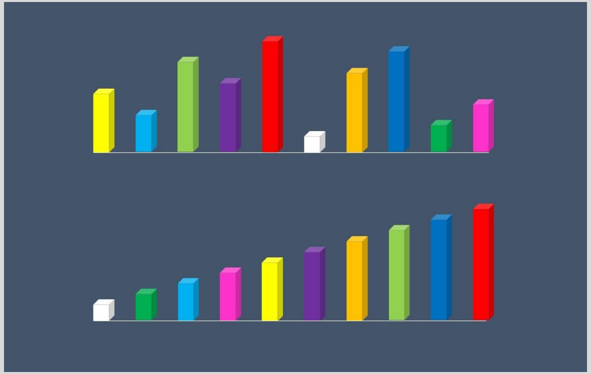
Ordenamientos

¿Qué tener en cuenta en un algoritmo de ordenamiento?



- Tiempo de ejecución
- Memoria extra
- Fiabilidad

Función swap

Regularmente en varios algoritmos de ordenamientos se usa alguna función swap que intercambia dos elementos de posición en una misma colección.

```
swap (array, i, j):
    tmp = array[ i ]
    array[ i ] = array[ j ]
    array[ j ] = tmp
```

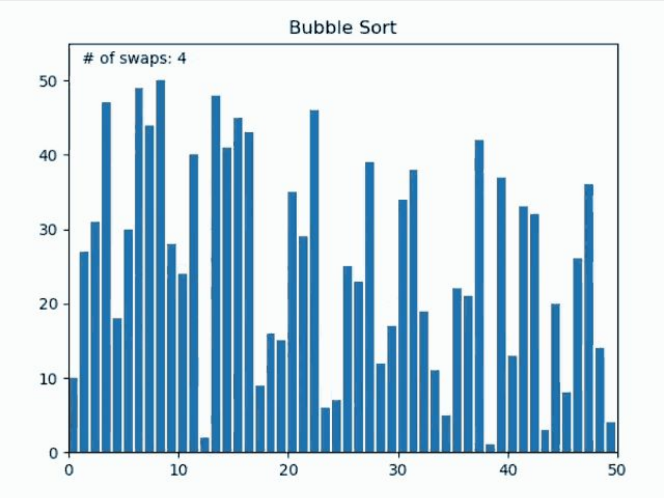
Programa de ordenamientos

ED

ESTRUCTURAS DE DATOS

Bubblesort

```
for ( i = N - 1; i > 0; i-- )
    for ( j = 0; j < i; j++ )
        if ( array[ j ] > array[ j + 1 ]  )
            swap ( array, j, j + 1 )
```



* Esto es solo pseudocódigo. Para que la comparación funcione en Java tendría que ser un arreglo de números, o bien, usar algún método para comparar objetos.

ED
ESTRUCTURAS DE DATOS

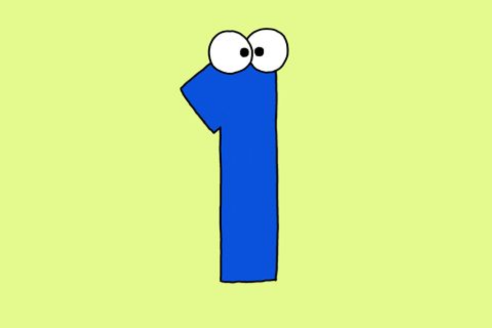
Invariante en ciclos

Una **invariante** es un enunciado lógico que no varía durante el ciclo.

Para demostrar que un algoritmo iterativo es correcto, se establece alguna invariante y se debe demostrar:

1. **Inicialización:** La invariante es verdadera antes de la primera iteración.
2. **Mantenimiento:** Si la invariante es verdadera antes de cierta iteración del ciclo, entonces debe ser verdadera antes de la siguiente iteración.
3. **Terminación:** Al terminar el ciclo, la invariante nos ayuda a demostrar que el algoritmo es correcto.

Ejemplo



Demostrar la correctud del algoritmo Bubblesort:

```
1  for ( i = N - 1; i > 0; i-- )
2      for ( j = 0; j < i; j++ )
3          if ( array[ j ] > array[ j + 1 ]  )
4              swap ( array, j, j + 1 )
```

Para demostrar que el for de las líneas 1-4 ordena correctamente, primero hay que demostrar que el for de las líneas 2-4 es correcto.

Por demostrar que el ciclo de las líneas 2-4 coloca en **array[i]** un elemento máximo para el subarreglo **array[0 : i]**.

<u>Invariante del ciclo:</u> Al iniciar la j-ésima iteración, en la posición **array[j]** está el máximo del subarreglo **array[0 : j]**.

Ejemplo

```
2    for ( j = 0; j < i; j++ )
3        if ( array[ j ] > array[ j + 1 ] )
4            swap ( array, j, j + 1 )
```

<u>Invariante del ciclo:</u> Al iniciar la j-ésima iteración, en la posición **array[j]** está un máximo del subarreglo **array[0 : j]**.

Inicialización: Antes de iniciar la primera iteración en la que j tiene que valer 0, el subarreglo **array[0 : j]** sería **array[0 : 0]**, que consiste solo del elemento **array[0]**, por lo que ese elemento es un máximo de dicho subarreglo.

Mantenimiento: Antes de iniciar la j-ésima iteración, en **array[j]** se tiene un máximo para el subarreglo **array[0 : j]**.

Al hacerse la comparación de la línea 3 hay 2 casos:

- **array[j] ≤ array[j + 1]**

En este caso terminaría la iteración por no entrar al cuerpo del if. Como en **array[j]** hay un máximo para el subarreglo **array[0 : j]** y **array[j] ≤ array[j + 1]**, entonces en **array[j + 1]** se tiene un máximo para el subarreglo **array[0 : j + 1]**.

Ejemplo

```
2     for ( j = 0; j < i; j++ )
3         if ( array[ j ] > array[ j + 1 ]  )
4             swap ( array, j, j + 1 )
```

Invariante del ciclo: Al iniciar la j-ésima iteración, en la posición **array[j]** está un máximo del subarreglo **array[0 : j]**.

Mantenimiento(Continuación): El otro caso para la línea 3 es:

- **array[j] > array[j + 1]**

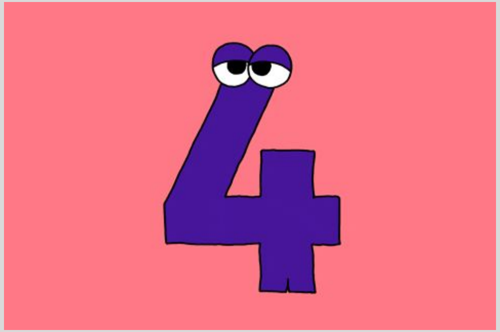
Digamos que en **array[j]** está el elemento **m** que es mayor o igual que cualquier elemento en **array[0 : j - 1]**, mientras que en **array[j + 1]** está el elemento **s**, tal que **m > s**.

Después de hacer el swap de la línea 4, en **array[j]** ahora estará **s** y en **array[j + 1]** estará **m**. Como **m > s**, entonces ahora **array[j + 1] > array[j]**. Además como **m** es mayor o igual que cualquier elemento en **array[0 : j - 1]**, entonces **array[j + 1]** es mayor o igual que cualquier elemento en **array[0 : j - 1]**.

Por lo tanto, al finalizar esta iteración **array[j + 1]** es un máximo del subarreglo **array[0 : j + 1]**

Como ya abordamos todos los casos para la j-ésima iteración, podemos concluir que, antes de iniciar la j+1-ésima iteración, en **array[j + 1]** hay un máximo del subarreglo **array[0 : j + 1]**

Ejemplo



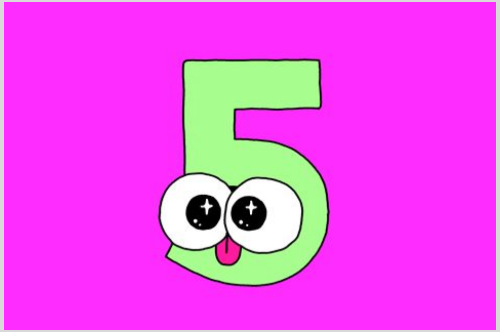
```
2     for ( j = 0; j < i; j++ )
3         if ( array[ j ] > array[ j + 1 ]  )
4             swap ( array, j, j + 1 )
```

Invariante del ciclo: Al iniciar la j-ésima iteración, en la posición **array[j]** está un máximo del subarreglo **array[0 : j]**.

Terminación: El ciclo termina cuando **j == i**, por lo que al iniciar una supuesta siguiente iteración tendríamos en **array[i]** un máximo del subarreglo **array[0 : i]**.

∴ El ciclo de las líneas 2-4 coloca en **array[i]** un elemento máximo para el subarreglo **array[0 : i]**.

Ejemplo



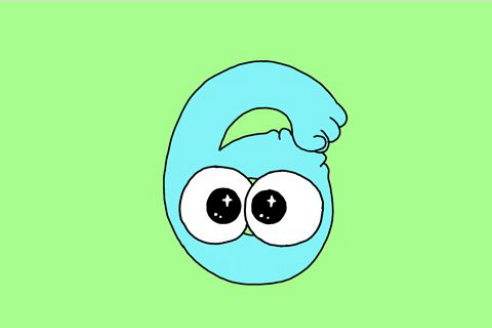
```
1  for ( i = N - 1; i > 0; i-- )
2      for ( j = 0; j < i; j++ )
3          if ( array[ j ] > array[ j + 1 ]  )
4              swap ( array, j, j + 1 )
```

Por demostrar que el ciclo de las líneas 1-4 ordena correctamente los elementos del arreglo.

Invariante del ciclo: Al iniciar la iteración en la que **i == k**, los elementos del subarreglo **array[k + 1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que cualquier elemento del subarreglo **array[0 : k]**.

Inicialización: Antes de iniciar la primera iteración en la que i tiene que valer N - 1, el subarreglo **array[k + 1 : N - 1]** sería **array[(N - 1) + 1 : N - 1] = array[N : N - 1]**, que consiste en ningún elemento puesto que **N > N - 1**, por lo que podemos decir que los elementos del subarreglo ya están ordenados de forma ascendente y son mayores que cualquier elemento del subarreglo **array[0 : k]** por vacuidad.

Ejemplo



```
1    for ( i = N - 1; i > 0; i-- )
2-4       Colocar en array[ i ] un elemento
          máximo para el subarreglo array[ 0 : i ]
```

<u>Invariante del ciclo:</u> Al iniciar la iteración en la que **i == k**, los elementos del subarreglo **array[k + 1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que cualquier elemento del subarreglo **array[0 : k]**.

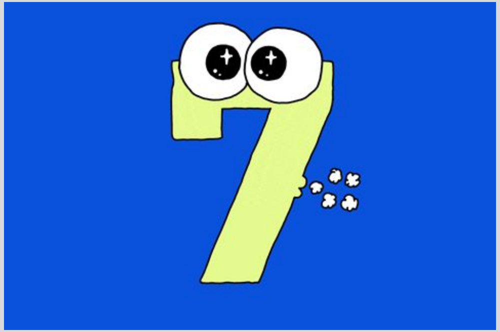
Mantenimiento: Al iniciar la iteración en la que **i == k** los elementos del subarreglo **array[k + 1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que cualquier elemento del subarreglo **array[0 : k]**.

Después de las líneas 2-4 tendríamos en **array[k]** un máximo para el subarreglo **array[0 : k]**.

Finalmente, el subarreglo **array[k : N - 1]** se encuentra ordenado porque **array[k]** es menor o igual que cualquier elemento en **array[k + 1 : N - 1]** que además ya estaba ordenado y, al ser **array[k]** un máximo para el subarreglo **array[0 : k]**, entonces cualquier elemento de **array[k : N - 1]** es mayor o igual que cualquier elemento del subarreglo **array[0 : k - 1]**.

Por lo tanto, antes de iniciar la siguiente iteración en la que **i == k - 1**, los elementos del subarreglo **array[k : N - 1] = array[(k - 1) + 1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que cualquier elemento del subarreglo **array[0 : k - 1]**.

Ejemplo



```
1  for ( i = N - 1; i > 0; i-- )
2      for ( j = 0; j < i; j++ )
3          if ( array[ j ] > array[ j + 1 ]  )
4              swap ( array, j, j + 1 )
```

Invariante del ciclo: Al iniciar la iteración en la que **i == k**, los elementos del subarreglo **array[k + 1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que cualquier elemento del subarreglo **array[0 : k]**.

Terminación: El ciclo termina cuando **i == 0**, por lo que al iniciar una supuesta siguiente iteración tendríamos que los elementos del subarreglo **array[0 + 1 : N - 1] = array[1 : N - 1]** están ordenados de forma ascendente y son mayores o iguales que los elementos del subarreglo **array[0 : 0]** que consiste solo en el elemento **array[0]**.

De esto se sigue que los elementos del subarreglo **array[0 : N - 1]** están ordenados de forma ascendente. Como ese subarreglo abarca a todos los elementos del arreglo, entonces **array** ya está ordenado.

∴ Por lo tanto el ciclo de las líneas 1-4 que corresponde con el algoritmo Bubblesort ordena correctamente.

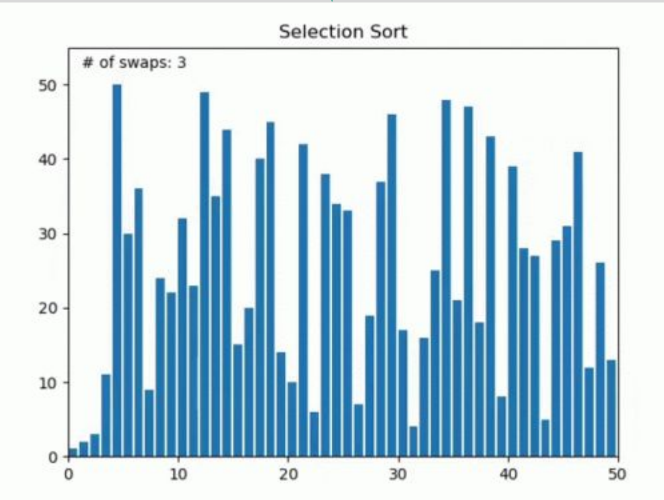
Ejercicio

Calcula la complejidad en tiempo del algoritmo Bubblesort:

```
for ( i = N - 1; i > 0; i-- )
    for ( j = 0; j < i; j++ )
        if ( array[ j ] > array[ j + 1 ]  )
            swap ( array, j, j + 1 )
```

Selectionsort

En este gif se selecciona el mínimo en lugar del máximo.




```
1  for ( i = N - 1; i > 0; i-- )
2      max = 0
3      for ( j = 1; j ≤ i; j++ )
4          if ( array[ j ] > array[ max ]  )
5              max = j
6      swap ( array, max, i )
```

<u>Invariante para el ciclo 3-5</u>: Al inicio de la j-ésima iteración, en **array[max]** hay un máximo para el subarreglo **array[0 : j - 1]**.

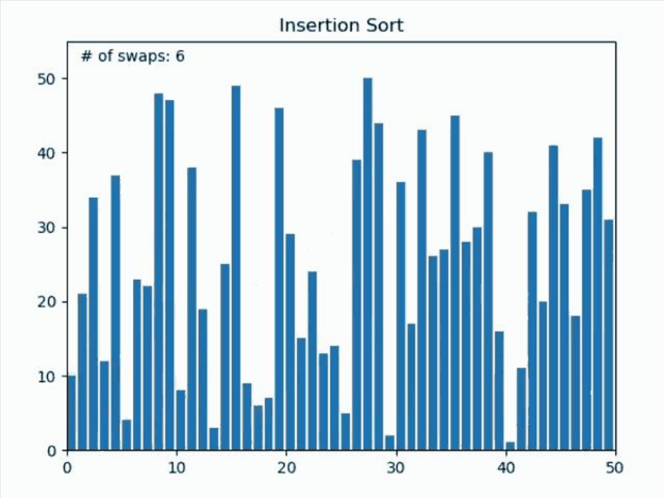
Ejercicio

Calcula la complejidad en tiempo del algoritmo Selectionsort:

```
for ( i = N - 1; i > 0; i-- )
    max = 0
    for ( j = 1; j < i + 1; j++ )
        if ( array[ j ] > array[ max ]  )
            max = j
    swap ( array, max, i )
```

Insertionsort

```
for ( i = 0; i < N - 1; i++ )
    for ( j = i + 1; j > 0 && arr[ j - 1 ] > arr[ j ]; j-- )
        swap ( array, j, j - 1 )
```



ED
ESTRUCTURAS DE DATOS

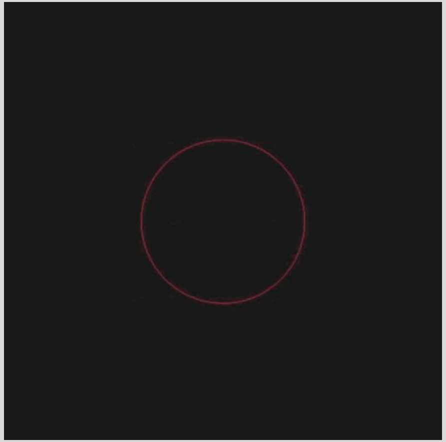
Ejercicio

Calcula la complejidad en tiempo del algoritmo Insertionsort en **el mejor caso:**

```
for ( i = 0; i < N - 1; i++ )
    for ( j = i + 1; j > 0 && arr[ j - 1 ] > arr[ j ]; j-- )
        swap ( array, j, j - 1 )
```

Estrategia "Divide y vencerás"

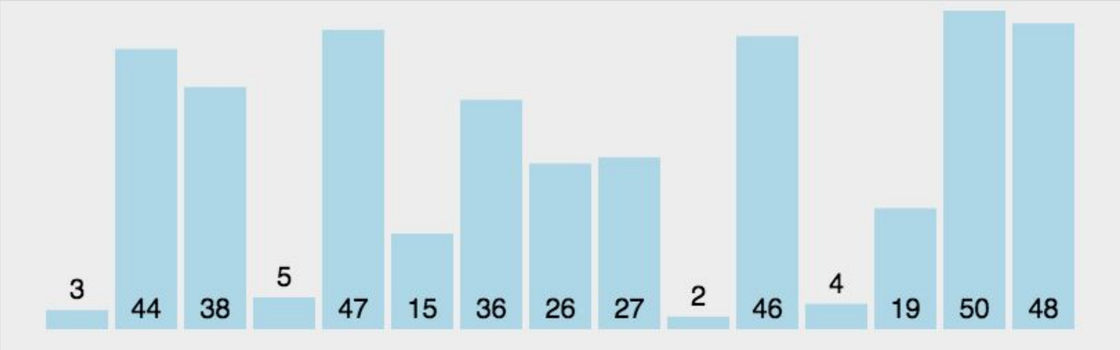
Algunos algoritmos emplean una estrategia en la que dividen al problema en subproblemas más pequeños, con la finalidad de armar una solución total a partir de las soluciones de los subproblemas.



Quicksort - 1961 C. A. R. Hoare

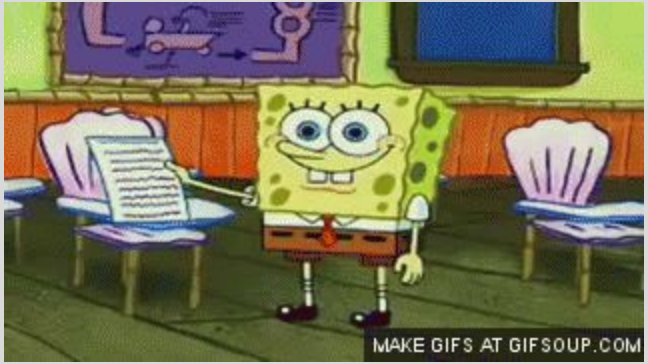
```
quicksort ( arr [ ] ) :
    quicksort ( arr, 0, N - 1 )
```

```
quicksort ( arr [ ], lo, hi ) :
    if ( hi ≤ lo ) return
    j = partition (arr, lo, hi)
    quicksort ( arr, lo, j - 1)
    quicksort ( arr, j + 1, hi)
```



Quicksort - Algoritmo de partición

```
partition ( arr [ ], lo, hi ) :
   i = lo
   j = hi + 1
   piv = arr [ lo ]
   while ( true ) :
      while ( arr [ ++i ] < piv ) if ( i == hi ) break
      while ( piv < arr [ --j ] ) if ( j == lo ) break
      if ( i ≥ j ) break
      swap ( arr, i, j )
   swap (arr, lo, j )
   return j
```



Complejidad de Quicksort

	Tiempo	Espacio
Caso promedio	O(n log n)	O(log n)
Peor caso	O(n^2)	O(n)

A A A A A A
A A A A A A
A A A A A A
A A A A A A
A A A A A A
A A A A A A
A A A A A A

Un mal escenario para Quicksort es cuando ordena elementos repetidos

ED
ESTRUCTURAS DE DATOS

Estabilidad en ordenamientos

Sea **A** una colección cualquiera y **A'** el resultado de aplicarle un ordenamiento, se dice que dicho ordenamiento es ***estable*** si y solo si:

$$A[\,i\,] = A[\,j\,] \text{ y } i < j \Rightarrow A'.\text{indexOf}(\,A[\,i\,]\,) < A'.\text{indexOf}(\,A[\,j\,]\,)$$

Resultado de un ordenamiento estable:

$C\ B_1\ B_2\ B_3\ A \Rightarrow A\ B_1\ B_2\ B_3\ C$

Resultado de un ordenamiento inestable:

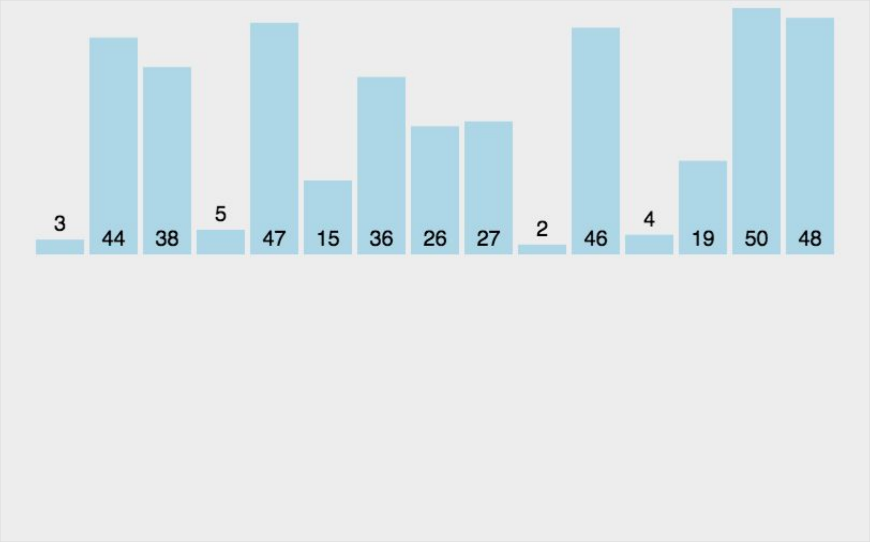
$C\ B_1\ B_2\ B_3\ A \Rightarrow A\ B_3\ B_2\ B_1\ C$

* El ordenamiento Quicksort puede llegar a ser inestable.

ED
ESTRUCTURAS DE DATOS

Mergesort - 1945 Von Neumann

```
mergesort ( arr [ ] ) :
    mergesort ( arr, 0, N - 1 )
```

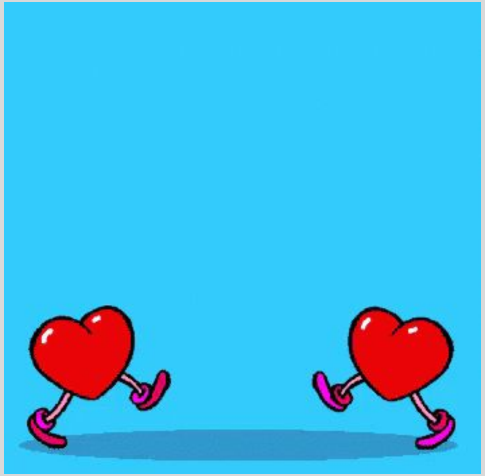



```
mergesort ( arr [ ], lo, hi ) :
    if ( hi ≤ lo ) return
    mid = lo + (hi - lo) / 2
    mergesort ( arr, lo, mid)
    mergesort ( arr, mid + 1, hi)
    merge ( arr, lo, mid, hi )
```

ED
ESTRUCTURAS DE DATOS

Mergesort - Algoritmo de mezcla

```
merge ( arr [ ], lo, mid, hi ) :
    i = lo
    j = mid + 1
    aux = arr.copia()
    for ( k = lo; k ≤ hi; k++ ) :
        if ( i > mid )                    a[ k ] = aux[ j++ ]
        else if ( j > hi )                a[ k ] = aux[ i++ ]
        else if ( aux[ j ] < aux[ i ] )   a[ k ] = aux[ j++ ]
        else                              a[ k ] = aux[ i++ ]
```



El algoritmo de mezcla requiere de memoria extra, pues tiene que crear una copia de la subcolección.

Ejercicio

Calcula la complejidad en **espacio** del algoritmo Mergesort:

```
mergesort ( arr [ ], lo, hi ) :
    if ( hi ≤ lo ) return
    mid = lo + (hi - lo) / 2
    mergesort ( arr, lo, mid)
    mergesort ( arr, mid + 1, hi)
    merge ( arr, lo, mid, hi )
```

Comparación entre algoritmos

	Tiempo promedio	Peor tiempo	Espacio promedio	Peor espacio
Bubblesort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
Selectionsort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
Insertionsort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	$O(n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n)$ *	$O(n)$ *

Enlace a un comparador de los algoritmos

ED
ESTRUCTURAS DE DATOS

Búsqueda binaria

Si tenemos una colección ordenada en la que podamos acceder al i-ésimo elemento en tiempo constante, podemos emplear búsqueda binaria para encontrar a un elemento en particular.

```
busquedaBinaria( array, elem, lo, hi):
    if ( lo > hi ) Fracaso
    mid = lo + (hi - lo) / 2
    if ( array[ mid ] == elem ) Éxito
    if ( array[ mid ] < elem )
        return busquedaBinaria( array, elem, lo, mid - 1 )
    else
        return busquedaBinaria( array, elem, mid + 1, hi )
```

ED
ESTRUCTURAS DE DATOS

Ejercicio

Calcula la complejidad en tiempo y espacio de hacer búsqueda binaria en una colección ordenada.

Manteniendo colecciones ordenadas

Si tenemos una colección ordenada y queremos seguir manteniéndola así, entonces al agregar o eliminar un elemento debemos preservar el orden. Para ello hay dos alternativas:

- Aplicar un algoritmo de ordenamiento después de cada inserción o eliminación.
  - Mala idea
- Crear un mecanismo propio en la colección para agregar y eliminar elementos preservando el orden.
  - Buena idea

Pregunta

Si tenemos un **arreglo ordenado**, ¿qué mecanismo recomiendas para preservar el orden después de una eliminación o inserción?

Pregunta

Si tenemos una **lista ordenada**, ¿qué mecanismo recomiendas para preservar el orden después de una eliminación o inserción?