ITCR – ANÁLISIS Y DISEÑO DE ALGORITMOS
Tarea 1: práctica de elementos de computación

INSTRUCCIONES:

- Los productos entregables son los programas codificados en Python 3.
- Haga solo un programa fuente de nombre **tarea1_su_nombre.py**.
- Pruebe las funciones desde el modo comando.
- Ponga documentación interna de cada función: lo que hace la función, entradas, salidas
- En la etapa de diseño del algoritmo se sugiere hacer un esquema donde determine el comportamiento del algoritmo para luego proceder con su desarrollo.
- Recuerde usar buenas prácticas de programación como nombres significativos, reutilización de código (funciones), documentación interna, etc.
- Haga las validaciones de los datos de entrada que se indiquen explícitamente en cada ejercicio.
- Tenga en cuenta que además de la función principal puede necesitar otras funciones para obtener la solución.
- Enviar al tecDigital en la sección de EVALUACIONES / TAREAS.

PARA CUALQUIER TRABAJO QUE ENVÍE POR MEDIOS DIGITALES DEBE COMPROBAR QUE EL MISMO SE HAYA ENVIADO CORRECTAMENTE (revise que el trabajo enviado sea el que solicitaron, que se envíe al destino indicado -profesor, curso, plataforma-, que tenga una fecha y hora de envío).

- Cada ejercicio vale 10 puntos.
- Fecha de entrega: 31 de julio, 11 pm.

Ejercicio 1

En matemáticas un número abundante n cumple la siguiente propiedad: la suma de los divisores positivos de n incluyéndole es mayor al producto de 2 * n. La abundancia sería esta diferencia: suma de divisores - 2 * n.
Haga la función **numero_abundante** que reciba un número entero n (>= 1) e imprima (no retorna, solo imprime) los siguientes resultados:

Divisores de n
Suma de los divisores
Determinar: Es abundante o No es abundante
Abundancia: Cálculo de la abundancia o el mensaje “no hay” en caso de no haber abundancia

Ejemplos del funcionamiento:

Número recibido: 24
Divisores: [1, 2, 3, 4, 6, 8, 12, 24]
Suma de los divisores: 60
Es abundante (porque la suma de los divisores es mayor a 2 * número)
Abundancia: 12 (es la resta entre la suma de los divisores y 2 * número)

Número recibido: 38
Divisores: [1, 2, 19, 38]
Suma de los divisores: 60
No es abundante
Abundancia: no hay

Ejercicio 2

Considere el siguiente algoritmo para encriptar mensajes que solo contienen las 27 letras del alfabeto en minúscula (abcdefghijklmnñopqrstuvwxyz). Se escoge un número entero n entre 1 y 25 como clave y se suma a cada letra en el alfabeto n posiciones para obtener la letra encriptada pasada a mayúscula, por ejemplo si la clave escogida fuera 5 la “a” pasa a “F”, la “j” pasa a “Ñ”. Para las últimas letras del alfabeto se continúa desde el inicio, por ejemplo en este caso la “y” pasa a “D”. Haga la función **encripta** que reciba un string y retorne el mensaje encriptado según ese algoritmo y la clave escogida dentro de la función (use por ejemplo la función randint). En caso de recibir caracteres no permitidos retorne un string con esos caracteres y -1 en la clave.

Ejemplos del funcionamiento:

```
>>> encripta(“estudiando”)
(“JXYZINFRIT”, 5)

>>> encripta(“itcr”)
(“RDMB”, 10)

>>> encriptar(“Hoy es martes”)
(“H  “, -1)    # en la entrada solo se permiten las letras minúsculas
```

Ejercicio 3

En matemáticas un par de números *m* y *n* es llamado par amigable (o números amistosos), si la suma de todos los divisores de *m* (excluyendo a *m*) es igual al número *n*, y la suma de todos los divisores del número *n* (excluyendo a *n*) es igual a *m* (donde $m \neq n$).

Ejemplo: los números 220 y 284 son un par amigable. La explicación es la siguiente:

Los únicos números que dividen de forma exacta a 220 son 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 y 110, y la suma de ellos es: 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284

Los únicos números que dividen de forma exacta a 284 son 1, 2, 4, 71 y 142, y la suma de ellos es: 1 + 2 + 4 + 71 + 142 = 220

Haga la función **pares_amigables** que reciba una lista de números enteros (> 0 y diferentes entre sí) y retorne (recuerde que para retornar se usa **return**) una lista con los pares amigables que contiene la lista de entrada. Cada elemento de la lista retornada es una sublista con cada par amigable. Ejemplos del funcionamiento:

```
>>> pares_amigables([100, 1184, 220, 15, 18, 1, 1210, 284, 25])
[ [1184, 1210], [220, 284] ]

>>> pares_amigables([100, 284, 25])
[ ]
```

Ejercicio 4

La sucesión de Fibonacci es la siguiente sucesión infinita de números naturales:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

La sucesión empieza con los números 0 y 1, y a partir de estos el siguiente elemento es la suma de los dos números anteriores.

Haga la función **fibonacci** que reciba un entero n (>= 1) y retorne una tupla con los primeros n términos de la sucesión.
Valide que el dato de entrada sea entero >= 1.

Ejemplos del funcionamiento:

```
>>> fibonacci(5)
(0, 1, 1, 2, 3)

>>> fibonacci(8)
(0, 1, 1, 2, 3, 5, 8, 13)

>>> fibonacci(1)
(0,)

>>> fibonacci(-3)
Error: la entrada debe ser un entero >= 1

>>> fibonacci("hola")
Error: la entrada debe ser un entero >= 1
```

Ejercicio 5

Haga la función **serie** que reciba tres números enteros (son >= 1, el primer número es <= que el segundo número) y de como resultado una sola variable numérica que contenga todos los números enteros sucesivos que se encuentren en el rango indicado por los dos primeros números leídos. El primer número de la serie es el número más a la izquierda en el resultado, luego se le pega a la derecha el segundo número de la serie y así sucesivamente se van pegando a la derecha cada uno de los términos hasta llegar al último. El tercer número leído da el valor que debe aumentarse cada término para obtener el siguiente número de la serie (de 1 en 1, de 2 en 2, de 5 en 5, etc.). En este algoritmo no se permite el uso de: secuencias (strings, listas, tuplas), diccionarios, conjuntos.
Ejemplos del funcionamiento:

```
>>> serie(98, 101, 1)
9899100101

>>> serie(9, 14, 2)
91113

>>> serie(997, 1005, 1)
997998999100010011002100310041005
```

Ejercicio 6

Los números primos y semiprimos son altamente utilizados en los campos de la criptografía (codificación de datos) y de la teoría de números. Los números primos son los enteros >= 2 que tienen exactamente dos divisores: el 1 y él mismo, entre ellos: 2, 3, 5, etc. Los números semiprimos (o biprimos) son los enteros >= 2 formados por el producto de dos números primos, entre ellos: 4 (2 x 2), 39 (3 x 13), etc. Haga la función ***semiprimos*** que reciba un intervalo (2 enteros >= 2, el primero menor que el segundo) e imprima estos resultados:

+ cada número semiprimo en el intervalo y los primos que lo forman
+ cantidad total de semiprimos

En caso de no haber semiprimos imprima solamente el mensaje "NO HAY SEMIPRIMOS EN ESTE RANGO". Ejemplos del funcionamiento:

>>> semiprimos(14, 24)

Semiprimos	Primos que lo conforman
14	2 x 7
15	3 x 5
21	3 x 7
22	2 x 11

Cantidad total de semiprimos: 4

>>> semiprimos(17, 20)

Semiprimos	Primos que lo conforman

NO HAY SEMIPRIMOS EN ESTE RANGO

Ejercicio 7

Desarrolle la función **suma_diagonal** que reciba dos parámetros:

- Una matriz representada por filas de tamaño m x n (m > 0, n > 0, elementos numéricos)
- Un número de diagonal (entero).

La función debe **retornar** la suma de los elementos que están en el número de diagonal dado en el segundo parámetro. La diagonal principal es la diagonal 0, las diagonales arriba de la principal se numeran de 1 en adelante y las que están debajo de -1 en adelante tal como se muestra a continuación:

	0	1	2	3	4
	20	**50**	**60**	**70**	**80**
-1	**15**	**20**	**16**	**40**	**50**
-2	**30**	**56**	**60**	**25**	**30**
-3	**41**	**85**	**90**	**64**	**70**
-4	**68**	**43**	**12**	**24**	**16**

Validaciones: la matriz debe ser cuadrada de lo contrario retornar el mensaje “ERROR: NO ES CUADRADA”, la diagonal debe existir de lo contrario retornar el mensaje “ERROR: NO EXISTE LA DIAGONAL”.

Ejemplos del funcionamiento:

>>> suma_diagonal([[20, 50, 60, 70, 80], [15, 20, 16, 40, 50], [30, 56, 60, 25, 30], [41, 85, 90, 64, 70], [68, 43, 12, 24, 16]], 2)
130 # suma de 60, 40, 3

>>> suma_diagonal([[20, 50, 60, 70, 80], [15, 20, 16, 40, 50], [30, 56, 60, 25, 30], [41, 85, 90, 64, 70], [68, 43, 12, 24, 16]], -3)

>>> suma_diagonal([[20, 50, 60, 70, 80], [15, 20, 16, 40, 50], [30, 56, 60, 25, 30], [41, 85, 90, 64, 70], [68, 43, 12, 24, 16]], 8)
ERROR: NO EXISTE LA DIAGONAL

>>> suma_diagonal([[20, 50, 60], [15, 20, 16]], 1)
ERROR: NO ES CUADRADA

Ejercicio 8

Desarrolle la función **primos_pal**. La función recibe dos números enteros: el inicio y el fin de un rango (inicio >0, fin > 0, inicio <= fin). La función debe hacer lo siguiente:

- Debe validar que los dos números de entrada sean números binarios, es decir números formados solo por 0's y 1's. En caso de no cumplir con esto se imprime el mensaje "ERROR: LAS ENTRADAS DEBEN SER NÚMEROS BINARIOS" y la función termina. Si las entradas están bien la función sigue ejecutando.
- Para los números que están en el rango de entrada hay que imprimir solamente aquellos cuya representación decimal cumpla con estas dos condiciones:
    - que sea un número primo y
    - palíndromo.

En caso de no haber números con estas condiciones imprimir el mensaje "NO HAY NÚMEROS CUYA REPRESENTACIÓN DECIMAL SEA UN PRIMO Y PALÍNDROMO A LA VEZ". La impresión debe ser según ejemplos del funcionamiento:

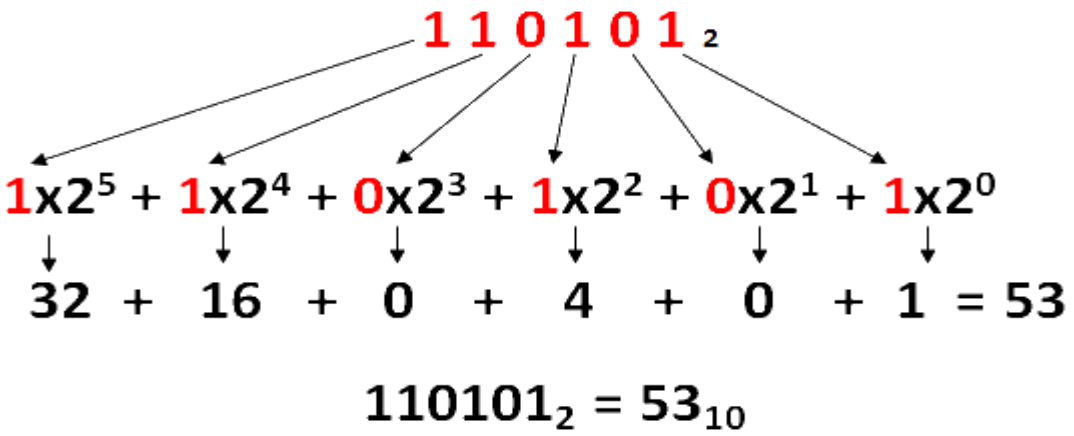
```
>>> primos_pal(10, 501)
ERROR: LAS ENTRADAS DEBEN SER NÚMEROS BINARIOS

>>> primos_pal(110, 1101)        # del 6 al 13
7 es primo y palíndromo
11 es primo y palíndromo

>>> primos_pal(101101001, 1011101110)        # del 361 al 750
373 es primo y palíndromo
383 es primo y palíndromo
727 es primo y palíndromo

>>> primos_pal(1100, 1110)     # del 12 al 14
NO HAY NÚMEROS CUYA REPRESENTACIÓN DECIMAL SEA UN PRIMO Y PALÍNDROMO A LA VEZ
```

La conversión de un número binario a su representación decimal se debe realizar con el siguiente procedimiento: cada dígito del número a convertir se multiplica por una potencia de 2. El último dígito del número, es decir el que está más a la derecha, se multiplica por 2 con potencia 0, el penúltimo dígito se multiplica por 2 con potencia 1 y así sucesivamente se va aumentando la potencia de 2 según la posición de los dígitos. La suma de todas las multiplicaciones da el número convertido. Esto se muestra en el ejemplo:



$$110101_2 = 53_{10}$$

Ejercicio 9

Haga la función **factorial**. El factorial de un número n, denotado como n!, está definido solo para enteros naturales y es el producto de todos los números enteros desde 1 hasta n:

n! = 1 * 2 * 3 … * (n – 1) * n,

por definición; 0! = 1.

Ejemplos del funcionamiento:

```
>>> factorial(5)
120
```

¿ cómo se calculó ? 1 * 2 * 3 * 4 * 5

```
>>> factorial(0)
1

>>> factorial(4)
24
```

Ejercicio 10

Haga la función **triángulo_de_pascal**. Recibe un entero n (>= 1) y retorna una lista de tuplas representando el triángulo de Pascal hasta la fila n.
En matemáticas el triángulo de Pascal es una representación de los coeficientes binomiales ordenados en forma triangular. Se le llama así en honor al matemático y filósofo francés Blaise Pascal quien introdujo la notación.
La construcción del triángulo está relacionada con los coeficientes binomiales. Cada elemento del triángulo se puede calcular según la fórmula combinatoria sin repetición (llamada también regla de Pascal):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$ donde n es la fila – 1, y k es la posición del elemento - 1en la fila

El cálculo de cada elemento del triangulo sería así para n = 4:

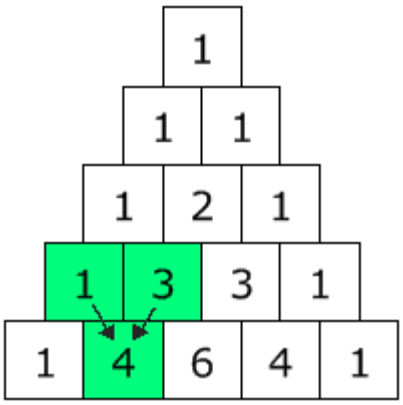
				n
		1 (0,0)		1
	1 (1,0)		**1** (1,1)	2
1 (2,0)		**2** (2,1)		3
1 (3,0)	**3** (3,1)		**3** (3,2)	4

Ejemplo del funcionamiento:

```
>>> triángulo_de_pascal(4)
[ (1,), (1, 1), (1, 2, 1), (1, 3, 3, 1) ]
```

Ejercicio 11

Haga la función **triángulo_de_pascal_2**. Recibe un entero n (>= 1) y retorna una lista de tuplas representando el triángulo de Pascal hasta la fila n.
Otra forma de construir este triángulo: se pone un 1 en la cima. También se pone un 1 en los extremos de cada línea: a la izquierda y a la derecha. Los números dentro del triángulo representan la suma de los dos números adyacentes que están encima según se muestra en la figura siguiente para n = 5. En este algoritmo se debe calcular la siguiente tupla (un nivel del triángulo) basada en los valores de la tupla anterior.



n
1
2
3
4
5

Ejemplo del funcionamiento:

>>> triángulo_de_pascal_2(5)
[(1,), (1, 1), (1, 2, 1), (1, 3, 3, 1), (1, 4, 6, 4, 1)]

Ejercicio 12

Haga la función ***cuenta_palabras*** que reciba dos tuplas: una donde cada elemento es una palabra y otra donde cada elemento es una frase. En la función debe construir y retornar un diccionario donde cada elemento tiene la estructura:

palabra : cantidad

donde palabra es cada palabra de la tupla de palabras y cantidad es la cantidad de veces que aparece dicha palabra en la tupla de las frases.

Ejemplo del funcionamiento:

>>> cuenta_palabras((“calor”, “ayer”, “el”, “mañana”), (“ayer hizo bastante calor”, “en el laboratorio hace calor”))

{ “calor”: 2, “ayer”: 1, “el”: 1, “mañana”: 0 }

Ejercicio 13

Haga la función **rombo** que reciba un valor entero n (>=2) que representa el tamaño de cada uno de sus cuatro lados e imprima un rombo de “*”.

Ejemplo del funcionamiento para n = 5:

```
>>> rombo(5)
        *
       ***
      *****
     *******
    *********
     *******
      *****
       ***
        *
```

Ejercicio 14

En teoría de números la factorización de enteros o factorización de primos consiste en expresar un número compuesto (número no primo) como un producto de factores primos. Un procedimiento para obtener esta factorización consiste en tomar el número y dividirlo por el menor número primo posible (2 o 3 o 5, etc.), luego el cociente lo dividimos por el mismo primo de ser posible, sino seguimos probando con el siguiente primo, y así sucesivamente hasta que el cociente sea igual al valor 1. Los divisores calculados son los factores primos. Ejemplos:

120	2
60	2
30	2
15	3
5	5
1	

2 x 2 x 2 x 3 x 5

33	3
11	11
1	

3 x 11

Haga la función **factores_primos** que reciba un entero (>= 2) e imprima su factorización según el procedimiento explicado. En caso de recibir un número primo se imprime el mismo número.

Ejemplos del funcionamiento:

```
>>> factores_primos(120)
2 x 2 x 2 x 3 x 5

>>> factores_primos(33)
3 x 11

>>> factores_primos(11)
11
```

Ejercicio 15

Haga la función **es_identidad** que reciba una matriz de números y determine si es una matriz identidad. En caso de ser una matriz identidad retorna True, sino retorna False. La matriz identidad es una matriz cuadrada (es decir, tiene el mismo número de filas y columnas) en la cual todos los elementos de la diagonal principal son iguales a 1 y todos los demás elementos son iguales a 0. La diagonal principal es la que va desde la esquina superior izquierda hasta la esquina inferior derecha.

A continuación un ejemplo de una matriz identidad de tamaño 3:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Observe que los elementos de la diagonal se encuentran donde el índice de fila es igual al índice de columna, en este caso: [0][0], [1][1], [2][2].
Valide que la matriz de entrada sea cuadrada, de lo contrario retorne un mensaje de error.

Ejemplos del funcionamiento:

```
>>> es_identidad([[1, 0, 0], [0, 1, 0], [ 0, 0, 1]])
True

>>> es_identidad([[1, 0, 0], [0, 1, 0], [ 7, 0, 1]])
False

>>> es_identidad([[5, 0, 0, ], [0, 12, 0 ], [0, 0, 7], [0, 9, 0]])
Error: matriz debe ser cuadrada
```

Última línea