Ficha 14

Arreglos – Casos de Estudio I

1.] Arreglos correspondientes (o "paralelos").

En muchas ocasiones será necesario almacenar información en varios arreglos unidimensionales a la vez, pero de tal forma que haya correspondencia entre los valores almacenados en la casillas con el mismo índice (llamadas *casillas homólogas*). Por ejemplo, podría ocurrir que se pida guardar en un arreglo los nombres de ciertas personas, y en otro arreglo el importe del sueldo que perciben:

Figura 1: Dos arreglos *correspondientes* o *paralelos*.

nombres	Juan	Ana	Alejandro	María	Pedro
sueldos	1100	2100	1300	750	800
	0	**1**	**2**	**3**	**4**

Como se está pidiendo almacenar los datos en *dos* arreglos, sólo se debe cuidar que el nombre y el importe del sueldo de una misma persona aparezcan en ambos arreglos en *casillas homólogas*. Así, en nuestro ejemplo, la persona representada por la casilla 0 del arreglo *nombres* (o sea, *nombres[0]*) se llama "Juan", y el sueldo que percibe aparece en el arreglo *sueldos* también en la casilla con índice 0 (o sea, *sueldos[0]*) y es el valor 1100. Cuando dos o más arreglos se usan de esta forma, se los suele designar como *arreglos correspondientes* o *paralelos* [1].

El manejo de ambos arreglos es simple: sólo debe recordar el programador que hay que usar el mismo índice en *ambos arreglos* para entrar a la información de una misma persona. El siguiente esquema, muestra por pantalla el nombre y el sueldo de la persona en la casilla 2:

```
print('Nombre:', nombres[2])     # Alejandro
print('Sueldo:', sueldos[2])     # 1300
```

El siguiente problema servirá para aclarar la forma general de manejar *arreglos paralelos*:

Problema 37.) *Desarrollar un programa que permita cargar tres arreglos con n nombres de personas, sus edades y los sueldos que ganan. Luego de realizar la carga de todos los datos, mostrar los nombres de las personas mayores de 18 años que ganan menos de 10000 pesos, pero de forma que el listado salga ordenado en forma alfabética.*

Discusión y solución: El programa usará una función *read()* cuyo objetivo es cargar por teclado los tres arreglos. Cuando se trabaja con arreglos paralelos, en los que la componente en la casilla *i* de cada arreglo contiene información relacionada al mismo objeto o entidad del problema, es conveniente que la carga por teclado se haga de forma de barrer los tres arreglos al mismo tiempo: se carga primero la componente 0 de *todos* los arreglos, luego la componente 1 de *todos* ellos, y así sucesivamente. De esta forma, quien hace la carga de

datos ve facilitada su tarea pues carga los datos de una persona de una sola vez, sin tener que volver atrás luego de cargar los nombres o las edades para cargar los sueldos (lo que sería realmente incómodo). La función *read()* de nuestro programa sería la siguiente:

```
def read(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n):
        nombres[i] = input('Nombre[' + str(i) + ']: ')
        edades[i] = int(input('Edad: '))
        sueldos[i] = int(input('Sueldo: '))
        print()
```

El proceso de ordenamiento será realizado por la función *sort()* mediante nuestro ya conocido algoritmo de ordenamiento por *Selección Directa*. El arreglo de nombres debe ser ordenado en forma alfabética o *lexicográfica*: como sabemos, la idea intuitiva es que las palabras que en un diccionario aparecerían primero se consideran "menores" (alfabéticamente hablando) que las palabras que en el diccionario aparecerían después. Así, el nombre "*Ana*" es lexicográficamente menor que el nombre "*Luis*", pues "*Ana*" aparecería primero en un diccionario que "*Luis*". En Python, La comparación de cadenas es simple y directa: como sabemos, dos cadenas pueden ser comparadas directamente mediante los operadores relacionales típicos, y Python hará el trabajo correcto [2] [3].

Note, no obstante, que la función *sort()* no sólo debe ordenar el arreglo de nombres, sino que debe cuidar que al ordenar ese arreglo se mantenga la correspondencia entre los elementos de ese arreglo y los elementos de los otros dos que contienen las edades y los sueldos. El principio es simple: si se realiza un cambio en el arreglo de nombres, el mismo cambio debe reflejarse en los otros dos, por lo cual se realizan tres operaciones de intercambio (una para el arreglo de nombres, otra para el arreglo de edades y otra para el de sueldos). La función que hace el ordenamiento sería la que sigue:

```
def sort(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n-1):
        for j in range(i+1, n):
            if nombres[i] > nombres[j]:
                nombres[i], nombres[j] = nombres[j], nombres[i]
                edades[i], edades[j] = edades[j], edades[i]
                sueldos[i], sueldos[j] = sueldos[j], sueldos[i]
```

Finalmente, la función *display()* usa un ciclo *for* para analizar cada componente del arreglo de edades y del arreglo de sueldos. Si se encuentra que la edad en la posición *i* es mayor a 18 y al mismo tiempo el sueldo es menor a 10000, se muestran los tres datos que corresponden a esa persona, que se encuentra en cada arreglo en la misma posición *i*. La función podría ser la siguiente:

```
def display(nombres, edades, sueldos):
    n = len(nombres)
    print('ayores de 18 que ganan menos de 10000 pesos:')
    for i in range(n):
        if edades[i] > 18 and sueldos[i] < 10000:
            print('Nombre:', nombres[i], 'Edad:', edades[i], 'Sueldo:', sueldos[i])
```

El programa completo que se muestra a continuación. está incluido en el modelo *tes01.py* que forma parte del proyecto *[F14] Arreglos* que acompaña a esta Ficha.

```
__author__ = 'Cátedra de AED'


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 read(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n):
        nombres[i] = input('Nombre[' + str(i) + ']: ')
        edades[i] = int(input('Edad: '))
        sueldos[i] = int(input('Sueldo: '))
        print()


def sort(nombres, edades, sueldos):
    n = len(nombres)
    for i in range(n-1):
        for j in range(i+1, n):
            if nombres[i] > nombres[j]:
                nombres[i], nombres[j] = nombres[j], nombres[i]
                edades[i], edades[j] = edades[j], edades[i]
                sueldos[i], sueldos[j] = sueldos[j], sueldos[i]


def display(nombres, edades, sueldos):
    n = len(nombres)
    print('Personas mayores de 18 que ganan menos de 10000 pesos:')
    for i in range(n):
        if edades[i] > 18 and sueldos[i] < 10000:
            print('Nombre:', nombres[i], 'Edad:', edades[i], 'Sueldo:', sueldos[i])


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

    # crear los tres arreglos de n elementos...
    nombres = n * [' ']
    edades = n * [0]
    sueldos = n * [0]

    # cargar los tres arreglos por teclado...
    print('\nCargue los datos de las personas:')
    read(nombres, edades, sueldos)

    # ordenar alfabéticamente el sistema...
    sort(nombres, edades, sueldos)

    # avisar por pantalla el resultado de la búsqueda...
    display(nombres, edades, sueldos)


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

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 aun cuando los mismos no sean específicamente tratados en clase.

a.) Conteo y/o acumulación por acceso directo (*vectores de conteo y/o acumulación*).

Los arreglos son estructuras de datos especialmente útiles, puesto que brindan la posibilidad del *acceso directo* a cada componente. Es decir: si se necesita acceder directamente a un elemento, sólo se requiere conocer el índice del mismo y no es necesario pasar en forma secuencial por todos los elementos anteriores. Existen situaciones (el caso de la *búsqueda binaria* es una de ellas) en las cuales esta propiedad permite resolver problemas en forma notablemente sencilla y rápida. Por ejemplo, considérese el siguiente problema:

Problema 38.) *Cargar por teclado un conjunto de valores tales que todos ellos estén comprendidos entre 0 y 99 (incluidos ambos). Se indica el fin de datos con el número -1. Determinar cuántas veces apareció cada número.*

Discusión y solución: A primera vista, el problema parece de solución obvia: se usa un ciclo para cargar de a un valor (*num*) por vez, de forma que el ciclo sólo se detenga cuando sea *num == -1*. Como se pide determinar cuántas veces apareció cada valor posible de *num*, se usa un contador distinto por cada valor diferente que *num* pueda asumir. Así, para contar cuántas veces apareció el *1*, se puede usar el contador *c1*, para el *2* se podrá usar *c2*, y así sucesivamente. En cada repetición del ciclo se usa un *if* encadenado para determinar qué número se cargó en esa vuelta, y en función del número se elige el contador correspondiente. Al finalizar el ciclo, se muestran todos los contadores, y asunto terminado...

Sin embargo, esa solución dista mucho de ser buena y sobre todo en casos como éste, en que la variable *num* puede asumir valores en un rango muy amplio (0 a 99). El programador debería declarar y poner en cero *cien contadores*, y luego, dentro del ciclo, usar *¡cien condiciones anidadas!* para determinar qué contador debería usar de acuerdo al número ingresado. Además del gran esfuerzo de codificación que el programador deberá realizar, se tendrá el problema de la enorme redundancia de instrucciones semejantes, y por si esto fuera poco, el programa resultante será muy poco flexible en la práctica: si luego de plantearlo nos cambian las condiciones supuestas (por ejemplo: si los valores que puede asumir *num* pasaran a ser entre 0 y 150) entonces el programa original ya no serviría de nada...

La solución correcta es usar un *arreglo unidimensional c de cien componentes*, de forma que cada componente se use como uno de los contadores que se están necesitando [1]. Se pone inicialmente en cero cada componente del vector (dado que esos componentes serán contadores). Luego comienza el ciclo para cargar los valores *num*. La idea central, es que si *num* vale *0*, entonces debe usarse *c[0]* para contarlo. Si *num* vale *1*, se usa *c[1]*, y así sucesivamente. En general, para cada valor de la variable *num* tal que *0 <= num <= 99*, debe usarse *c[num]* para contarlo... En otras palabras: dentro del ciclo *no es necesario* usar

condiciones anidadas para determinar qué casillero del arreglo usar para contar cada número: se accede *directamente* al casillero cuyo índice es *num* y se incrementa el mismo en uno... Al cortar el ciclo, se muestra el contenido del arreglo, y ahora sí el problema queda resuelto de manera convincente.

Cuando un arreglo unidimensional se usa de esta forma, se lo suele designar como *vector de conteos* o simplemente como *vector de contadores*[1]. Si el arreglo se usara para *acumular* valores (en vez de *contarlos* como se supuso en el ejemplo), se hablaría entonces de un *vector de acumulación*.

Se muestra a continuación el programa que aplica lo dicho y resuelve el problema (ver modelo *test02.py* en el proyecto *[F14] Arreglos*): en la función *test()* se define el vector de conteos *c* de *n = 100* elementos valiendo cada uno de ellos inicialmente 0, que se usará para contar cuántas veces aparecen los números en el rango *[0, n-1]*. Luego se cargan los números mediante un ciclo de doble lectura, *y se aplica la técnica vista para proceder a contarlos*. Al finalizar, se recorre el vector de conteos y se visualizan los resultados, pero de forma que sólo se muestran los conteos de los números que hayan entrado al menos una vez (lo cual ahorra algo de espacio en la consola de salida…):

```
__author__ = 'Cátedra de AED'


def test():
    # crear el vector de conteo...
    n = 100
    c = n * [0]

    # cargar y contar los números...
    num = int(input('Ingrese un valor entre 0 y 99 (con -1 corta): '))
    while num != -1:
        if 0 <= num < n:
            c[num] += 1
        else:
            print('Error... el número debía ser >= 0 y <', n)
        num = int(input('Ingrese otro valor entre 0 y 99 (con -1 corta): '))

    # mostrar los resultados...
    print('Resultados:')
    for i in range(n):
        if c[i] != 0:
            print('Número', i, '- Frecuencia de aparición', c[i])


if __name__ == '__main__':
    test()
```

[1] Una situación de conteo muy conocida se da en las famosas *cuentas regresivas* que tienen lugar en distintos tipos de eventos (lanzamientos de naves espaciales, llegada del Año Nuevo, etc.) A su vez, es muy común en el cine encontrar escenas basadas en cuentas regresivas: Una de esas escenas se dio en la película *Independence Day* (o *Día de la Independencia*), de 1996, dirigida por *Roland Emmerich* y protagonizada por *Will Smith* y *Jeff Goldblum*. Una avanzada (y hostil) civilización extraterrestre llega a la Tierra y posiciona numerosas naves en el espacio aéreo de muchas ciudades importantes de todo el mundo, como si estuviesen vigilándolas. Estas naves emiten una extraña señal mientras se están posicionando, y pronto alguien descubre que la señal es en realidad una cuenta regresiva… que en cuanto llegue a cero provocará el ataque simultáneo de todas esas naves a todas las ciudades vigiladas, iniciando una invasión.

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, y en especial las *ing. Analía Guzmán*, *Cynthia Corso* y *Karina Ligorria*, 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.