el valor *v=14* obtenemos la posición promedio calculándola como k=(i+j)/2.

- En este caso k = (0+15)/2 = 7

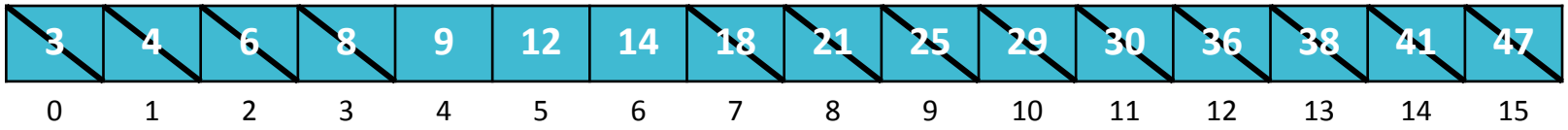
- En la posición 7 encontraremos el valor 18 que, al ser mayor que v, nos permite descartar esta posición y todas las posiciones posteriores porque, al tratarse de un arreglo ordenado, en caso de estar el valor 14, este se encontrará en alguna posición anterior.



- Si ahora repetimos la operación, la posición promedio será k=3.

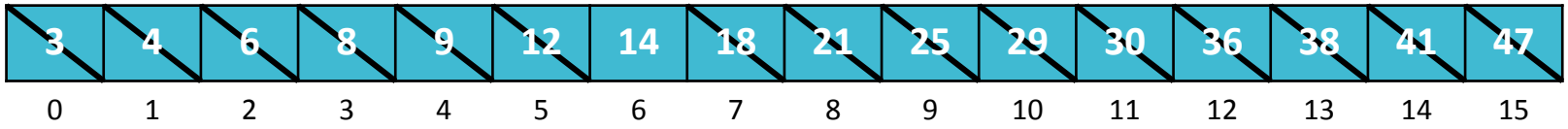
- Como **arr[k]** es menor que **v** estamos en condiciones de descartar todas las posiciones anteriores.

- Para esto, invocamos a la función recursiva pasándole ***arr***, ***v*** y los valores ***i=k+1*** y ***j***.



- La posición promedio ahora será k=5.

- Como arr[k] es menor que v, de estar el 14, debería ser en alguna posición posterior.



- Luego, en la posición promedio k=6, arr[k] contiene al valor que buscábamos.

arr[6] = 14

Búsqueda binaria

```
int busquedaBinaria(int arr[], int v, int i, int j)
{
    int k = (i+j)/2;

    //condición de corte
    if(i>j)
        return -i;

    if(arr[k]==v)
        return k;
    else
    {
        if(arr[k]<v)
            i=k+1;
        else
            j=k-1;

        //invocacion recursiva
        return busquedaBinaria(arr,v,i,j);
    }
}
```

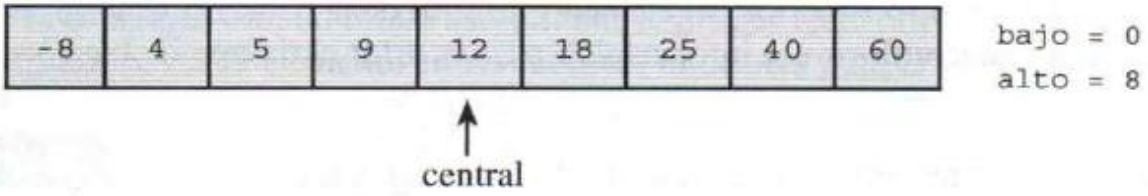
```
int main()
{
    int arreglo[] = {2,4,5,8,12,18,23,45};
    int v = 6;

    printf("%i", busquedaBinaria(arreglo, v, 0, 7));
}
```

- La condición de corte en esta función se dará cuando se crucen los índices ***i*** y ***j***.

- En este caso la función retornará un valor negativo tal que, tomándolo en absoluto, coincidirá con la posición en la que debería insertarse el valor ***v*** que buscamos y no encontramos dentro del arreglo.

Búsqueda binaria

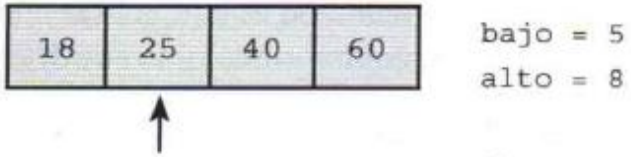
1.



$$central = \frac{bajo + alto}{2} = \frac{0 + 8}{2} = 4$$

clave (40) > a[4] (12)

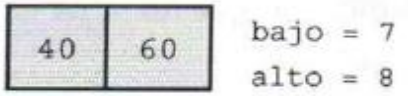
2. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{5 + 8}{2} = 6 \ (\textit{división entera})$$

clave (40) > a[6] (25)

3. Buscar en sublista derecha



$$central = \frac{bajo + alto}{2} = \frac{7 + 8}{2} = 7$$

clave (40) = a[7](40) búsqueda con éxito

Búsqueda binaria

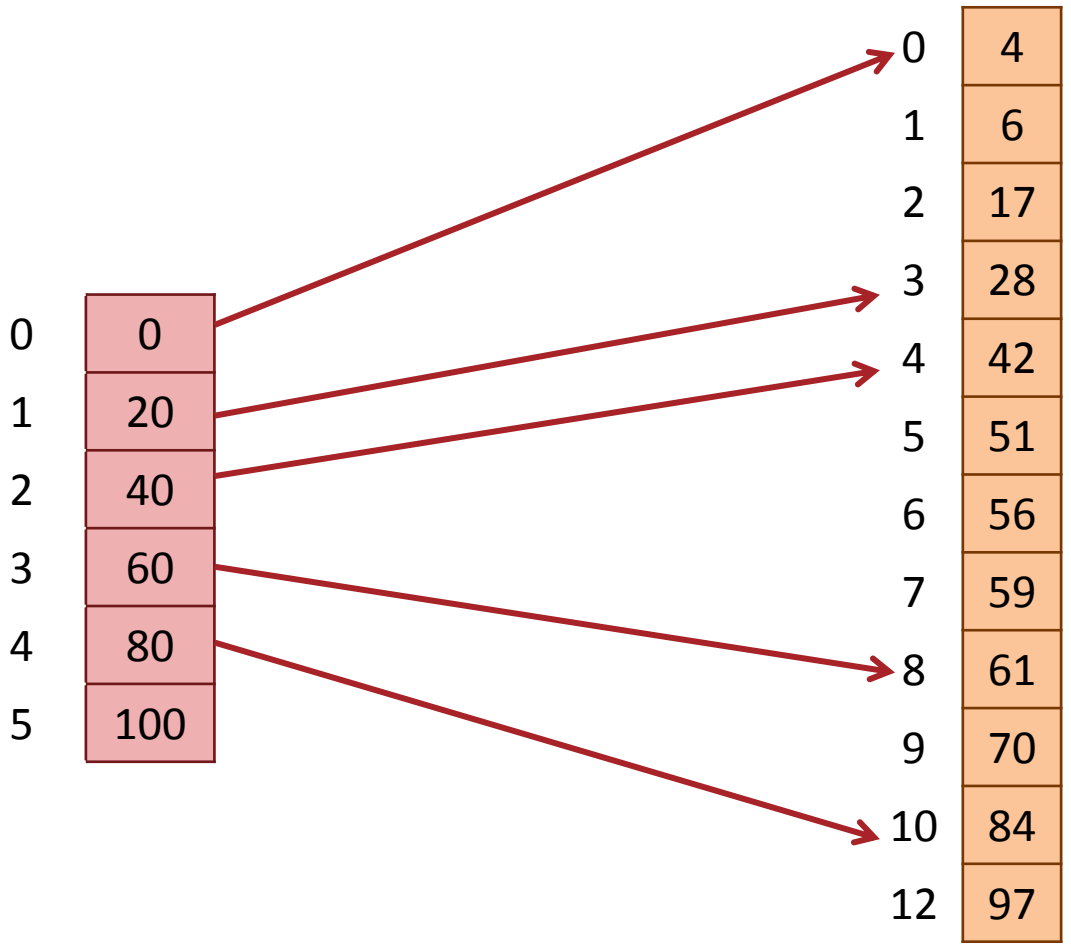
```
int busquedaBin(int lista[], int n, int clave)
{
    int central, bajo, alto;
    int valorCentral;
    bajo = 0;
    alto = n-1;

    while(bajo <= alto)
    {
        central = (bajo + alto)/2;        //índice de elemento central
        valorCentral = lista[central];    //valor del índice central
        if(clave == valorCentral)
            return central;               //encontrado devuelve posicion
        else if(clave < valorCentral)
            alto = central - 1;           //ir a sublista inferior
        else
            bajo = central + 1;           //ir a sublista superior
    }
    return -1;                            //elemento no encontrado
}
```

Búsqueda indexada

Búsqueda indexada

- Mediante cada elemento del arreglo índice se asocian grupos de elementos del arreglo inicial.
- Los elementos en el índice y en el arreglo deben estar ordenados.
- El método consta de dos pasos: Buscar en el arreglo índice el intervalo correspondiente al elemento buscado y restringir la búsqueda a los elementos del intervalo que se localizó previamente.
- La ventaja es que la búsqueda se realiza en el arreglo de índices y no en el arreglo de elementos.
- Cuando se ha encontrado el intervalo correcto se hace segunda búsqueda en una parte reducida del arreglo.
- Éstas dos búsquedas pueden ser secuenciales o binarias y, el tiempo de ejecución dependerá del tipo de búsqueda utilizado en cada uno de los arreglos.

Indexación

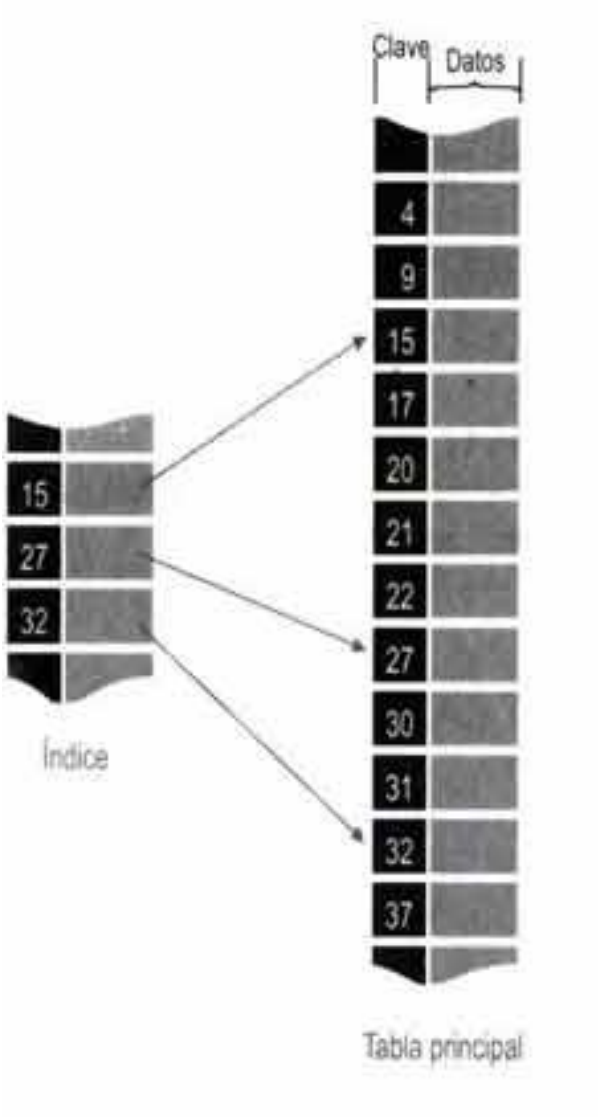

0
1
2
3
4
5
0
20
40
60
80
100
0
1
2
3
4
5
6
7
8
9
10
12
4
6
17
28
42
51
56
59
61
70
84
97

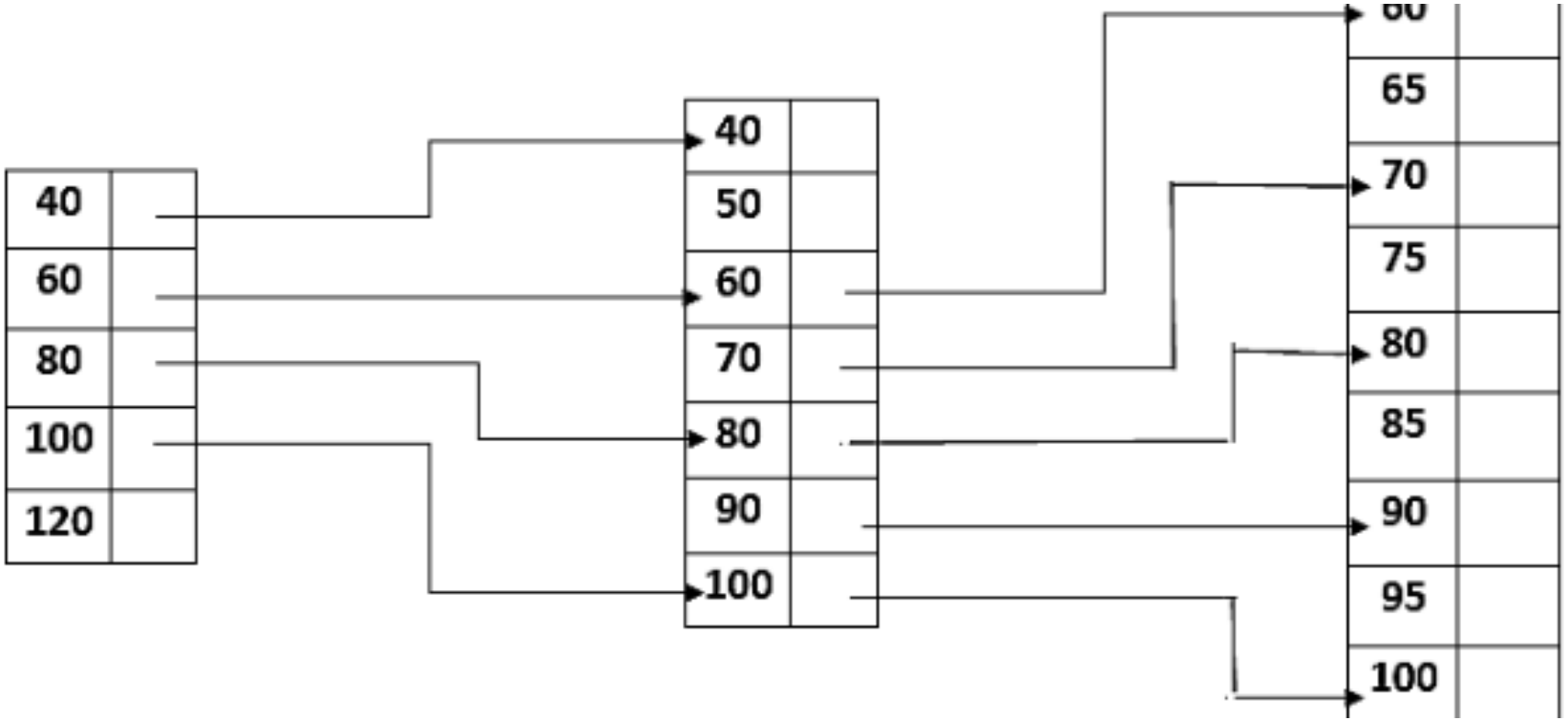
Búsqueda secuencial indexada

Un método popular para superar las desventajas de los archivos secuenciales es el del archivo de secuencias indexado; pero implica un aumento en la cantidad de espacio requerida.

Funciona de la siguiente manera:

- Se reserva una tabla auxiliar llamada índice, además del archivo ordenado mismo.
- Cada elemento en el índice consta de una llave kindex y un apuntador al registro en el archivo que corresponde a kindex.
- Los elementos en el índice al igual que los elementos en el archivo, deben estar ordenados en la llave.
- Si el índice es de un octavo del tamaño del archivo, se representa en el índice y cada octavo se registra el archivo.





Búsqueda secuencial indexada

Comparación de órdenes de complejidad de las búsquedas

Complejidad algorítmica

- Si dos algoritmos diferentes resuelven el mismo problema entonces los llamamos algoritmos equivalentes.

- La complejidad algorítmica permite establecer una comparación entre algoritmos equivalentes para determinar en forma teórica, cuál tendrá mejor rendimiento en condiciones extremas y adversas.

- Para esto se trata de calcular cuántas instrucciones ejecutará el algoritmo en función del tamaño de los datos de entrada.
- Llamamos “instrucción” a cada línea de código.
- Como resultaría imposible medir el tiempo que demora una computadora en ejecutar una instrucción, simplemente diremos que cada una demora 1 unidad de tiempo en ejecutarse.
- Luego, el algoritmo más eficiente será aquel que requiera menor cantidad de unidades de tiempo para concretar su tarea.

Análisis de búsqueda secuencial

```
int busquedaSecuencial(int arr[], int len, int v)
{
    int i = 0;
    while( i<len && arr[i]!=v )
    {
        i = i + 1;
    }
    return  i<len?i:-1;
}
```

La complejidad búsqueda secuencial es lineal

O(n)

Análisis búsqueda binaria

- Después de ordenar un conjunto numérico en una secuencia creciente o decreciente, el algoritmo de búsqueda binaria empieza desde la parte media de la secuencia.
- Si el punto de prueba es igual al punto medio de la secuencia, finaliza el algoritmo.
- En caso contrario, dependiendo del resultado de comparar el elemento de prueba y el punto central de la secuencia, de manera recurrente se busca a la izquierda o a la derecha de la secuencia.

Primera iteración:
i=0 y j=len

Segunda iteración:
Se procesará la mitad del array ya que habremos desplazado algunos de los índices i, j.

len/2 elementos

Tercera iteración:
Se procesará la mitad de la mitad del array, es decir, len/2/2, lo que equivale a decir:

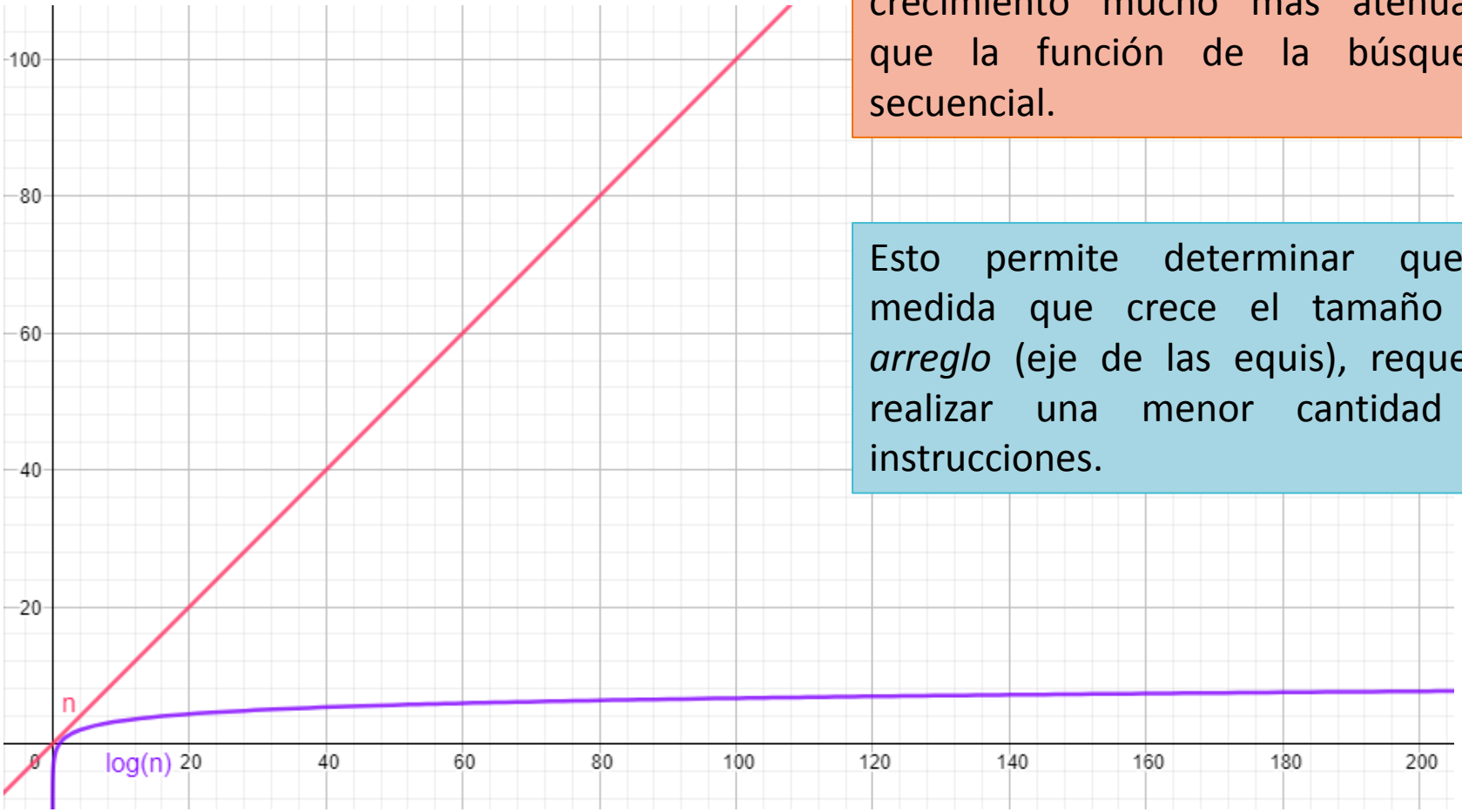
len/2^2 elementos

Cuarta iteración:
len/2/2/2 = len/2^3

$$\frac{len}{2^n}=1$$

```
int busquedaBinaria(int arr[], int len, int v)
{
	int i = 0;
	int j = len;
	int enc = 0; //false

	while(i<=j && !enc)
	{
		int k = (i+j)/2;
		if(arr[k]==v)
			enc = 1; //true
		else
		{
			if(arr[k]<v)
				i = k + 1;
			else
				j = k - 1;
		}
	}
	return enc?k:-1;
}
```

La complejidad de **búsqueda binaria** es logarítmica
O(log n)



Como vemos, la función de la búsqueda binaria tiene un crecimiento mucho más atenuado que la función de la búsqueda secuencial.

Esto permite determinar que a medida que crece el tamaño del *arreglo* (eje de las equis), requerirá realizar una menor cantidad de instrucciones.

Tiempos de ejecución

Número de datos búsqueda	Secuencial	Binaria
10	0.05	0.001
100	0.29	0.003
*1,000	0.32	0.003
*10,000	0.64	0.001
*100,000	4.8	0.001
*200,000	7.3	0.001