Ficha 16

Ordenamiento y Búsqueda: Fundamentos

1.] Ordenamiento de un arreglo unidimensional (Algoritmo: Selección Directa).

Una operación muy común cuando se trabaja con arreglos unidimensionales es la de *ordenar* sus elementos, ya sea en secuencia de menor a mayor o de mayor a menor. Entre otras ventajas, la importancia del ordenamiento de un arreglo (y en general, de cualquier estructura de datos que admita algún tipo de ordenamiento) está en relación directa con la operación de *buscar* valores dentro de ella. Como se verá, pueden plantearse algoritmos que resultan más eficientes para buscar un valor en un arreglo *ordenado* que para buscarlo en uno desordenado.

Existen muchísimos algoritmos diferentes para proceder al ordenamiento de un arreglo de *n* componentes[1]. El estudio de todos y cada uno de ellos, sería una tarea ajena al propósito introductorio de esta Ficha (que está orientada a técnicas básicas, aunque analizaremos muchos de esos algoritmos en una Ficha posterior), y por lo tanto nos concentraremos en comprender el modo de funcionamiento de sólo uno de ellos, de naturaleza sencilla, conocido como *Ordenamiento de Selección Directa*. No obstante, aclaramos que incluso este sencillo método ofrece variantes y mejoras que no serán analizadas aquí.

Si se desea ordenar el arreglo de menor a mayor usando *selección directa*, la idea central es la adaptación y repetición sistemática del simple algoritmo que mostramos a continuación (suponemos que el arreglo a ordenar es *v* y que ya está cargado con valores de tipo *int*) [1]:

```
i = 0
for j in range(i+1, n):
    if v[i] > v[j]:
        v[i], v[j] = v[j], v[i]
```

Un rápido análisis muestra que el algoritmo anterior recorre el arreglo buscando el valor *menor* del mismo, y al terminar lo deja en la casilla indicada por la variable *i* (que en este caso vale cero). Se comienza asumiendo que el menor está en la casilla cuyo índice es *i = 0*, y

[1] Si se trata de mundos ordenados, no se puede menos que citar la película *Divergent* (en español conocida como *Divergente*) del año 2014, dirigida por *Neil Burger* y protagonizada por *Shailene Woodley*. A su vez, está basada en la primera parte de la trilogía de novelas juveniles escrita por *Veronica Roth*. El mundo ha sido devastado por la guerra y la sinrazón, y los sobrevivientes se reorganizaron en una sociedad estrictamente ordenada y dividida en clanes o facciones. Cada persona es asignada a una de esas facciones de acuerdo a sus aptitudes, y durante toda su vida contribuye al desarrollo de la sociedad desde esa facción. La protagonista Tris Prior descubre que no encaja en ninguna de las facciones (y por eso es "divergente") pero ese descubrimiento es peligroso porque altera el orden pre-establecido… En 2015 llegó al cine la segunda parte conocida como *The Divergent Series: Insurgent* (o *La Serie Divergente: Insurgente*) dirigida por *Robert Schwentke*. Y como era de esperar, la tercera parte se dividió en dos películas: *The Divergent Series: Allegiant* (*La Serie Divergente: Leal*) estrenada en 2016 y otra vez dirigida por *Robert Schwentke*, y *The Divergent Series: Ascendant* (*La Serie Divergente: Ascendente*) con la dirección anunciada de *Lee Toland Krieger* (fue agendada para 2017, pero aún en espera… y no se sabe si alguna vez la completarán).

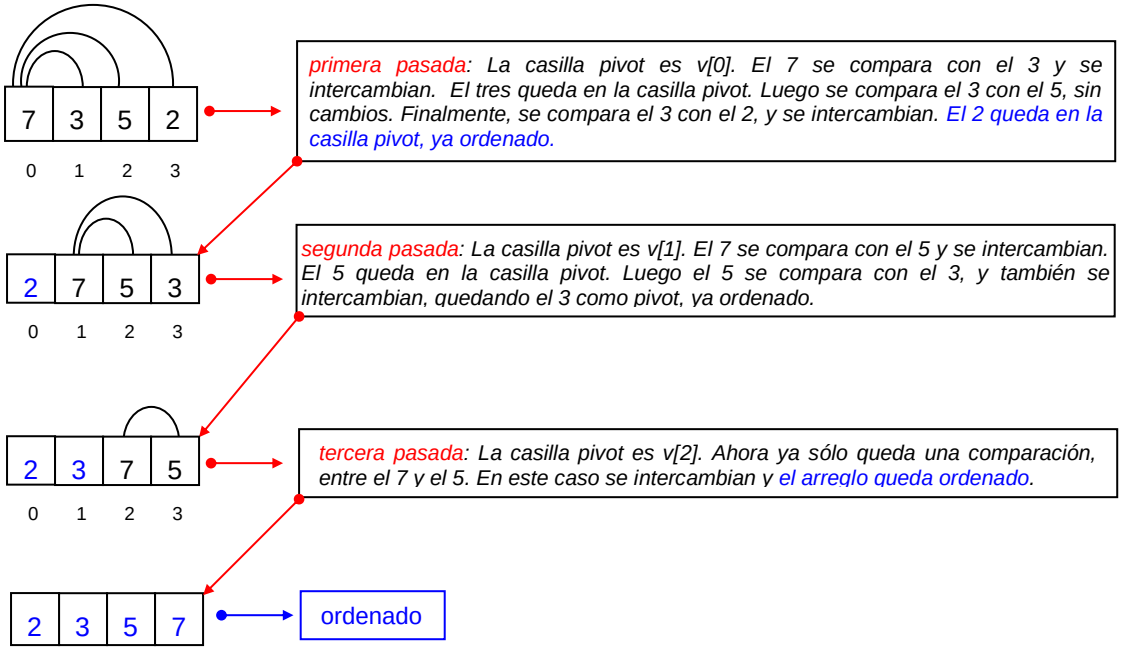
se recorre el resto del arreglo con *j* comenzando desde el valor *i+1*. Cada valor en la casilla *j* se compara con el valor en la casilla *i*. Si el valor en *v[i]* resulta mayor que *v[j]* entonces los valores se intercambian, y así se prosigue hasta que el ciclo controlado por *j* termina.

El algoritmo presentado *no ordena* el arreglo: sólo garantiza que el menor valor será colocado en la casilla con índice *i = 0*. Pero entonces sólo basta un pequeño cambio para que se ordene ***todo*** el arreglo: hacer que la variable *i* modifique su valor con *otro* ciclo, comenzando desde cero y terminando antes de llegar a la última casilla (para evitar que *j* comience valiendo un índice fuera de rango). La función *selection_sort()* que sigue (incluida en el módulo *arreglos.py* del proyecto *[F16] Ordenamiento y Búsqueda* que acompaña a esta Ficha) hace justamente eso:

```
def selection_sort(v):
    # ordenamiento por seleccion directa
    n = len(v)
    for i in range(n-1):
        for j in range(i+1, n):
            if v[i] > v[j]:
                v[i], v[j] = v[j], v[i]
```

Gráficamente, el proceso completo puede mostrarse así:

Figura 1: Esquema de trabajo completo del algoritmo de ordenamiento por Selección Directa.



El mismo proyecto *[F16] Ordenamiento y Búsqueda* incluye el programa *test01.py*, en el cual se carga por teclado y se ordena un arreglo usando esta función:

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
```

```
    return n


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglos de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    print('\nVector v:')
    arreglos.read(v)

    # ordenar el arreglo...
    c = arreglos.selection_sort(v)

    # mostrar el vector ordenado...
    print('\nEl vector ordenado es:', v)


# script principal...
if __name__ == '__main__':
    test()
```

2.] Búsqueda secuencial.

Muchas veces se presente el problema de determinar si cierto valor *x* está contenido o no en un arreglo *v* de *n* componentes. La convención suele ser que si se encuentra se informa en qué posición, y si no se encuentra se informa con un mensaje.

Si el arreglo está desordenado, o no se sabe nada acerca de su estado, la única forma inmediata de buscar un valor en él consiste en hacer una *búsqueda secuencial*: con un ciclo *for* se comienza con el primer elemento del arreglo. Si allí se encuentra el valor buscado, se detiene el proceso y se retorna el índice del componente que contenía al valor. Si allí no se encuentra el valor, se salta al componente siguiente y se repite el esquema. Si se llega al final del arreglo sin encontrar el valor buscado, lo cual ocurrirá cuando *i* (la variable de control del ciclo) sea igual al tamaño del arreglo, la función de búsqueda retorna el valor –1, a modo de indicador para avisar que la búsqueda falló [1]. El algoritmo de *búsqueda secuencial* en un arreglo se aplica en la función *linear_search()* que sigue (incluida en el módulo *arreglos.py* del proyecto *[F16] Ordenamiento y Búsqueda* que acompaña a esta Ficha):

```
def linear_search(v, x):
    # busqueda secuencial...
    for i in range(len(v)):
        if x == v[i]:
            return i

    return -1
```

El programa *test02.py* (en el mismo proyecto *[F16] Ordenamiento y Búsqueda*), carga un arreglo por teclado y procede luego a buscar un valor *x* en el mismo por medio de la función anterior:

```
__author__ = 'Cátedra de AED'

import arreglos
```

```
def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')

    return n


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    print('\nVector v:')
    arreglos.read(v)

    x =  int(input('Valor a buscar en el arrreglo: '))

    # aplicar búsqueda secuencial...
    ind = arreglos.linear_search(v, x)

    # avisar por pantalla el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')


# script principal...
if __name__ == '__main__':
    test()
```

3.] Búsqueda binaria.

Si el arreglo estuviera ordenado (por ejemplo de menor a mayor), se puede aplicar un método de búsqueda mucho más eficiente (en el sentido de su velocidad de ejecución) que la búsqueda secuencial ya vista, conocido como *Búsqueda Binaria*. La idea básica es la siguiente [1]: se usan dos índices auxiliares *izq* y *der* cuya función es la de marcar dentro del arreglo los límites del intervalo en donde se buscará el valor. Como al principio la búsqueda se hace en *todo* el vector, originalmente *izq* comienza valiendo *0* y *der* comienza valiendo *n-1* (siendo *n* la cantidad de componentes del arreglo). Dentro del intervalo marcado por *izq* y *der*, se toma el elemento central, cuyo índice *c* es:

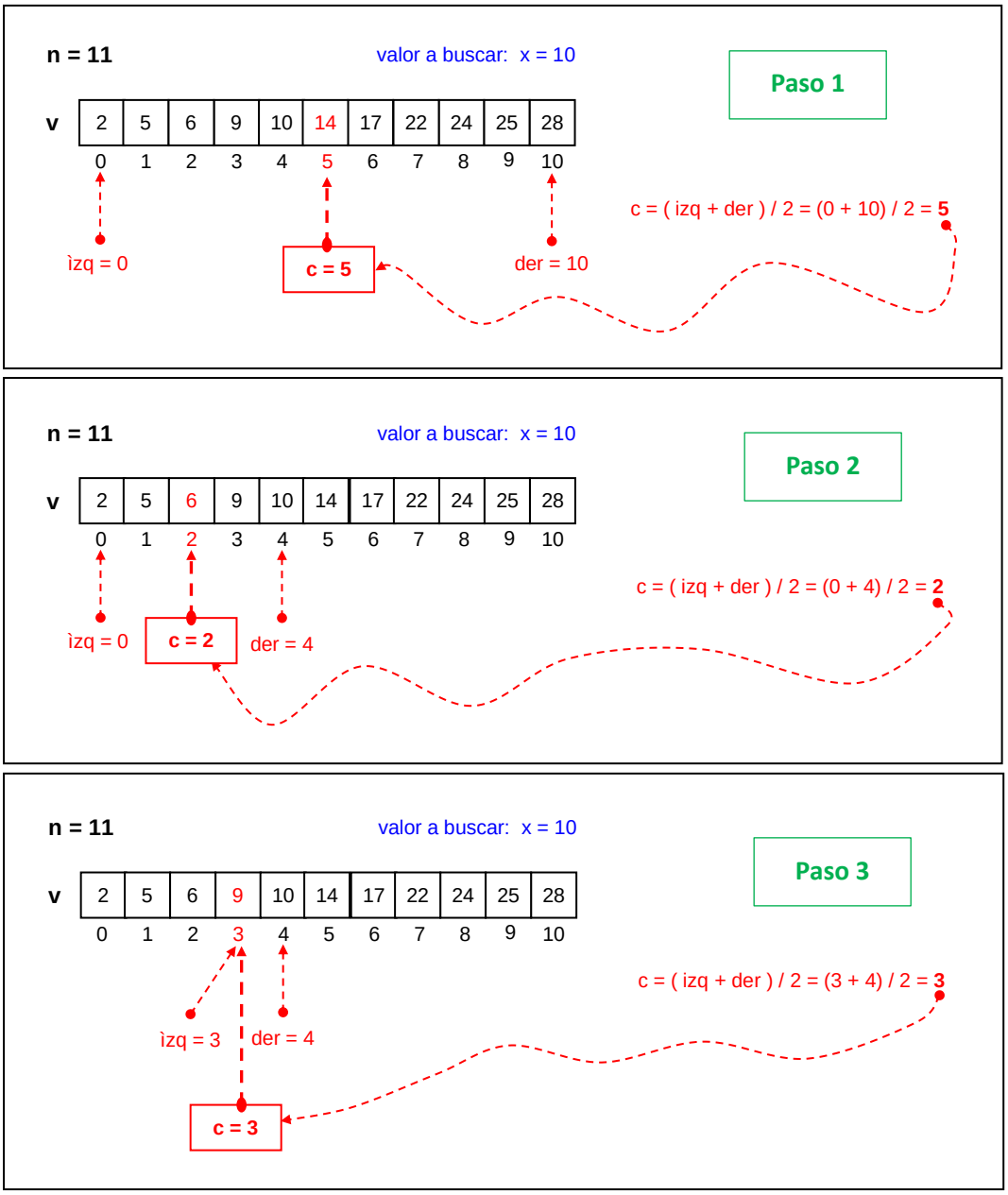
```
c  =  (izq + der) // 2
```

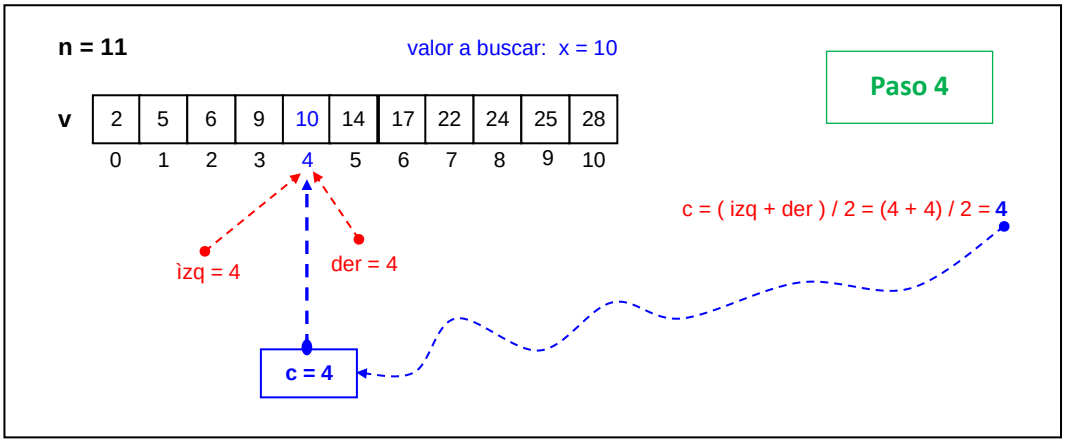
o sea, el promedio de los valores de *izq* y *der*. Luego de esto, se verifica si el valor contenido en *v[c]* coincide o no con el número buscado. Si coincide, se termina la búsqueda, y se retorna el valor de *c* para indicar la posición del número dentro del arreglo. Si no coincide, es entonces cuando se aprovecha que el arreglo está ordenado de menor a mayor: si el valor buscado *x* es menor que *v[c]*, entonces si *x* está en el arreglo debe estar a la izquierda de *v[c]* y por lo tanto, el nuevo intervalo de búsqueda debe ir desde el valor de *izq* hasta el valor de *c-1*. Entonces, se ajusta *der* para que valga el valor *c-1*, y se repite el proceso descripto. Si el valor *x* es mayor que *v[c]*, entonces la situación es la misma pero simétrica hacia la derecha, y debe ajustarse *izq* para valer *c+1*.

El proceso continúa hasta que se encuentre el valor (en cuyo caso se retorna el último valor *c* calculado), o hasta que el valor de *izq* se haga mayor que el de *der* (es decir, hasta que los índices se *crucen*), lo cual indicará que ya no quedan intervalos en los que buscar, y por lo tanto el valor *x* no estaba en el arreglo (y en este caso, la función que hace la búsqueda retornará el valor –1).

El proceso que se describió se denomina *búsqueda binaria* porque *parte* al arreglo en dos intervalos a partir del valor central, y a cada intervalo en otros dos, hasta dar con el valor o no poder generar nuevos intervalos. La eficiencia del método se basa en que se toma sólo uno de los dos intervalos para buscar al valor, y el otro se desecha por completo. En la primera partición, la mitad de los elementos del arreglo se desechan, en la segunda se desecha la cuarta parte (o sea, la mitad de la mitad), y así sucesivamente. Con muy pocas comparaciones, el valor es encontrado (si existe). La siguiente secuencia de gráficos ilustra el proceso completo:

Figura 2: Esquema de trabajo completo del algoritmo de Búsqueda Binaria.





La desventaja obvia es que el arreglo *debe estar ordenado*. Si se trabaja en un contexto donde el arreglo sufre cambios permanentes de contenido y debe ser ordenado periódicamente para facilitar la búsqueda, entonces se pierde la ventaja de la rapidez del algoritmo, pues se supera largamente con el tiempo que se pierde ordenando. En todo caso, la enseñanza final de toda esta cuestión es que existen herramientas algorítmicas de tipos y condiciones diversas, algunas muy buenas en ciertas condiciones; pero depende de la capacidad de análisis del programador el poder determinar en qué casos usar una u otra técnica.

El algoritmo de *búsqueda binaria* se aplica en la función *binary_search()* que sigue (incluida también en el módulo *arreglos.py* del proyecto *[F16] Ordenamiento y Búsqueda* que acompaña a esta Ficha):

```
def binary_search(v, x):
    # busqueda binaria... asume arreglo ordenado...
    izq, der = 0, len(v) - 1
    while izq <= der:
        c = (izq + der) // 2
        if x == v[c]:
            return c
        if x < v[c]:
            der = c - 1
        else:
            izq = c + 1
    return -1
```

El programa *test03.py* (en el mismo proyecto *[F16] Ordenamiento y Búsqueda*), crea un arreglo y procede luego a buscar un valor *x* en el mismo por medio de la función anterior. Observe que para facilitar que el arreglo *v* esté ordenado, en este programa ese arreglo no se carga por teclado sino que se genera su contenido a través de una función *generate()* que simplemente asigna en el arreglo una secuencia ordenada de múltiplos de 3 (dejamos su análisis para el estudiante):

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
```

```
    return n


def generate(v):
    # carga el arreglo con una secuencia ordenada simple...
    n = len(v)
    for i in range(n):
        v[i] = 3*i


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # generar el arreglo ordenado...
    generate(v)
    print('\nEl vector v fue creado con los siguientes valores:')
    print(v)

    x = int(input('\nValor a buscar en el arrreglo: '))

    # aplicar búsqueda secuencial...
    ind = arreglos.binary_search(v, x)

    # avisar por pantalla el resultado de la búsqueda...
    if ind >= 0:
        print('\nEstá en la casilla', ind)
    else:
        print('\nNo está en el arreglo')


# script principal...
if __name__ == '__main__':
    test()
```

4.] Fusión de arreglos ordenados.

Este proceso consiste en generar un *arreglo ordenado c* a partir de otros dos ya cargados y *ordenados previamente*, que designaremos como *a* (con *n* componentes) y *b* (con *m* componentes).

La idea es plantear una función *merge()* para recorrer en paralelo los arreglos *a* y *b*, usando respectivamente los índices *i* y *j*, que originalmente se inicializan en cero. Se toma el valor *a[i]* y se lo compara con *b[j]*. El menor de ambos valores es copiado en el arreglo *c*, en el primer componente libre (que se indica con la variable *k*) [1].

Si el menor fue *a[i]*, entonces se incrementa *i* en uno y el nuevo *a[i]* se compara con el mismo *b[j]* anterior. Si el menor fue *b[j]*, entonces se incrementa *j* en uno, y se compara el nuevo *b[j]* con el mismo valor *a[i]* anterior. En cualquiera de los dos casos, se incrementa también el índice *k* del arreglo *c*, para habilitar un nuevo componente en ese arreglo. El proceso continúa hasta que se llega al final de uno de los dos arreglos originales. O sea, cuando *i* llega a valer *n* (en cuyo caso se llegó al final de *a*), o cuando *j* llega a valer *m* (en cuyo caso se llegó al final de *b*).

Sin embargo, todavía quedan por procesar todos los elementos ubicados al final del arreglo que *aún no se terminó de recorrer*. Aquí la cuestión es simple: todos los valores que

quedaron en ese arreglo, se llevan sin más trámite al final del arreglo *c*, dado que esos elementos están ordenados y son mayores que todos los que ya se cargaron en *c*.

La función *merge()* que aplica este algoritmo se muestra a continuación, y está incluida en el módulo *arreglos.py* del proyecto *[F16] Ordenamiento y Búsqueda* que acompaña a esta Ficha:

```
def merge(a, b):
    # crear el tercer arreglo con lugar para n + m elementos...
    n, m = len(a), len(b)
    t = n + m
    c = t * [0]

    # aplicar proceso de fusión...
    i = k = j = 0
    while i < n and j < m:
        if a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
        k += 1

    # determinar cuál de los vectores (a o b) terminó primero...
    # ... apuntar con v al otro...
    v, pos = b, j
    if i < n:
        v, pos = a, i

    # copiar en el vector de salida todos los valores que
    # quedaban en el vector v...
    while pos < len(v):
        c[k] = v[pos]
        pos += 1
        k += 1

    # retornar el vector fusionado...
    return c
```

A su vez, el programa que mostramos más abajo genera los dos arreglos *a* y *b*, de tamaños *n* y *m* respectivamente, con secuencias previamente ordenadas (evitando tener que hacerlo por teclado), y luego invoca a la función *merge()* para proceder a la fusión (ver modelo *test05.py* en el proyecto *[F16] Ordenamiento y Búsqueda*):

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> a ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
    return n


def generate(v, factor=3):
    # carga el arreglo con una secuencia ordenada simple...
    n = len(v)
```

```
    for i in range(n):
        v[i] = factor*i


def test():
    # generar el primer vector...
    print('Vector a:')
    n = validate(0)
    a = n * [0]
    generate(a, 2)

    # generar el segundo vector...
    print('\nVector b:')
    m = validate(0)
    b = m * [0]
    generate(b, 5)

    # fusionar y obtener el tercer vector ordenado...
    c = arreglos.merge(a, b)

    # mostrar el vector fusionado...
    print('\nEl vector fusionado es:', c)

# script principal...
if __name__ == '__main__':
    test()
```

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[3] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.