Ficha 15

Arreglos – Casos de Estudio II

1.] Arreglos: análisis y solución de problemas básicos.

El contenido completo de esta Ficha está orientado al análisis, discusión y solución de diversos problemas que requieren el uso y aplicación de arreglos. La Ficha aporta entonces algunas técnicas elementales en ese marco y sirve al mismo tiempo como un elemento de repaso de los temas generales que surgieron hasta ahora. El primero de los problemas a discutir es el siguiente:

Problema 39.) *Desarrollar un programa que permita cargar un arreglo con las alturas de n personas. Determinar la altura media del grupo, e informar cuántas de esas personas tienen altura mayor a la media, y cuántas tienen altura menor o igual a la media.*

Discusión y solución: El programa completo se muestra directamente a continuación:

```
__author__ = 'Cátedra de AED'


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

    return n


def read(alt):
    n = len(alt)
    print('Cargue ahora las alturas (en centimetros) del grupo...')
    for i in range(n):
        alt[i] = int(input('Altura[' + str(i) + ']: '))


def average(alt):
    n, s = len(alt), 0
    for i in range(n):
        s += alt[i]

    return s/n


def count(alt, med):
    n = len(alt)
    c1 = c2 = 0
    for i in range(n):
        if alt[i] > med:
            c1 += 1
```

```
        else:
            c2 += 1

    return c1, c2


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

    # crear el arreglo de n elementos...
    alturas = n * [0]

    # cargar arreglo por teclado...
    read(alturas)

    # calcular el promedio y efecturar el conteo...
    media = average(alturas)
    mayores, menores = count(alturas, media)

    # mostras resultados...
    print('La altura media del grupo es:', media)
    print('Alturas mayores a la media:', mayores)
    print('Alturas menores a la media:', menores)


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

La creación y carga del arreglo con todas las alturas a procesar no ofrece dificultades. La función *test()* (que sirve como punto de arranque del programa) carga por teclado el valor *n* (validando que se mayor a cero), crea el arreglo de componentes (inicialmente con ceros en cada casillero), e invoca a la función *read()* para cargar por teclado efectivamente los datos. Por razones de brevedad, la carga de estos datos se hizo sin validación (dejamos para los estudiantes la tarea de hacer esas validaciones).

El paso siguiente es el cálculo de la *altura media* (o *altura promedio*). La función *test()* invoca para eso a la función *average()*, la cual toma como parámetro al arreglo con las alturas, acumula esas alturas, y retorna el cociente entre el acumulador y el valor de n (lo cual es efectivamente la altura media pedida).

La función *count()* desarrolla el proceso final: toma como parámetro el arreglo de alturas y la altura media, recorre nuevamente el arreglo y simplemente usa dos contadores para llevar la cuenta de los valores que sean mayores que la media y los que sean menores o iguales a ella. Esos dos contadores son retornados a la función *test()*, que finalmente muestra en pantalla los resultados así obtenidos.

Está claro que el problema que se acaba de presentar es de naturaleza sencilla y no ofrece dificultades especiales, pero resulta de interés para remarcar un hecho práctico: el conteo de los valores que son mayores (o menores) que el promedio de los datos contenidos en un arreglo, requiere ***obligatoriamente dos recorridos*** de ese arreglo: uno para hacer la acumulación necesaria para el cálculo del promedio (función *average()* en nuestro caso), y otro para comparar los valores contra el promedio ya calculado y efectuar el conteo (función *count()* en nuestro caso) [1].

Presentamos ahora un nuevo problema para analizar y discutir:

Problema 40.) *Desarrollar un programa que permita cargar por teclado dos arreglos de números enteros de n y m componentes respectivamente, y genere y muestre un tercer arreglo que contenga los valores que aparecen repetidos en los dos arreglos originales. Es decir, el nuevo arreglo debe contener todos los números que estando en uno de los dos arreglos originales, están también el otro. Si un número está dos o más veces en el mismo arreglo (y también figura en el segundo arreglo), sólo debe aparecer una vez en el tercer arreglo.*

Discusión y solución: Diseñaremos una función *generate()* que será la encargada de crear el tercer arreglo *v3* a partir de los dos originales *v1* y *v2* que reciba como parámetro. La consigna general es que todo elemento de *v1* que esté también en *v2*, se agregue a *v3*. En principio, la estrategia básica es simple: se recorre el arreglo v1 con un ciclo, y se compara el elemento actual *v1[i]* contra cada uno de los elementos de *v2* (usando otro ciclo). Cada vez se detecte que *v1[i]* está en *v2*, se agrega *v1[i]* a *v3* con el método *append()*. El planteo mínimo y esencial de esa función podría ser el siguiente:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):
        for j in range(len(v2)):
            if v1[i] == v2[j]:
                # si v1[i] está en v2, agregarlo en v3...
                v3.append(v1[i])

    # retornar el nuevo arreglo...
    return v3
```

Así planteada, la función *generate()* efectivamente encontrará todos los números que estén a la vez en *v1* y *v2*, y los copiará en *v3*. Si los arreglos fuesen (por ejemplo) v1 = [5, 3, 2, 4] y v2 = [1, 5, 7, 3, 8] entonces la función *generate()* crearía y retornaría correctamente el arreglo v3 = [5, 3].

Sin embargo, hay al menos un problema con el planteo propuesto para *generate()*: podría ocurrir que v2 contuviese más de una vez un mismo número que también esté en v1. Por ejemplo, si el arreglo v2 del ejemplo anterior fuese v2 = [1, 5, 7, 3, 5] entonces la función *generate()* que propusimos generaría un arreglo de la forma v3 = [5, 5, 3] con el 5 repetido… Esto se debe a que dentro del ciclo que recorre a v2 (el ciclo regulado por j) se comparan ***todos*** los elementos de v2 contra el elemento actual v1, y cada vez que se la condición es *True* el elemento v1[i] se agrega en v3. La forma obvia de arreglar este problema es cortar con ***break*** el ciclo más interno [2] (el que recorre a v2, regulado con la variable j) en el primer momento en que la condición sea *True*. El cambio es simple de implementar:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):
        for j in range(len(v2)):
            if v1[i] == v2[j]:
                # si v1[i] está en v2, agregarlo en v3...
                v3.append(v1[i])

                # .. y cortar el ciclo para evitar repeticiones...
                break
```

```
        # retornar el nuevo arreglo...
        return v3
```

Con este cambio, si el valor que se agrega en v3 está repetido dos o más veces en v2, será añadido sólo una vez en v3, con lo que parecería que el problema ha quedado resuelto…

Pero queda un caso más: el valor que se toma de v1 podría estar repetido a su vez en el propio arreglo v1, y eso también provocaría que se copie más de una vez en v3. Por ejemplo, si fuese v1 = [5, 3, 5, 4] y v2 = [1, 5, 7, 3, 8] entonces se crearía el arreglo v3 = [5, 3, 5] con el 5 repetido… La solución a este caso consiste en que antes de buscar el valor v1[i] en v2, se lo busque primero en v3 para comprobar si ya había sido agregado anteriormente, y en ese caso no buscarlo en v2. La versión final de la función *generate()* sería la que sigue:

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):

        # comprobar si v1[i] ya está en v3...
        exists = False
        for k in range(len(v3)):
            if v3[k] == v1[i]:
                exists = True
                break

        # si v1[i] no está en v3, buscarlo en v2...
        if not exists:
            for j in range(len(v2)):
                if v1[i] == v2[j]:
                    # si está en v2  agregarlo y cortar el ciclo...
                    v3.append(v1[i])
                    break

    # retornar el nuevo arreglo...
    return v3
```

El programa completo se muestra a continuación (dejamos el resto de los detalles del programa para ser analizados por cada estudiante):

```
__author__ = 'Cátedra de AED'


def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Cantidad (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')

    return t


def read(v):
    n = len(v)
    print('Cargue ahora los datos de este arreglo...')
    for i in range(n):
        v[i] = int(input('Valor[' + str(i) + ']: '))
```

```
def generate(v1, v2):
    v3 = []
    for i in range(len(v1)):

        # comprobar si v1[i] ya está en v3...
        exists = False
        for k in range(len(v3)):
            if v3[k] == v1[i]:
                exists = True
                break

        # si v1[i] no está en v3, buscarlo en v2...
        if not exists:
            for j in range(len(v2)):
                if v1[i] == v2[j]:
                    # si está en v2  agregarlo en v3 y cortar el ciclo...
                    v3.append(v1[i])
                    break

    # retornar el nuevo arreglo...
    return v3


def test():
    # primer arreglo...
    print('Primer arreglo...')
    n = validate(0)
    v1 = n * [0]
    read(v1)
    print()

    # segundo arreglo...
    print('Segundo arreglo...')
    m = validate(0)
    v2 = m * [0]
    read(v2)
    print()

    # generar y mostrar el tercer arreglo...
    v3 = generate(v1, v2)
    print('Los valores presentes en ambos arreglos son:')
    print(v3)


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

El último problema que veremos es el siguiente:

Problema 41.) *Cargar por teclado un arreglo de longitud n, y determinar si los elementos del mismo están en secuencia de k en k, siendo k un número que se ingresa también por teclado. Por ejemplo:*

v = [2, 5, 8, 11, 14] está en secuencia de 3 en 3 (k = 3)
v = [4, 6, 8, 10] está en secuencia de 2 en 2 (k = 2)

Discusión y solución: El análisis del contenido del arreglo para comprobar si sus elementos están en secuencia de *k* en *k* será realizado a través de la función *sequence(v, k)*, la cual toma

como parámetro al propio arreglo *v* ya creado y cargado, y al valor k que indica la amplitud de la secuencia. La función *sequence(v, k)* retorna *True* si efectivamente los datos contenidos en *v* están en secuencia de *k* en *k*, o retorna *False* en caso contrario.

La idea es directa: se recorre el arreglo con un ciclo comenzando desde la casilla 1, y se comprueba en cada vuelta si el valor de la casilla *v[i]* coincide con el de la casilla inmediatamente anterior *v[i-1]* pero sumado a *k*. Si la secuencia se cumple en todo el arreglo, entonces todas las comprobaciones de la forma *v[i] != v[i-1] + k* deben ser falsas. Por lo tanto, si alguna de esas comprobaciones fuese *True*, la función puede terminar en ese mismo momento retornando *False* sin necesidad de analizar el resto del arreglo (la secuencia se rompe con un solo par de casillas en las que no se cumpla). La función *sequence()* puede entonces quedar así:

```
def sequence(v, k):
    n = len(v)
    for i in range(1, n):
        if v[i] != v[i-1] + k:
            return False

    return True
```

El resto del programa (que se muestra completo a continuación) no presenta mayores dificultades, por lo que dejamos su análisis para el estudiante:

```
__author__ = 'Cátedra de AED'


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

    return n


def read(v):
    n = len(v)
    print('Cargue ahora los datos...')
    for i in range(n):
        v[i] = int(input('v[' + str(i) + ']: '))


def sequence(v, k):
    n = len(v)
    for i in range(1, n):
        if v[i] != v[i-1] + k:
            return False

    return True


def test():
    # cargar tamaño del arreglo...
    print('Ingrese tamaño del arreglo...')
    n = validate(0)
    print()
```

```
    # crear el arreglo de n elementos...
    v = n * [0]

    # cargar arreglo por teclado...
    read(v)
    print()

    # cargar el valor de k...
    print('Ingrese el valor de k...')
    k = validate(0)

    # comprobar la secuencia...
    ok = sequence(v, k)

    # mostras resultados...
    if ok:
        print('El arreglo está en secuencia de', k, 'en', k)
    else:
        print('El arreglo no está en secuencia de', k, 'en', k)


# 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.) Encriptación simple de mensajes.

La humanidad desarrolló la escritura hace aproximadamente 6000 años... y es posible que también desde esa misma época haya surgido la necesidad de proteger un escrito importante para que sólo pueda ser entendido y leído por las personas adecuadas. Muy probablemente haya sido el campo militar el que primero requirió desarrollar técnicas para *encriptar* (o *cifrar*) un mensaje: es común en este campo que un comandante deba enviar órdenes a algún subalterno o a algún aliado, pero de tal forma que esas órdenes sólo puedan ser interpretadas por el subjefe o aliado correcto. Si el mensaje cayera en manos extrañas (por ejemplo, en manos del enemigo) el daño recibido podría ser tan grande como la pérdida de la guerra.

A medida que la civilización progresó, aumentaron también las necesidades de proteger un mensaje contra ojos extraños u hostiles. Sobran los ejemplos. Hoy en día, es muy común (e indispensable...) proteger con algún tipo de encriptado las claves que se usan para acceder a información bancaria o a cualquier tipo de información que un usuario considere tan importante como para impedir a toda costa su acceso por parte de personas no autorizadas.

Como se ve, la *criptología* (ciencia que estudia la forma de ocultar un mensaje o una transmisión) existe desde mucho tiempo antes que la ciencia de la computación, pero esta última ha provisto nuevos elementos (en velocidad y potencia de cálculo) que han abierto el juego e incluso han cambiado sus reglas. En esta Ficha de estudio veremos una aproximación a un par de métodos ***muy simples y muy elementales*** para encriptar un mensaje, y sus implementaciones básicas, pero quede claro que esas implementaciones son sólo ejemplos o modelos de análisis, y no son implementaciones finales ni listas para ser usadas en situaciones reales. La aplicación de cualquier técnica de encriptación a una situación real, con verdadero peligro de transgresión por parte de un "atacante", debe ser estudiada con mucho cuidado y profesionalismo, sin subestimar ningún detalle ni hacer suposiciones que luego puedan significar caer en una trampa...

En general, llamaremos *mensaje en claro* (o *mensaje abierto*) al texto o información original que se quiere proteger; y designaremos como *mensaje encriptado* (o *mensaje cifrado*) al mensaje o cadena de símbolos que se obtiene una vez aplicado el mecanismo de ocultación que se haya elegido. Ese mecanismo de ocultación, a su vez, será designado como el algoritmo o *método de encriptación* (o *método de cifrado*). Un *criptógrafo* es la persona que intenta encriptar un mensaje, y un *criptoanalista* es la persona que intenta descifrarlo o desencriptarlo (sin conocer el algoritmo ni las claves originales para hacerlo) [3]. Dependiendo el cristal con que se mire, algunos considerarán como "buenos" a los criptógrafos y como "malos" a los criptoanalistas... y otros lo harán al revés. El hecho es que parece inevitable que haya "bandos" que querrán ocultar sus transmisiones, y otros "bandos" que querrán violar las medidas de seguridad del bando opuesto para ver esas transmisiones.

Uno de los métodos más simples y más antiguos para encriptar un mensaje se atribuye a Julio César, el general y luego emperador romano. Se designa por ese motivo como el *método de César*, y la idea es simple y directa: Si una letra del mensaje ocupa el *n-ésimo* lugar del alfabeto, reemplácela por la que ocupa el *(n+k)-ésimo* lugar, donde *k* es un número entero prefijado y sólo conocido por el encriptador y sus aliados [3].

Por ejemplo si *k = 3* y el mensaje original es *'CASA'*, entonces el mensaje codificado sería *'FDVD'* (suponiendo que el alfabeto original consta sólo de las letras mayúscula y ningún otro símbolo): cada letra se reemplaza por la que esté a k = 3 lugares más adelante en el alfabeto, y así la letra 'C' se reemplaza por la 'F', la letra 'A' se reemplaza por la 'D' y la 'S' por la 'V'. Puede verse que esta técnica es muy fácil de transgredir: en nuestro ejemplo, el criptoanalista o atacante sólo debería saber cuál es el alfabeto original, y probar un máximo de 27 valores diferentes de *k* (suponiendo que el alfabeto original tiene las 27 letras simples (sin la LL ni la CH pero incluida la Ñ) del idioma español, hasta dar con un mensaje legible. En nuestros días, aun cuando el alfabeto original conste de muchos más símbolos (como podría ser la tabla ASCII completa con 256 símbolos) se pueden probar con rapidez todas las variantes de *k* usando una computadora.

Para mejorar la resistencia de esta técnica frente a un ataque, se puede ensayar una sencilla variante: en vez de usar el alfabeto original para obtener la letra o símbolo de reemplazo, se puede usar una *segunda tabla* (que llamaremos *tabla de transposición*) la cual puede estar formada por los mismos símbolos del alfabeto original (¡pero en otro orden!) o bien puede estar formada directamente por otros símbolos [3]. Si suponemos que el alfabeto original sólo consta de la letras mayúsculas, podemos usar (a modo de ejemplo) la *tabla de transposición* que se ve en el ejemplo siguiente:

Alfabeto:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	Ñ	O	P	Q	R	S	T	U	V	W	X	Y	Z
Transposición:	R	I	S	Q	P	A	N	O	W	X	U	M	D	H	Z	T	F	G	B	L	E	Y	K	C	J	V	Ñ

Así, si el mensaje fuera "*UN OJO EN EL CIELO*" tendría el siguiente cifrado (ignorando los espacios en blanco): "*YH TXT PH PM SWPMT*". Como se ve, la idea es simplemente reemplazar cada letra del texto en claro por su homóloga en la tabla de transposición. Obviamente, la tabla de transposición no debería ser un elemento conocido más que por los criptógrafos...

Con esta técnica, un atacante debería probar con una gran cantidad de tablas diferentes (si el alfabeto tiene *n* símbolos, serían alrededor de *n!* tablas diferentes) Sin embargo, todavía un criptoanalista podría violar el esquema haciendo un análisis minucioso: sabemos que en cada idioma hay letras que aparecen con más frecuencia que otras: en español la *'A'* o en inglés la *'E'* son las letras que más aparecen. Por lo tanto, un atacante podría buscar cuál es el símbolo que más se repite en el mensaje encriptado, y casi seguro tendría una letra en claro definida. También hay ciertas combinaciones que aparecen mucho (como *'la'* o *'en'* o *'el'* en español o *'the'* o *'and'* en inglés) y por lo tanto, analizando grupos de dos o tres símbolos en el mensaje encriptado, un criptoanalista también tendría razonables posibilidades de éxito (con cierta facilidad un atacante podría deducir del ejemplo anterior que la secuencia encriptada *'PH'* corresponde a la secuencia en claro *'EN'*, y de allí sería factible deducir el resto, sobre todo si el mensaje encriptado es largo).

En esta Ficha de estudios mostraremos solamente una forma de implementar la *técnica de César* y la *técnica de Tabla de Transposición*, pero hay que dejar dos hechos en claro: el primero, es que obviamente existen muchísimas estrategias adicionales de encriptación [3], como la *encriptación de Vigenère* (que permite mejorar la encriptación basada en tablas de transposición usando una secuencia numérica auxiliar designada como *clave de cifrado*) y que por mucho tiempo fue considerado indescifrable hasta que surgió el *método de Kasiski* especialmente diseñado para romper el *cifrado de Vigenère*), el *cifrado de Vernam* (una variante de la *encriptación de Vigenère* en la que la clave de cifrado tiene la misma longitud que el mensaje a encriptar) o el *cifrado de Mauborgne* (también conocido como *cifrado de un solo uso*, y que esencialmente es el mismo cifrado de Vernam pero haciendo que la clave de cifrado sea aleatoria y se use sólo una vez[1]).

El segundo hecho a dejar claro, es que las técnicas que someramente hemos descripto en los párrafos anteriores son estrategias antiguas y hoy en día obsoletas (se las designa como *métodos de encriptación clásicos*). Todos ellos cayeron en desuso con el advenimiento de las computadoras, cuya potencia de cálculo y velocidad de ejecución para probar miles de

[1] El tema de las consecuencias de cualquier evento aleatoriamente generado y su impacto en los hechos posteriores, es un tema recurrente en el cine y la literatura. En 2004, la película *The Butterfly Effect* (o *El Efecto Mariposa*) (dirigida por *Eric Bress* e interpretada por *Ashton Kutcher*) planteó en forma cruda la historia de un joven estudiante de psicología que de pronto descubre que puede regresar en el tiempo, y decide hacerlo para cambiar sus decisiones pasadas y alterar con eso algunos sucesos tristes o lamentables de su vida… sin tener en cuenta el significado profundo del eslogan básico de la película: "el aleteo de una simple mariposa puede provocar un tsunami al otro lado del mundo". Y las cosas cambiaron, pero no en la forma que él esperaba…

combinaciones de patrones en poco tiempo hicieron que los métodos clásicos se convirtiesen en simples juegos de ingenio.

En la civilización moderna, los sistemas de cifrado deben ser capaces de resistir un ataque informático masivo y por ello las técnicas usadas son dramáticamente diferentes a las clásicas. La mayor parte de las técnicas modernas (como el famoso *algoritmo RSA de clave pública* [3], diseñado por *Rivest*, *Shamir* y *Adleman* –de cuyas iniciales toma su nombre la técnica-) están basadas en algoritmos de base matemática que en alguna parte incluyen la necesidad de encontrar números primos muy pero muy grandes (del orden de 100 dígitos o más cada uno) o de factorizar números muy grandes para encontrar su factores primos (que también son muy grandes). Sabemos (por lo que hemos mostrado en la Ficha 09) que el proceso de factorización de un número en sus divisores primos puede demorar una increíble cantidad de tiempo (incluso para una computadora) si el número es muy grande o alguno de sus factores primos lo es. No se conocen hasta ahora algoritmos rápidos para hacer ese trabajo, y de alguna manera toda la seguridad de los procesos informáticos modernos (transacciones bancarias, acceso a sitios académicos, redes sociales, etc.) descansa en nuestra confianza de que esos algoritmos rápidos de factorización no existen… aún.

Formalmente, la implementación de los algoritmos de *cifrado de César* y de *Transposición* se muestra en la solución al siguiente problema:

Problema 42.) *Desarrollar un programa que permita cargar por teclado un mensaje compuesto sólo por mayúsculas y espacios en blanco, y lo encripte usando el cifrado de César y el cifrado de Transposición. Diseñe también los algoritmos de desencriptado.*

Discusión y solución: El programa completo que resuelve este problema se puede ver en el modelo *test04.py* del proyecto *F[15] Arreglos* que acompaña a esta Ficha.

Comencemos por el cifrado de César. La función *code_cesar(mens, ABC, k)* que mostramos a continuación, toma como parámetro una cadena con el mensaje a encriptar (parámetro *mens*), otra cadena con el alfabeto empleado (parámetro *ABC*) y el factor de desplazamiento a usar para el cifrado (parámetro *k*, un número entero que debería ser mayor a 1). A su vez, La función auxiliar *is_ok(cad, ABC)* controla si la cadena recibida en el parámetro *cad* está formada sólo por blancos y caracteres que estén en el alfabeto recibido en el parámetro *ABC* (retorna *True* si ese fuese el caso, o *False* si algún carácter en *mens* no es válido):

```
def is_ok(cad, ABC):
    if cad is None:
        return False

    if cad == '':
        return False

    for i in range(len(cad)):
        # si cad[i] no es blanco y no está en el alfabeto, retornar False...
        if cad[i] != ' ' and ABC.find(cad[i]) == -1:
            return False

    return True


def code_cesar(mens, ABC, k):
    if not is_ok(mens, ABC):
        return None
```

```
    na = len(ABC)
    b = []
    for i in range(len(mens)):
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        # si el caracter original no es blanco, reemplazarlo...
        else:
            # indice del caracter mens[i] en el alfabeto...
            im = ABC.find(mens[i])

            # indice del caracter de reemplazo en el alfabeto...
            ir = (im + k) % na

            # efecturar el reemplazo en el arreglo de salida b...
            b.append(ABC[ir])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr
```

La función *code_cesar(mens. ABC, k)* comienza controlando que la cadena recibida en el parámetro mens sea válida, invocando a la función *is_ok()*. Si algo estuviese mal, *code_cesar()* retorna *None* (a modo de aviso de procesamiento imposible).

Si la cadena que viene en *mens* es correcta, se crea un arreglo *b* inicialmente vacío para ir llenándolo luego con los caracteres encriptados que correspondan. Podría pensarse que *b* se inicie como una cadena vacía (en lugar de un arreglo vacío), pero como las cadenas son inmutables el proceso de ir llenando la cadena con los caracteres encriptados requeriría un mecanismo poco práctico y algo más lento que si se hace con un arreglo.

A continuación, la función *code_cesar()* emplea un ciclo *for* para recorrer la cadena *mens*, carácter a carácter. Si carácter actual *mens[i]* es un blanco, se copia el mismo blanco en el arreglo *b* (por razones de claridad, no encriptaremos los blancos, pero en un contexto real serían encriptados como cualquier otro caracter). Si el caracter actual *mens[i]* no es un blanco, se lo busca en el alfabeto (contenido en la cadena que viene en *ABC*) con el método *find()* provisto por la clase *str* (que representa a las cadenas de caracteres en Python). La instrucción [2]:

```
im = ABC.find(mens[i])
```

busca en la cadena *ABC* la cadena (en este caso, el caracter) representada por *mens[i]* y si la encuentra, retorna el índice donde *mens[i]* comienza dentro de *ABC*. Si no es encontrada, el método retorna el valor -1. Sabiendo entonces que *im* contiene el índice que le corresponde al caracter *mens[i]* dentro del alfabeto *ABC*, sólo quedaría sumar a *im* el valor *k* para obtener el índice del caracter de reemplazo. Pero hay un pequeño problema: la cadena *ABC* tiene cierta longitud *n* (que en nuestro caso es *n = 27*) y la suma *im + k* podría ser mayor a *n*. Por ejemplo, si el caracter *mens[i]* fuese la letra 'Y' (cuyo índice en *ABC* es *im = 25*) y el valor de *k* es 3, entonces la suma *im + k* sería 25 + 3 = 28 que no es un índice válido para *ABC*. Pero el caso es que la letra 'Y' necesita de todos modos un caracter de reemplazo, y la idea obvia en situaciones como esta es considerar al alfabeto como una *tabla circular*: si al buscar el reemplazo se supera el límite derecho de la tabla, continuar desde el principio y recorrer desde allí. En el caso de la 'Y', si tenemos que ir k = 3 casillas hacia "adelante", habría que avanzar hasta la letra 'B' (es decir, llegar hasta la 'Z' en la casilla 26, dar la vuelta y pasar por

la 'A' en la casilla 0, y parar en la 'B' en la 1). La forma de calcular el índice del caracter de reemplazo si la tabla ABC se toma en forma "circular", consiste en aplicar aritmética modular, tal como se vio en la Ficha 02: el índice del carácter de reemplazo puede calcularse con sencillez mediante la expresión:

```
na = len(ABC)
ir = (im + k) % na
```

Efectivamente, si el valor *im + k* es mayor que el tamaño *na* de la tabla entonces el resto de dividir por ese tamaño dará el valor del "exceso" a partir del final, que es justamente el índice del casillero de reemplazo comenzando desde el principio. En nuestro caso, si *im = 25* (el índice la letra 'Y'), y el valor de *k* es 3, con *na = 27*, entonces (25 + 3) % 27 = 28 % 27 = 1, que es el índice la letra 'B' en la cadena *ABC*… Es interesante recordar que si *im + k* es menor que *na*, entonces el resto es igual al propio valor *im + k*, y en ese caso el índice de reemplazo es exactamente el que se habría usado si la tabla no se tomase como circular. Por caso, el índice la letra 'C' es *im = 2*. Si se toma *k = 3* con *na = 27*, la expresión de cálculo resulta entonces (2 + 3) % 27 = 5 % 27 = 5, con lo que la letra 'C' sería reemplazada (correctamente) por la 'F'.

La función *code_cesar()* calcula uno a uno los índices de los caracteres de reemplazo aplicando las técnicas explicadas más arriba, y agrega en el arreglo *b* esos caracteres de reemplazo con el método *append()*. Al terminar el ciclo, sólo restaría entonces convertir el arreglo *b* a una cadena de caracteres, y retornar esa cadena. Pero esto nos lleva al último detalle técnico a resolver: si *b* es un arreglo (un objeto de tipo *list*) que contiene caracteres, podría pensarse que la función *str()* bastaría para convertir ese arreglo en una cadena… pero *str()* no hace exactamente lo que esperaríamos. Supongamos que el arreglo *b* quedó con los valores b = ['F', 'D', 'V', 'D']. Si entonces hacemos algo como:

```
encr = str(b)
```

para obtener la cadena final, lo que *str()* haría en realidad sería almacenar en *encr* la cadena "['F', 'D', 'V', 'D']" que claramente no es lo que necesitamos. La manera final y correcta de hacer la conversión consiste en invocar al método *join()* de la clase *str()* en la forma que se muestra a continuación:

```
encr = ''.join(b)
```

El método *join()* ***concatena*** los elementos que contiene la secuencia que toma como parámetro (el arreglo *b* en nuestro caso) creando una cadena de caracteres normal, y retorna la cadena así creada [2]. El método usa como separador de las cadenas que concatena a la cadena que se empleó para invocar al método (en nuestro caso, la cadena vacía: ''). Así, si el arreglo *b* fuese *b = ['F', 'D', 'V', 'D']*, entonces la instrucción *encr = ''.join(b)* asignaría en *encr* la cadena 'FDVD' esperada. Note que si la cadena usada para invocar al método fuese un espacio en blanco (o sea: ' ') en lugar de la cadena vacía, el resultado sería una cadena en la que todas las letras estarían separadas por un espacio en blanco. El siguiente script:

```
b = ['F', 'D', 'V', 'D']
encr = ' '.join(b)
print('Cadena:', encr)
```

produciría la siguiente salida:

```
Cadena: F D V D
```

La función *decode_cesar(encr, ABC, k)* que mostramos a continuación, es la encargada de desencriptar el mensaje almacenado en la cadena *encr* que toma como parámetro, usando el alfabeto *ABC* y el factor de desplazamiento *k* que también entran como parámetros:

```
def decode_cesar(encr, ABC, k):
    if not is_ok(encr, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(encr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # indice del caracter encr[i] en el alfabeto...
            ie = ABC.find(encr[i])

            # indice del caracter original en el alfabeto...
            io = (ie - k + na) % na
            b.append(ABC[io])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens
```

Esencialmente, esta función hace el proceso inverso del que hacía *code_cesar()*: ahora cada caracter que aparece en la cadena *encr* es un caracter encriptado y debe ser reemplazado por el caracter original, que en el alfabeto *ABC* estará *k* posiciones *hacia atrás*. Si *ie* es el índice que el carácter *encr[i]* tiene en el alfabeto *ABC*, entonces la expresión *ie – k* obtendría el índice del caracter original en ese mismo alfabeto. Pero como también aquí el resultado de *ie – k* podría ser menor que cero, se aplica la misma idea de tabla circular. Si el tamaño del alfabeto es *na*, se puede mostrar con relativa sencillez que la expresión

```
io = (ie - k + na) % na
```

asigna en la variable *io* el índice del caracter original que corresponde al caracter encriptado cuya posición en el alfabeto *ABC* es *ie*. Como el valor *ie – k* podría ser negativo, se suma el valor de *na* para ajustar el signo del resultado, sin alterar las propiedades de la aritmética modular. Por caso, si la letra encriptada fuese una 'B' (cuyo índice es *ie = 1* en el alfabeto) y el valor de *k* fuese *k = 5* con *na = 27*, entonces el índice de la letra original sería io = (1 – 5 + 27) % 27 = 23 % 27 = 23 que corresponde a la letra 'W'. El resto de los detalles de la función *decode_cesar()* no presenta mayor dificultad, y se dejan para el análisis del estudiante.

En cuanto a la *encriptación con Tabla de Transposición*, el mismo modelo *test04.py* del proyecto *[F15] Arreglos* que acompaña a esta Ficha incluye dos funciones similares a las que presentamos antes: una se llama *code_transposition(mens, ABC, TRANS)* y la otra se llama *decode_transposition(encr, ABC, TRANS)*. Se muestran ambas a continuación:

```
def code_transposition(mens, ABC, TRANS):
    if not is_ok(mens, ABC):
        return None

    b = []
    for i in range(len(mens)):
```

```
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        else:
            # si el caracter original no es blanco, reemplazarlo...
            im = ABC.find(mens[i])
            b.append(TRANS[im])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr


def decode_transposition(encr, ABC, TRANS):
    if not is_ok(encr, ABC):
        return None

    b = []
    for i in range(len(encr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # si el caracter encriptado no es blanco, reemplazarlo...
            ie = TRANS.find(encr[i])
            b.append(ABC[ie])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens
```

En ambas funciones, el parámetro *ABC* es el alfabeto original, y el parámetro *TRANS* es la tabla de transposición propuesta (o sea, el mismo alfabeto original, pero mezclado). En la función *code_transposition()* el parámetro *mens* es el mensaje a encriptar y la función retorna una cadena con ese mensaje encriptado. En la función *decode_transposition()* el parámetro *encr* es el mensaje ya encriptado, y la función retorna una cadena con el mensaje original desencriptado. Ambas usan un arreglo auxiliar *b* para ir armando la secuencia de salida (en la misma forma que ya vimos para el *cifrado de César*), y ese arreglo se convierte a cadena de caracteres al final (antes de ser retornado) mediante el método *join()* (como también se explicó para el *cifrado de César*).

Tanto en una como en la otra no hay demasiado misterio: la primera función debe tomar cada caracter *mens[i]*, buscar su posición o índice en el alfabeto *ABC*, y reemplazar *mens[i]* (en el arreglo *b* de salida) por el caracter que esté exactamente en la misma posición de la tabla TRANS, como se muestra en el siguiente segmento:

```
im = ABC.find(mens[i])
b.append(TRANS[im])
```

Y la segunda función hace lo mismo, con el caracter *encr[i]*, pero al revés:

```
ie = TRANS.find(encr[i])
b.append(ABC[ie])
```

Confiamos en que el estudiante podrá analizar y comprender por sí mismo los detalles que restan en ambas funciones. Para finalizar, mostramos ahora el programa completo (incluyendo la función *test()* de entrada):

```
__author__ = 'Cátedra de AED'
```

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

    return k


def is_ok(cad, ABC):
    if cad is None:
        return False

    if cad == '':
        return False

    for i in range(len(cad)):
        # si cad[i] no es blanco y no está en el alfabeto, retornar False...
        if cad[i] != ' ' and ABC.find(cad[i]) == -1:
            return False

    return True


def code_cesar(mens, ABC, k):
    if not is_ok(mens, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(mens)):
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        # si el caracter original no es blanco, reemplazarlo...
        else:
            # indice del caracter mens[i] en el alfabeto...
            im= ABC.find(mens[i])

            # indice del caracter de reemplazo en el alfabeto...
            ir = (im + k) % na

            # efecturar el reemplazo en el arreglo de salida b...
            b.append(ABC[ir])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr


def decode_cesar(encr, ABC, k):
    if not is_ok(encr, ABC):
        return None

    na = len(ABC)
    b = []
    for i in range(len(encr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # indice del caracter encr[i] en el alfabeto...
            ie = ABC.find(encr[i])
```

```
            # indice del caracter original en el alfabeto...
            io = (ie - k + na) % na
            b.append(ABC[io])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens


def code_transposition(mens, ABC, TRANS):
    if not is_ok(mens, ABC):
        return None

    b = []
    for i in range(len(mens)):
        # si el caracter original es blanco, dejar ese blanco...
        if mens[i] == ' ':
            b.append(' ')

        else:
            # si el caracter original no es blanco, reemplazarlo...
            im = ABC.find(mens[i])
            b.append(TRANS[im])

    # convertir el arreglo b a cadena de caracteres y retornar...
    encr = ''.join(b)
    return encr


def decode_transposition(encr, ABC, TRANS):
    if not is_ok(encr, ABC):
        return None

    b = []
    for i in range(len(encr)):
        # si el caracter encriptado es blanco, dejar ese blanco...
        if encr[i] == ' ':
            b.append(' ')

        # si el caracter encriptado no es blanco, reemplazarlo...
        else:
            # si el caracter encriptado no es blanco, reemplazarlo...
            ie = TRANS.find(encr[i])
            b.append(ABC[ie])

    # convertir el arreglo b a cadena de caracteres y retornar...
    mens = ''.join(b)
    return mens


def test():
    # el alfabeto general...
    ALFABETO = "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ"

    # Cifrado de César...
    # cargar el mensaje a encriptar (sin validación)...
    mensaje = input('Mensaje a encriptar (sólo mayúsculas y blancos...): ')
    print()

    print('Encriptación de César...')

    # cargar el valor del desplazamiento k...
    k = validate(1)

    # encriptar el mensaje con encriptación de César...
    encriptado = code_cesar(mensaje, ALFABETO, k)
```

```
    # desencriptar el mensaje encriptado (para comprobación)...
    desencriptado = decode_cesar(encriptado, ALFABETO, k)

    # si no hubo problemas, mostrar mensaje en claro y mensaje encriptado...
    if encriptado is not None:
        print('Mensaje original (tal como se cargó por teclado):', mensaje)
        print('Encriptado con clave k=', k, ': ', encriptado, sep='')
        print('Desencriptado (a partir del encriptado):', desencriptado)
    else:
        print('Error: posiblemente está mal el formato del mensaje original...')

    # Cifrado de Transposición...
    print()
    print('Encriptación de Transposición...')

    # una tabla de transposición para el alfabeto dado...
    TRANSPOSICION = "RISQPANOWXUMDHZTFGBLEYKCJVÑ"

    # encriptar el mensaje con encriptación de Transposición...
    encriptado = code_transposition(mensaje, ALFABETO, TRANSPOSICION)

    # desencriptar el mensaje encriptado (para comprobación)...
    desencriptado = decode_transposition(encriptado, ALFABETO, TRANSPOSICION)

    # si no hubo problemas, mostrar mensaje en claro y mensaje encriptado...
    if encriptado is not None:
        print('Mensaje original (tal como se cargó por teclado):', mensaje)
        print('Encriptado (tabla: ', TRANSPOSICION, '): ', encriptado, sep='')
        print('Desencriptado (a partir del encriptado):', desencriptado)
    else:
        print('Error: posiblemente está mal el formato del mensaje original...')


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

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