

Ficha 24

Archivos: Aplicaciones Prácticas

1.] Introducción.

En esta Ficha no se agregan temas teóricos nuevos. El contenido completo se orienta al desarrollo de ejercicios y casos de análisis para integrar las estructuras de arreglos, registros y archivos que se han visto hasta ahora y mostrar distintas formas prácticas en que un archivo puede usarse para almacenar datos que estaban contenidos en otras estructuras.

En todos los ejercicios que siguen, se suponen situaciones directas en la que se requiere procesar datos que están en una estructura (por ejemplo, un arreglo de registros) y copiar parte de estos datos (o un subconjunto de ellos) a otra estructura (por ejemplo, un archivo). No se trata de situaciones completas en las que un archivo de registros podría requerir un procesamiento ABM, sino de aplicaciones inmediatas para la práctica en el uso combinado de distintas estructuras de datos.

Por lo tanto, en la gestión de los archivos de registros que se pedirá manejar, no será necesario prever un *campo de marcado lógico* para eventuales operaciones de *bajas lógicas*, ni aplicar las estrategias básicas y esenciales que soportan a un programa ABM en general: sólo se espera que se apliquen las estrategias generales para cumplir con el objetivo inmediato sugerido por cada enunciado.

Cada uno de los casos de análisis que siguen, se presenta en su propia sección o capítulo dentro de esta Ficha, para facilitar su identificación y agrupar mejor las explicaciones que acompañen a cada uno.

2.] Caso de análisis: Combinación de estructuras – Arreglo generado en forma ordenada.

El primer problema o caso propuesto incluye una novedad, como es la generación de un arreglo de registros pero de forma que al agregar un nuevo registro se mantenga ordenado el contenido del arreglo (en base al valor de un campo particular). El enunciado es el siguiente:

Problema 55.) *Un consultorio médico necesita un programa para gestionar los datos de sus pacientes. Por cada paciente, se deben almacenar los siguientes elementos: número de historia clínica (un número entero), nombre del paciente, fecha de la última visita (en días – otro número entero) y código de la enfermedad o problema registrado (un valor entre 0 y 9 incluidos). Los datos deben cargarse y almacenarse inicialmente en un **arreglo de registros**, a razón de un registro por paciente, el cual debe mantenerse en todo momento ordenado de menor a mayor de acuerdo al valor del número de historia clínica de los pacientes. El programa debe incluir un menú con las opciones siguientes:*

1. *Cargar el arreglo con los registros pedidos (recuerde: el arreglo debe mantenerse ordenado por historia clínica: cada registro debe insertarse en el lugar correcto cuando se agrega al arreglo).*
2. *Cargar por teclado un número entero **d**, y mostrar por pantalla los datos de todos los pacientes del arreglo que hayan asistido al consultorio por última vez en un período de **d** días o más.*
3. *Determinar si en el arreglo existe un paciente con número de historia clínica igual a **x**. Si existe, mostrar todos sus datos. Si no, dar un mensaje de error.*
4. *Mostrar todos los datos del arreglo.*
5. *Grabe todos los datos del arreglo en un archivo (para favorecer el desarrollo de los puntos que siguen, asegúrese de hacerlo de forma que cada registro se grabe por separado, uno por uno). El archivo debe ser creado si no existía, y todo dato que hubiese contenido debe ser eliminado si ya existía.*
6. *Mostrar el archivo generado en el punto anterior.*
7. *Usando el archivo creado en el punto 5, crear en memoria **otro** arreglo que contenga los registros de los pacientes cuyo código de enfermedad sea 8 o 9. Recuerde: debe crear **otro arreglo** de registros, y no eliminar ni modificar el original que se creó en el punto 1.*
8. *Mostrar el arreglo creado en el punto 7.*

Discusión y solución: El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej01.py* con el programa completo que resuelve este caso de análisis.

La parte inicial del programa contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Paciente*¹, las funciones `__init__()` y `to_string()` que tradicionalmente usamos para gestionar un registro, y un par de funciones de validación de carga por teclado:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Paciente:
    def __init__(self, hc, nom, fec, cod):
        self.hist_clinica = hc
        self.nombre = nom
```

¹ El cine ha explorado desde siempre el mundo de la medicina y nos ha entregado conmovedoras historias que constituyen un merecido reconocimiento a muchos médicos e investigadores que realizaron avances extraordinarios en el campo de la salud humana. En ese sentido, en 2004 la película para televisión cuyo título original es *Something The Lord Made* (dirigida por *Joseph Sargent* e interpretada por *Alan Rickman* y *Mos Def*) (en algunos sitios web el título fue traducido como *Una Creación del Señor*) está basada en la historia real del Dr. Alfred Blalock y su ayudante Vivien Thomas, quienes en la década de 1940 desarrollaron una técnica quirúrgica para salvar la vida de muchos niños recién nacidos que mostraban una malformación cardíaca congénita llamada "Tetralogía de Fallot" (o también "Síndrome de los Bebés Azules" debido a que esa malformación genera fallas en la irrigación sanguínea y por lo tanto en la oxigenación de los bebés, que se manifiesta en una coloración azulada en los labios y las puntas de los dedos). Sin la técnica de Blalock y Thomas, esos niños estaban condenados a morir a los pocos meses de nacidos. No dejen de verla...

```

        self.fecha = fec
        self.cod_problema = cod

def to_string(pac):
    r = ''
    r += '{:<25}'.format('Historia Clinica: ' + str(pac.hist_clinica))
    r += '{:<30}'.format('Nombre: ' + pac.nombre)
    r += '{:<20}'.format('Fecha: ' + str(pac.fecha))
    r += '{:<20}'.format('Problema: ' + str(pac.cod_problema))
    return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... se pidio mayor a', lim, '... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup) + ' : '))
        if n < inf or n > sup:
            print('\t\tError... cargue de nuevo...')
    return n

```

La carga por teclado del arreglo original p se hace mediante la función `cargar_arreglo_ordenado()`, la cual a su vez invoca a la función `add_in_order()` para insertar cada registro en la posición correcta del arreglo y mantenerlo ordenado de menor a mayor por número de historia clínica [1]:

```

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i
            break

    p[pos:pos] = [paciente]

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de pacientes...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los pacientes...')
    for i in range(n):
        print('Número de historia clínica...')
        hc = validar_mayor(0)

        nom = input('Nombre: ')

        print('Días transcurridos desde su última visita...')
        dias = validar_mayor(0)

        print('Código de enfermedad o problema...')
        cod = validar_intervalo(0, 9)

        paciente = Paciente(hc, nom, dias, cod)

```

```
        add_in_order(p, paciente)
        print()

    return p
```

La función `cargar_arreglo_ordenado()` comienza creando un arreglo `p` inicialmente vacío. Luego pide al operador que ingrese la cantidad de pacientes `n` que serán cargados y mediante un ciclo `for` se cargan los datos de cada uno, en forma tradicional y validando aquellos que lo requieran. Cuando todos los datos de un paciente han sido cargados, se crea un registro de tipo `Paciente` mediante la función `init()`, y en lugar de agregarlo directamente al arreglo con el conocido método `append()`, se invoca a en su lugar a la función `add_in_order()`.

Esta función toma dos parámetros: el arreglo de registros `p`, y el registro `paciente` que se acaba de crear. En la función `add_in_order()` se supone que el arreglo `p` en todo momento está ordenado de menor a mayor de acuerdo al número de historia clínica, y la idea es recorrer el arreglo, encontrar la posición en la que debería insertarse el nuevo registro *para el arreglo continúe ordenado*, y agregarlo en esa posición.

La idea es esencialmente simple: se recorre el arreglo `p` con un ciclo `for`, controlando que en cada casilla el valor `p[i].hist_clinica` sea menor que `paciente.hist_clinica`: esto es, continuar mientras el paciente que ya está en el arreglo tenga un número de historia clínica *menor* al número del paciente que se quiere insertar. Está claro que el nuevo paciente *no puede* insertarse delante de otro cuyo número sea *menor* al propio, pues se perdería el orden de menor a mayor. Si durante el recorrido se encuentra un registro `p[i]` cuyo número de historia clínica sea mayor (o incluso igual) al que se quiere agregar, entonces en ese momento se detiene el ciclo, guardando en una variable local `pos` el índice de la casilla que contenía al registro con número mayor.

Al salir del ciclo, la variable `pos` contiene el índice del paciente que debería quedar como el siguiente del que se quiere insertar (el primer paciente con número de historia clínica mayor al del paciente nuevo). Por lo tanto, el registro `paciente` debe agregarse justamente en la posición `pos`, corriendo hacia la derecha a todos los registros desde allí en adelante, lo cual sabemos que puede hacerse con la instrucción siguiente [2] [3]:

```
p[pos:pos] = [paciente]
```

Un detalle interesante es que si el registro `paciente` que se quiere agregar tiene un número de historia clínica *mayor al de todos los registros del arreglo p*, entonces el paciente debe agregarse al final del arreglo. Antes de comenzar el ciclo `for`, la variable `pos` se inicializa con el valor `n` (el tamaño del arreglo en ese momento). Si el ciclo `for` no encuentra ningún paciente con número de historia clínica menor al que se quiere agregar, se detendrá al llegar al final del arreglo sin modificar el valor de `pos`. Pero en este caso, `pos` quedará valiendo finalmente `n`, por lo que la instrucción

```
p[pos:pos] = [paciente]
```

agregará efectivamente el registro `paciente` al final del arreglo.

Note que la función `add_in_order()` efectivamente agrega un registro en un arreglo ordenado, de forma que el arreglo siga ordenado, pero para encontrar el *punto de inserción* (el lugar donde insertarse el nuevo registro) usa el algoritmo de búsqueda secuencial. Claramente esto podría mejorarse: en el contexto de este problema, el *arreglo está siempre*

ordenado por número de historia clínica, y por lo tanto, se puede aplicar el *algoritmo de búsqueda binaria* para encontrar el punto de inserción, en lugar del algoritmo de búsqueda secuencial. El tiempo de ejecución de esta búsqueda bajaría entonces de $O(n)$ a $O(\log(n))$, mejorando mucho el rendimiento si el arreglo fuese muy grande. La forma de hacer este ajuste se discute en la *Sección de Temas Avanzados* (ver *página 512*) de esta misma Ficha.

Las dos funciones que siguen muestran en pantalla el contenido del arreglo. La primera se llama *listado_por_dias()*, y toma como parámetro al arreglo *p* y a un número *d* que representa una cantidad de días. La función recorre el arreglo y muestra un *listado filtrado*: sólo los registros en los que el campo *fecha* sea mayor o igual al valor *d* recibido como parámetro. La segunda función se llama *mostrar_arreglo()* y también toma dos parámetros: el arreglo *p* y una variable llamada *mensaje* que contiene una cadena de caracteres con el título que debe mostrarse antes del listado del contenido del arreglo. La función recorre ese arreglo y muestra su contenido completo (sin filtro):

```
def listado_por_dias(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    d = int(input('Días a comprobar desde la última visita: '))
    print('Pacientes con', d, 'o más días desde la última visita')
    for paciente in p:
        if paciente.fecha >= d:
            print(to_string(paciente))

    print()

def mostrar_arreglo(p, mensaje='Contenido:'):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print(mensaje)
    for paciente in p:
        print(to_string(paciente))

    print()
```

La que sigue es la función *buscar()*, que toma como parámetro al arreglo *p* y un número *x*, y aplica el ya conocido algoritmo de *búsqueda binaria* para determinar si existe un registro cuyo número de historia clínica coincida con *x*, mostrando su contenido en caso de encontrarlo o un mensaje aclaratorio en caso contrario. En este caso, *se puede aplicar sin dudar* el algoritmo de *búsqueda binaria* [1] porque el arreglo *p* en todo momento está ordenado justamente por el campo *hist_clinica*:

```
def buscar(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Número de historia clínica a buscar: '))

    n = len(p)
    izq, der = 0, n - 1
    while izq <= der:
        c = (izq + der) // 2
```

```

    if p[c].hist_clinica == x:
        print('Paciente encontrado...')
        print(to_string(p[c]))
        print()
        return

    if x < p[c].hist_clinica:
        der = c - 1
    else:
        izq = c + 1

print('No hay un paciente con ese número de historia clínica')
print()

```

Las funciones *crear_archivo()* y *mostrar_archivo()* son directas: la primera toma como parámetro el arreglo *p* y graba con la función *pickle.dump()* su contenido completo, registro por registro, en el archivo cuyo nombre físico viene en la variable global *FD*. La grabación se hace registro por registro debido a que en el punto 7 se pide recuperar *una parte* de esos registros para almacenarlos en un segundo arreglo. Si el arreglo original hubiese sido almacenado en forma directa con una sola instrucción *pickle.dump()*, entonces para generar el segundo arreglo se tendría que volver a recuperar con *pickle.load()* el arreglo original completo, y crear el segundo a partir de este (ambas técnicas son válidas). La segunda función se llama *mostrar_archivo()* y también es directa: recorre, lee y muestra registro por registro el contenido completo del archivo *FD*:

```

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Grabando todos los datos en el archivo', FD, '...')
    m = open(FD, 'wb')
    for paciente in p:
        pickle.dump(paciente, m)

    m.close()
    print('... hecho')
    print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        print(to_string(pac))

    m.close()
    print()

```

Sólo nos queda la función *crear_segundo_arreglo()*, que toma el archivo generado con *crear_archivo()*, y crea a partir de este un segundo arreglo conteniendo solamente los registros de pacientes cuyo código de problema sea 8 o 9, retornando luego el arreglo creado:

```
def crear_segundo_arreglo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    p2 = []
    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Creando el segundo vector desde el archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        if pac.cod_problema in [8, 9]:
            p2.append(pac)

    m.close()
    print('... hecho')
    print()

    return p2
```

La función *main()* es la que despliega y gestiona el menú de opciones y es el punto de entrada del programa completo que se muestra a continuación:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Paciente:
    def __init__(self, hc, nom, fec, cod):
        self.hist_clinica = hc
        self.nombre = nom
        self.fecha = fec
        self.cod_problema = cod

def to_string(pac):
    r = ''
    r += '{:<25}'.format('Historia Clinica: ' + str(pac.hist_clinica))
    r += '{:<30}'.format('Nombre: ' + pac.nombre)
    r += '{:<20}'.format('Fecha: ' + str(pac.fecha))
    r += '{:<20}'.format('Problema: ' + str(pac.cod_problema))
    return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... se pidio mayor a', lim, '... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
```

```

while n < inf or n > sup:
    n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup)+ ' : '))
    if n < inf or n > sup:
        print('\t\tError... cargue de nuevo...')
return n

def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i
            break

    p[pos:pos] = [paciente]

'''
# Misma función, pero con búsqueda binaria...
def add_in_order(p, paciente):
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == paciente.hist_clinica:
            pos = c
            break

        if paciente.hist_clinica < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
        pos = izq

    p[pos:pos] = [paciente]
'''

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de pacientes...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los pacientes...')
    for i in range(n):
        print('Número de historia clínica...')
        hc = validar_mayor(0)

        nom = input('Nombre: ')

        print('Días transcurridos desde su última visita...')
        dias = validar_mayor(0)

        print('Código de enfermedad o problema...')
        cod = validar_intervalo(0, 9)

        paciente = Paciente(hc, nom, dias, cod)
        add_in_order(p, paciente)
        print()

    return p

```



```
def listado_por_dias(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    d = int(input('Días a comprobar desde la última visita: '))
    print('Listado de pacientes con', d, 'o más días desde la última visita')
    for paciente in p:
        if paciente.fecha >= d:
            print(to_string(paciente))

    print()

def mostrar_arreglo(p, mensaje='Contenido:'):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print(mensaje)
    for paciente in p:
        print(to_string(paciente))

    print()

def buscar(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Número de historia clínica a buscar: '))

    n = len(p)
    izq, der = 0, n - 1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == x:
            print('Paciente encontrado...')
            print(to_string(p[c]))
            print()
            return

        if x < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    print('No hay un paciente registrado con ese número de historia clínica')
    print()

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Grabando todos los datos en el archivo', FD, '...')
    m = open(FD, 'wb')
    for paciente in p:
        pickle.dump(paciente, m)
```

```
m.close()
print('... hecho')
print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        print(to_string(pac))

    m.close()
    print()

def crear_segundo_arreglo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    p2 = []
    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Creando el segundo vector desde el archivo', FD, '...')
    while m.tell() < tbm:
        pac = pickle.load(m)
        if pac.cod_problema in [8, 9]:
            p2.append(pac)

    m.close()
    print('... hecho')
    print()

    return p2

def main():
    global FD
    FD = 'pacientes.med'

    p, p2 = [], []
    op = 0
    while op != 9:
        print('Consultorio Dr. Matasanos')
        print('  1. Registrar pacientes en forma ordenada')
        print('  2. Listado de pacientes según días de su última visita')
        print('  3. Buscar un paciente por historia clínica')
        print('  4. Listado completo de pacientes')
        print('  5. Creación de un archivo con todos los datos registrados')
        print('  6. Mostrar el archivo')
        print('  7. Crear arreglo desde el archivo (código de enfermedad 8 o 9)')
        print('  8. Mostrar el arreglo (código de enfermedad 8 o 9)')
        print('  9. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
```

```
print()

if op == 1:
    p = cargar_arreglo_ordenado()

elif op == 2:
    listado_por_dias(p)

elif op == 3:
    buscar(p)

elif op == 4:
    mostrar_arreglo(p, 'Listado completo de pacientes registrados')

elif op == 5:
    crear_archivo(p)

elif op == 6:
    mostrar_archivo()

elif op == 7:
    p2 = crear_segundo_arreglo()

elif op == 8:
    mostrar_arreglo(p2, 'Pacientes (código de enfermedad 8 o 9)')

elif op == 9:
    pass

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

3.] Caso de análisis: Gestión de un archivo en forma directa.

Se propone ahora otro ejercicio a modo de caso de análisis para trabajar en clase, sobre manejo de archivos. La idea es mostrar una aplicación práctica que trabaje *directamente* sobre archivos, para resaltar que *no es obligatorio mover y/o copiar datos del archivo a un arreglo o viceversa...* Es correcto dominar las técnicas para mover datos de una estructura a la otra, pero también se debe comprender que operar directamente con los datos de un archivo es muy común, y muchas veces es lo único que se necesita hacer [1]. El enunciado del ejercicio propuesto es el siguiente:

Problema 56.) *La oficina regional de la Junta Electoral Provincial ha pedido un programa que almacene en un archivo los datos del padrón electoral de cierta localidad. Por cada persona habilitada para votar en esa localidad, se guarda su dni, su nombre, su edad y un indicador del sexo de esa persona ('v' para varón o 'm' para mujer). El programa debe incluir un menú de opciones que permita:*

1. *Cargar datos en el archivo, pero cuidando que no se repita ninguna persona dentro del mismo.*
2. *Mostrar los datos del archivo completo.*
3. *Determinar si en el archivo existe un votante con número de dni igual a x. Si existe, mostrar todos sus datos. Si no, informar que no existe.*
4. *Determinar cuántos votantes son varones y cuántos son mujeres en el archivo.*

5. *Genere un segundo archivo, que contenga sólo los datos de los votantes mayores a 70 años.*
6. *Mostrar el archivo generado en el punto anterior.*

Discusión y solución: El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej02.py* con el programa completo que resuelve este caso de análisis.

La parte inicial del programa como siempre, contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Votante*, las funciones *__init__()* y *to_string()* que tradicionalmente usamos para gestionar un registro, y un par de funciones de validación de carga por teclado:

```
import io
import pickle
import os.path

__author__ = 'Catedra de AED'

class Votante:
    def __init__(self, dn, nom, ed, sex):
        self.dni = dn
        self.nombre = nom
        self.edad = ed
        self.sexo = sex

def to_string(vot):
    sx = 'Hombre'
    if vot.sexo == 'm':
        sx = 'Mujer'

    r = ''
    r += '{:<20}'.format('DNI: ' + str(vot.dni))
    r += '{:<30}'.format('Nombre: ' + vot.nombre)
    r += '{:<20}'.format('Edad: ' + str(vot.edad))
    r += '{:<20}'.format('Sexo: ' + sx)
    return r

def validar_dni(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_edad(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Error... cargue de nuevo...')
    return n

def validar_sexo(caracteres):
```

```

c = ' '
while c not in caracteres:
    c = input('Letras válidas ' + str(caracteres) + ': ')
    if c not in caracteres:
        print('\t\tError... letra no válida... cargue de nuevo...')
return c

```

La función *cargar_archivo()* es la encargada de las operaciones de altas de nuevos registros en el archivo. Como se pide controlar que no se repitan los registros (es decir, controlar que no se graben dos registros con el mismo *dni*), se usa además la función *buscar()* para recorrer el archivo y controlar si un número de *dni* está ya registrado o no. Ambas funciones son prácticamente iguales a las que se presentaron en la *Ficha 23* para realizar altas y búsquedas, por lo que dejamos su análisis y repaso para el estudiante:

```

def buscar(m, dni):
    global FD1
    t = os.path.getsize(FD1)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        vot = pickle.load(m)
        if vot.dni == dni:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion

def cargar_archivo():
    global FD1
    m = open(FD1, 'a+b')

    print()
    print('DNI del votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)
    while dni != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, dni)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            print('Edad...')
            edad = validar_edad(0, 120)

            print('Sexo ("v": varón - "m": mujer)...')
            sex = validar_sexo(['V', 'v', 'M', 'm'])

            vot = Votante(dni, nom, edad, sex)

            # ...lo grabamos...
            pickle.dump(vot, m)

            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE

```

```

        # grabe en disco el registro...
        m.flush()
        print('Registro grabado en el archivo...')
    else:
        print('DNI repetido... alta rechazada...')

    print()
    print('Otro dni de votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)

m.close()
print()

```

La función *mostrar_archivo()* muestra en forma completa el contenido de un archivo. Como el problema sugiere mostrar dos archivos diferentes a través de dos opciones del menú, y dado que esos archivos contienen registros del mismo tipo, la función *mostrar_archivo()* toma un parámetro *FD* con el nombre físico del archivo a mostrar y abre, lee y muestra registro por registro ese archivo. La única diferencia entre esta función y otras que hemos visto para recorrer y mostrar un archivo, es que en esas funciones el nombre físico del archivo se suponía alojado en una *variable global*, y ahora ese nombre *viene como parámetro* (justamente, para poder usar la función con archivos de entrada diferentes):

```

def mostrar_archivo(FD):
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        print(to_string(vot))

    m.close()
    print()

```

La función *buscar_votante()* carga por teclado el número de *dni* de un votante, y utiliza la ya citada función *buscar()* para determinar si el archivo contiene o no a ese votante. Si existía, la función *buscar_votante()* simplemente se reposiciona con *seek()* [2] [3] en el byte donde comienza el registro, lo lee y finalmente lo muestra. Y en caso de no existir, sólo se avisa con un mensaje:

```

def buscar_votante():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    m = open(FD1, 'rb')

    print('DNI del votante a buscar: ')
    dni = validar_dni(0)
    pos = buscar(m, dni)
    if pos != -1:
        m.seek(pos, io.SEEK_SET)

```

```

        vot = pickle.load(m)
        print('Votante encontrado,...')
        print(to_string(vot))

    else:
        print('Ese votante no está registrado...')

m.close()
print()

```

La función *cantidad_por_sexo()* abre el archivo en modo de sólo lectura, lo lee registro por registro, y va contando en dos contadores cuántos de los votantes son varones y cuántos mujeres. El proceso es directo, y dejamos su análisis para alumno:

```

def cantidad_por_sexo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')

    cv, cm = 0, 0
    while m.tell() < tbm:
        vot = pickle.load(m)
        if vot.sexo == 'v':
            cv += 1
        else:
            cm += 1

    m.close()

    print('Cantidad de votantes varones:', cv)
    print('Cantidad de votantes mujeres:', cm)
    print()

```

La última función se llama *crear_segundo_archivo()* y se usa para generar un segundo archivo que contenga una copia de todos los registros del archivo original que correspondan a votantes con más de 70 años de edad. El nombre de ambos archivos viene dado respectivamente por las variables globales *FD1* y *FD2*. El original se abre en modo de sólo lectura, mientras que el nuevo archivo se abre en modo 'wb'. A medida que se lee un registro del archivo original, se chequea si el campo edad del mismo es mayor a 70, y en caso de serlo, simplemente se procede a grabarlo en el segundo archivo:

```

def crear_segundo_archivo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')
    s = open(FD2, 'wb')

    print('Creando archivo', FD2, 'con mayores a 70 años...')
    while m.tell() < tbm:

```

```

        vot = pickle.load(m)
        if vot.edad > 70:
            pickle.dump(vot, s)

    m.close()
    s.close()
    print('... hecho')
    print()

```

El programa completo que mostramos a continuación, incluye la función *main()* para manejar el menú de opciones, y esa función es el punto de entrada del programa:

```

import io
import pickle
import os.path

__author__ = 'Catedra de AED'

class Votante:
    def __init__(self, dn, nom, ed, sex):
        self.dni = dn
        self.nombre = nom
        self.edad = ed
        self.sexo = sex

def to_string(vot):
    sx = 'Hombre'
    if vot.sexo == 'm':
        sx = 'Mujer'

    r = ''
    r += '{:<20}'.format('DNI: ' + str(vot.dni))
    r += '{:<30}'.format('Nombre: ' + vot.nombre)
    r += '{:<20}'.format('Edad: ' + str(vot.edad))
    r += '{:<20}'.format('Sexo: ' + sx)
    return r

def validar_dni(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_edad(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('Error... cargue de nuevo...')
    return n

def validar_sexo(caracteres):
    c = ' '
    while c not in caracteres:

```



```

        c = input('Letras válidas ' + str(caracteres) + ': ')
        if c not in caracteres:
            print('\t\tError... letra no válida... cargue de nuevo...')
        return c

def buscar(m, dni):
    global FD1
    t = os.path.getsize(FD1)

    fp_inicial = m.tell()
    m.seek(0, io.SEEK_SET)

    posicion = -1
    while m.tell() < t:
        fp = m.tell()
        vot = pickle.load(m)
        if vot.dni == dni:
            posicion = fp
            break

    m.seek(fp_inicial, io.SEEK_SET)
    return posicion

def cargar_archivo():
    global FD1
    m = open(FD1, 'a+b')

    print()
    print('DNI del votante a registrar (cargue 0 para salir): ')
    dni = validar_dni(-1)
    while dni != 0:
        # buscamos el registro con ese legajo...
        pos = buscar(m, dni)
        if pos == -1:
            # no estaba repetido... lo cargamos por teclado...
            nom = input('Nombre: ')

            print('Edad...')
            edad = validar_edad(0, 120)

            print('Sexo ("v": varón - "m": mujer)...')
            sex = validar_sexo(['V', 'v', 'M', 'm'])

            vot = Votante(dni, nom, edad, sex)

            # ...lo grabamos...
            pickle.dump(vot, m)

            # ...volcamos el buffer de escritura
            # para que el sistema operativo REALMENTE
            # grabe en disco el registro...
            m.flush()

            print('Registro grabado en el archivo...')
        else:
            print('DNI repetido... alta rechazada...')

    print()

```

```
        print('Otro dni de votante a registrar (cargue 0 para salir): ')
        dni = validar_dni(-1)

    m.close()
    print()

def mostrar_archivo(FD):
    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        print(to_string(vot))

    m.close()
    print()

def buscar_votante():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
    m = open(FD1, 'rb')

    print('DNI del votante a buscar: ')
    dni = validar_dni(0)
    pos = buscar(m, dni)
    if pos != -1:
        m.seek(pos, io.SEEK_SET)
        vot = pickle.load(m)
        print('Votante encontrado,..')
        print(to_string(vot))

    else:
        print('Ese votante no está registrado...')

    m.close()
    print()

def cantidad_por_sexo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    print()
```

```
tbm = os.path.getsize(FD1)
m = open(FD1, 'rb')

cv, cm = 0, 0
while m.tell() < tbm:
    vot = pickle.load(m)
    if vot.sexo == 'v':
        cv += 1
    else:
        cm += 1

m.close()

print('Cantidad de votantes varones:', cv)
print('Cantidad de votantes mujeres:', cm)
print()

def crear_segundo_archivo():
    global FD1

    if not os.path.exists(FD1):
        print('El archivo', FD1, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD1)
    m = open(FD1, 'rb')
    s = open(FD2, 'wb')

    print('Creando archivo', FD2, 'con datos de mayores a 70 años...')
    while m.tell() < tbm:
        vot = pickle.load(m)
        if vot.edad > 70:
            pickle.dump(vot, s)

    m.close()
    s.close()
    print('... hecho')
    print()

def main():
    global FD1, FD2
    FD1 = 'votantes.vot'
    FD2 = 'mayores.vot'

    op = 0
    while op != 7:
        print('Padrón electoral')
        print('  1. Registrar votantes en el padrón')
        print('  2. Listado de votantes')
        print('  3. Buscar un votante por dni')
        print('  4. Cantidad de varones y mujeres')
        print('  5. Crear archivo con votantes mayores a 70 años')
        print('  6. Mostrar el archivo de votantes mayores a 70 años')
        print('  7. Salir')
        op = int(input('\t\tIngrese número de la opción elegida: '))
        print()

        if op == 1:
```

```
        cargar_archivo()

    elif op == 2:
        mostrar_archivo(FD1)

    elif op == 3:
        buscar_votante()

    elif op == 4:
        cantidad_por_sexo()

    elif op == 5:
        crear_segundo_archivo()

    elif op == 6:
        mostrar_archivo(FD2)

    elif op == 7:
        pass

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

4.] Caso de análisis: Combinación de estructuras y uso de una matriz de conteo.

Este ejercicio vuelve a la combinación práctica de estructuras de datos diversas, pero ahora agregando también una *matriz de conteo*. El enunciado es el siguiente:

Problema 57.) *Una editorial encargada de la publicación de una revista científica ha solicitado que se desarrolle un programa para gestionar su operatoria. Se deben almacenar en un arreglo unidimensional (un vector) los datos relacionados con los artículos disponibles para su publicación (cargar por teclado la cantidad n de artículos). Cada artículo tiene los siguientes datos: código (int), título (cadena), cantidad de páginas, tipo de artículo (puede ser un valor entre 0 y 9 identificando el campo de aplicación) y el idioma en que está escrito (un valor entre 0 y 5). Se pide un programa controlado por menú de opciones que permita:*

1. *Cargar por teclado el arreglo de artículos, que debe quedar ordenado alfabéticamente de acuerdo al título de los artículos. Alternativamente, la carga puede hacerse en forma automática, generando en forma aleatoria los valores a contener en cada registro. Además, verificar que ningún artículo aparezca repetido en el arreglo (no permita dos artículos con el mismo título).*
2. *Mostrar el arreglo completo.*
3. *Cargar por teclado el título de un artículo y determinar si existe alguno con ese título. Mostrar sus datos si existe, o informar que no existe.*
4. *Cargar por teclado el código de un artículo y determinar si existe alguno con ese código. Mostrar sus datos si existe, o informar que no existe.*
5. *Determinar la cantidad de artículos por tipo e idioma que hay en el arreglo (es decir, cuántos artículos tipo 0 está escritos en el idioma 0, cuántos tipo 0 están en el idioma 1, y así sucesivamente... con un total de $10 \times 6 = 60$ contadores).*

6. *Generar un archivo que contenga todos los datos de los artículos cuya cantidad de páginas sea superior a 10.*
7. *Mostrar el archivo completo.*

Discusión y solución: El proyecto [F24] Archivos - Aplicaciones que acompaña a esta Ficha contiene un modelo *ej03.py* con el programa completo que resuelve este caso de análisis.

Como de costumbre, la parte inicial del programa contiene las instrucciones para importar los módulos que serán necesarios, la declaración del registro *Articulo*, las funciones *init()* y *display()* para gestionar un registro, y las funciones de validación de carga por teclado.

La novedad es que ahora aparece una función *validar_titulo()* para verificar que el título del artículo propuesto no esté ya cargado en el arreglo. Y como el arreglo se genera de forma que quede ordenado por *título*, entonces la búsqueda del *título* en el vector se hace con la función *busqueda_binaria()*, que aplica justamente el algoritmo de *busqueda binaria*. La misma función se usa luego para buscar un artículo dado el título del mismo, y mostrarlo en pantalla.

Como el enunciado sugiere que también se deberá buscar un artículo pero dado ahora su *código* (y no su *título*), entonces el programa incluye también una función llamada *busqueda_secuencial()* para buscar secuencialmente ese *código* (ya que en ese caso, la *busqueda binaria* no es aplicable: el arreglo no está ordenado por código...)

La carga del arreglo se hace mediante la función *cargar_arreglo_ordenado()*, la cual carga por teclado los datos de cada registro, y añade cada registro al arreglo mediante la función *add_in_order()*, que a su vez busca el punto de inserción aplicando también *busqueda binaria* (en forma similar a lo expuesto en la sección de *Temas Avanzados*).

La función *conteo()* aplica una *matriz de conteos* para determinar cuántos artículos había de cada uno de los 10 tipos posibles y escritos en cada uno de los 6 idiomas disponibles. La matriz *cant* se crea con 10 filas (una por cada *tipo*) y 6 columnas (una por cada *idioma*), de forma que cada casillero valga inicialmente 0. Se recorre el arreglo de artículos, y por cada registro se toma el valor del campo *tipo* (lo cual da el número de fila *f* de su contador en la matriz) y el valor del campo *idioma* (que da el número de columna *c*). Luego se accede en forma directa al casillero *cant[f][c]* y se incrementa en uno su valor. Al finalizar, se recorre la matriz con un par de ciclos anidados, y se muestra adecuadamente cada casillero que sea diferente de cero [1].

Las funciones *crear_archivo()* y *mostrar_archivo()* no agregan novedad alguna en cuanto a las estrategias ya vistas para procesar un archivo, por lo que se deja su análisis para el estudiante. El programa completo se ve a continuación:

```
import pickle
import os.path

__author__ = 'Catedra de AED'

class Articulo:
    def __init__(self, cod, tit, pg, tp, id):
        self.codigo = cod
        self.titulo = tit
        self.paginas = pg
        self.tipo = tp
```

```

        self.idioma = id

def to_string(art):
    r = ''
    r += '{:<30}'.format('Titulo: ' + art.titulo)
    r += '{:<20}'.format('Codigo: ' + str(art.codigo))
    r += '{:<20}'.format('Paginas: ' + str(art.paginas))
    r += '{:<20}'.format('Tipo: ' + str(art.tipo))
    r += '{:<20}'.format('Idioma: ' + str(art.idioma))
    return r

def validar_mayor(lim):
    n = lim - 1
    while n <= lim:
        n = int(input('Valor mayor a ' + str(lim) + ' por favor: '))
        if n <= lim:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_intervalo(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor entre ' + str(inf) + ' y ' + str(sup) + ': '))
        if n < inf or n > sup:
            print('\t\tError... cargue de nuevo...')
    return n

def validar_titulo(p):
    tit = ''
    while tit == '' or busqueda_binaria(p, tit) != -1:
        tit = input('Título: ')
        if tit == '' or busqueda_binaria(p, tit) != -1:
            print('Nombre repetido o vacío... cargue de nuevo...')

    return tit

def busqueda_binaria(p, tit):
    # busca por título: algoritmo de búsqueda binaria...
    # ...el arreglo está ordenado por título...
    n = len(p)
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2

        if p[c].titulo == tit:
            return c

        if tit < p[c].titulo:
            der = c - 1
        else:
            izq = c + 1

    return -1

def busqueda_secuencial(p, cod):

```

```
# busca por código: algoritmo de búsqueda secuencial...
# ...el arreglo no está ordenado por código...
n = len(p)
for i in range(n):
    if p[i].codigo == cod:
        return i

return -1

def add_in_order(p, articulo):
    # inserta un registro en el arreglo, en forma ordenada...
    # ...pero aplicando búsqueda binaria para encontrar
    # el punto de inserción...
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2

        if p[c].titulo == articulo.titulo:
            pos = c
            break

        if articulo.titulo < p[c].titulo:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
        pos = izq

    p[pos:pos] = [articulo]

def cargar_arreglo_ordenado():
    p = []
    print('Cantidad de articulos...')
    n = validar_mayor(0)

    print()
    print('Ingrese los datos de los articulos...')
    for i in range(n):
        tit = validar_titulo(p)

        print('Código...')
        cod = validar_mayor(0)

        print('Cantidad de páginas...')
        pg = validar_mayor(0)

        print('Tipo...')
        tp = validar_intervalo(0, 9)

        print('Idioma...')
        id = validar_intervalo(0, 5)

        articulo = Articulo(cod, tit, pg, tp, id)
        add_in_order(p, articulo)
    print()
```

```
    return p

def mostrar_arreglo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Listado de artículos disponibles')
    for articulo in p:
        print(to_string(articulo))

    print()

def buscar_titulo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = input('Título a buscar: ')
    pos = busqueda_binaria(p, x)

    if pos != -1:
        print('Articulo encontrado...')
        print(to_string(p[pos]))
    else:
        print('No hay un articulo con ese título')

    print()

def buscar_codigo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    x = int(input('Código a buscar: '))
    pos = busqueda_secuencial(p, x)

    if pos != -1:
        print('Articulo encontrado...')
        print(to_string(p[pos]))
    else:
        print('No hay un articulo con ese código')

    print()

def conteo(p):
    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    cf, cc = 10, 6
    cant = [cc*[0] for f in range(cf)]
```



```
for art in p:
    f = art.tipo
    c = art.idioma
    cant[f][c] += 1

print('Cantidad de artículos por tipo e idioma...')
for f in range(cf):
    for c in range(cc):
        if cant[f][c] != 0:
            print('Tipo', f, 'Idioma', c, 'Cantidad:', cant[f][c])

def crear_archivo(p):
    global FD

    if len(p) == 0:
        print('No hay datos cargados...')
        print()
        return

    print('Cantidad de páginas a controlar...')
    cp = validar_mayor(0)

    print('Grabando artículos con más de', cp, 'en el archivo', FD, '...')
    m = open(FD, 'wb')
    for art in p:
        if art.paginas > cp:
            pickle.dump(art, m)

    m.close()
    print('... hecho')
    print()

def mostrar_archivo():
    global FD

    if not os.path.exists(FD):
        print('El archivo', FD, 'no existe...')
        print()
        return

    tbm = os.path.getsize(FD)
    m = open(FD, 'rb')

    print('Contenido del archivo', FD, '...')
    while m.tell() < tbm:
        art = pickle.load(m)
        print(to_string(art))

    m.close()
    print()

def main():
    global FD
    FD = 'articulos.edi'

    p = []
    op = 0
    while op != 8:
```

```

print('Editorial UTN')
print('  1. Registrar articulos ordenados por título')
print('  2. Listado completo de articulos')
print('  3. Buscar un articulo por título')
print('  4. Buscar un articulo por código')
print('  5. Conteo por tipo e idioma')
print('  6. Crear archivo desde el arreglo (cantidad de páginas)')
print('  7. Mostrar el archivo')
print('  8. Salir')
op = int(input('\t\tIngrese número de la opción elegida: '))
print()

if op == 1:
    p = cargar_arreglo_ordenado()

elif op == 2:
    mostrar_arreglo(p)

elif op == 3:
    buscar_titulo(p)

elif op == 4:
    buscar_codigo(p)

elif op == 5:
    conteo(p)

elif op == 6:
    crear_archivo(p)

elif op == 7:
    mostrar_archivo()

elif op == 8:
    pass

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

```

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) Inserción ordenada en un arreglo: localización del punto de inserción por búsqueda binaria.

En el *problema 55* de esta misma Ficha, se introdujo una función *add_in_order(p, paciente)* (ver 487 y siguientes) que tomaba como parámetro un arreglo de registros del tipo *Paciente*, supuestamente ordenado de acuerdo al campo *hist_clinica* de los registros contenidos (el número de historia clínica), y procedía a agregar en el arreglo el registro *paciente* que también entraba como parámetro, pero de forma tal que el arreglo se mantenga ordenado

de menor a mayor según el número de historia clínica (en el *problema 52* de esta misma Ficha, otra vez apareció la función *add_in_order()* pero esta vez para agregar el registro con los datos de un *artículo* a publicar... la idea es la misma). La función original era la siguiente:

```
def add_in_order(p, paciente):
    n = len(p)
    pos = n
    for i in range(n):
        if paciente.hist_clinica < p[i].hist_clinica:
            pos = i
            break
    p[pos:pos] = [paciente]
```

Como dijimos oportunamente, la función aplica el algoritmo de *búsqueda secuencial* para encontrar el *punto de inserción* (es decir, el índice del casillero donde debería agregarse el nuevo registro) para que el arreglo continúe ordenado. Pero como también dijimos, si el arreglo está ya ordenado podemos aplicar *búsqueda binaria* en lugar de búsqueda secuencial para encontrar ese punto de inserción, y hacer el proceso mucho veloz: en lugar de un tiempo de ejecución $O(n)$, pasaríamos a un tiempo $O(\log(n))$.

El algoritmo de búsqueda binaria que conocemos determina si un arreglo ordenado contiene o no a un determinado valor x , retornando el índice del casillero que lo contiene en caso de existir, o -1 en caso de no existir. Pero ahora necesitamos que el algoritmo *busque y retorne el índice del primer elemento del arreglo que sea mayor o igual al que se quiere insertar, o bien retorne el tamaño n del arreglo si el valor a insertar es mayor que todos los que el arreglo contiene*, y para ello necesitamos algunos ajustes.

Por lo pronto, si el número de historia clínica del paciente a agregar ya existe en la posición *pos* en el arreglo, podemos detener la búsqueda en forma normal, e insertar el nuevo paciente en la misma posición *pos*, moviendo a la derecha a todos los registros desde allí en adelante (incluyendo al que tenía el mismo número de historia clínica).

Pero si no existe un paciente con el mismo número de historia clínica, el ciclo de búsqueda continuará hasta que los índice auxiliares *izq* y *der* se crucen (es decir, hasta que *izq* se haga mayor que *der*). Lo interesante es que en el momento en que se crucen, el valor de *izq* será el índice de la casilla con el *primer valor mayor* que el que se quiere agregar (y de hecho, *der* contendrá el índice del casillero con el *último valor menor* al que se quiere insertar). Por lo tanto, en este caso, sólo debemos insertar el nuevo registro en la posición indicada por *izq*, y correr hacia la derecha a todos los registros desde esa posición en adelante.

Con estos cambios, la función *add_in_order()* podría quedar finalmente así:

```
def add_in_order(p, paciente):
    n = len(p)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2
        if p[c].hist_clinica == paciente.hist_clinica:
            pos = c
            break

        if paciente.hist_clinica < p[c].hist_clinica:
            der = c - 1
        else:
            izq = c + 1

    if izq > der:
```

```
pos = izq  
p[pos:pos] = [paciente]
```

La variable *pos* comienza valiendo *n*, el tamaño del arreglo, y ese es el valor que quedará asignado en *pos* **si no existe** un paciente con el mismo número de historia clínica que el que se quiere agregar. Si existiese un paciente con el mismo número, el valor de *pos* cambia para tomar el valor del índice *c* de la casilla que lo contiene, y el ciclo de búsqueda corta con *break*.

Al terminar el ciclo, se chequea si el valor de *izq* terminó siendo mayor que *der*. Si ese fue el caso, significa que el ciclo continuó su marcha sin activar el *break* citado en el párrafo anterior, y esto a su vez implica que ese registro no existía en el arreglo. Por lo tanto, sólo en ese caso, el valor de *pos* cambia para tomar el valor de *izq*, y finalmente se inserta el nuevo registro en la posición indicada por *pos*. Asegúrese el estudiante de hacer un análisis detallado del mecanismo aplicado en esta función, y comprender los detalles.

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el Ing. Valerio Frittelli para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

- [1] V. Frittelli, *Algoritmos y Estructuras de Datos*, Córdoba: Universitas, 2001.
- [2] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: <https://docs.python.org/3/>.
- [3] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: <http://www.diveintopython.net/toc/index.html..>