Introducción

El objetivo de esta práctica es la de sentar las bases en el trabajo con matrices, el trabajo con matrices es de gran relevancia, ya que es la estructura de datos más utilizada tanto en el mundo de la programación. Por ello, es importante que dediquéis tiempo y cariño a los siguientes ejercicios. Es una tarea que requiere abstracción y mucha paciencia, y en muchos casos deberéis reintentar realizar los ejercicios varias veces. No se espera que todo el mundo obtenga la máxima nota, es decir, que todos consigáis resolver todos los ejercicios. En caso de que no dispongáis del tiempo suficiente o no consigáis solucionar un ejercicio, se espera que lo dejéis en blanco.

La siguiente práctica se debe realizar **individualmente**, y su autoría no debe estar en ningún caso en entredicho con la de otros estudiantes o con materiales de la red. Todas las prácticas pasarán por sistemas antiplagio. **Dada la facilidad de copia de este tipo de tarea, se asumirá su copia bajo la mínima sospecha, conllevando un 0 a todos los implicados.**

El propósito de esta actividad es que vosotros desarrolléis vuestras propias habilidades de creación de código, una habilidad que no se ve desarrollada si no sois vosotros los que implementáis la solución.

Para la realización de los ejercicios **no se deben usar funciones o librerías no vistas en clase**, en especial no se puede usar `min`, `max`, `sum`, `count`, `sorted`. En caso de cualquier duda preguntad en el foro.

Evaluación

Cada ejercicio tiene una nota asignada en puntos siendo el máximo un 10.

Cada ejercicio tiene un set de tests públicos (asserts) para comprobar que su funcionamiento es correcto, estos asserts os ayudarán durante el desarrollo de la práctica para comprobar que lo que estáis implementando cumple con lo que se pide. Pero, además, **al corregir la práctica vuestras funciones serán testeadas con otro conjunto de asserts privados**, por ello debéis leer con atención el enunciado ya que los asserts públicos no comprueban exhaustivamente todos los casos. **Todo código entregado sin su archivo replay correspondiente no será corregido.**

Entrega

Para realizar la entrega la debéis realizar en https://www.codereplay.dev/, al finalizar cada ejercicio debéis pulsar dos botones "save file" y "save replay". Save file, os guardará vuestro código final, y save replay guardará como habéis generado este código. Se deben entregar ambos ficheros para cada ejercicio, renombrando los ficheros con el nombre del ejercicio, por ejemplo saludos_email.py y saludos_email.json. **Todo código entregado sin su archivo replay correspondiente no será corregido.**

Imprimir matriz (1 punto)

Crea una función que imprima todos los elementos de la matriz, uno por cada línea

In [ ]:
```python
def imprimir_matriz(m):
    pass


m = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

imprimir_matriz(m)


"""
Ejemplo
imprimir_matriz(m)

Resultado:
1
2
3
4
5
6
7
8
9

"""
```

Suma elementos de una matriz (1 punto)

Crea una función que devuelva la suma de todos los elementos de la matriz

In [ ]:
```python
def suma_elementos_matriz(m):
    pass


m = [[1, 2, 3],
     [2, 2, 2],
     [1, 2, 3]]

m2 = [[1, 2, 3],
      [2, 2, 2],
      [1, 2, 3],
      [1, 2, 3]]
assert(suma_elementos_matriz(m) == 18)
assert(suma_elementos_matriz(m2) == 24)
```

n_diagonal (1 punto)

Definimos una matriz n_diagonal como una matriz que tiene `n` o menos elementos distintos de cero fuera de la diagonal. Crea una función que tiene como parámetros de entrada, una matriz `m` y un número `n`. La función debe devolver `True` en caso de que sea n_diagonal y `False` en caso contrario. Puedes asumir que `m` es cuadrada.

In [ ]:
```python
def n_diagonal(m, n):
    pass



m_diagonal = [[1, 0, 0],
              [0, 7, 0],
              [0, 0, 3]]
assert(n_diagonal(m_diagonal, 0) == True)

m_no_diagonal = [[1, 0, 2],
                 [0, 7, 0],
                 [0, 0, 3]]
assert(n_diagonal(m_no_diagonal, 1) == True)

m_no_diagonal = [[1, 0, 2],
                 [0, 7, 0],
                 [0, 3, 3]]
assert(n_diagonal(m_no_diagonal, 2) == True)

m_no_diagonal = [[0, 0, 2],
                 [0, 7, 0],
                 [2, 3, 3]]
assert(n_diagonal(m_no_diagonal, 2) == False)
```

Máximo y Mínimo de una matriz (1 punto)

Crea una función max_min, que retorne el mínimo y el máximo elemento de una matriz. **Nota**: Para devolver más de una variable en una función se añaden las dos variables después del return separadas con comas. Por ejemplo: `return elemento_maximo, elemento_minimo`

In [ ]:
```python
def max_min(m):
    pass

m = [[1, 0, 2],
     [0, 7, 0],
     [0, 0, 3]]

m_1_elemento = [[10]]

assert(max_min(m) == (7, 0))
assert(max_min(m_1_elemento) == (10, 10))
```

Matrices bonitas (1 punto)

Definimos una matriz bonita como una matriz con la siguiente estructura:

$$, \begin{bmatrix} a & b & b & b \\ c & a & b & b \\ c & c & a & b \\ c & c & c & a \end{bmatrix},$$

Donde `a`, `b` y `c` son números naturales.

Implementa una función `bonita(m)` que dada una matriz m, devuelva sí es bonita o no. `True` en caso de que sea bonita, `False` en caso contrario. Puedes asumir que la matriz es cuadrada y que tiene como mínimo un tamaño de 2x2.

In [ ]:
```python
def bonita(m):
    pass


mat = [[2, 3, 3, 3],
       [1, 2, 3, 3],
       [1, 1, 2, 3],
       [1, 1, 1, 2]]
assert(bonita(mat))
mat = [[2, 3, 3, 3],
       [1, 2, 3, 3],
       [1, 1, 2, 2],
       [1, 1, 1, 2]]
assert(not bonita(mat))

mat_strings = [["Ciao", "Hola", "Hola", "Hola"],
               ["Adios", "Ciao", "Hola", "Hola"],
               ["Adios", "Adios", "Ciao", "Hola"],
               ["Adios", "Adios", "Adios", "Ciao"]]
assert(bonita(mat_strings))
```

Transpuesta (1 punto)

Crea una función `es_tranpuesta(m1, m2)` que dadas dos matrices diga si `m2` es la matriz transpuesta de `m1`. Puedes asumir que m1 y m2 son cuadradas y del mismo tamaño.

In [ ]:
```python
def es_transpuesta(m1, m2):
    pass


mat1 = [[2, 0, 0, 0],
        [3, 2, 0, 0],
        [3, 3, 2, 0],
        [3, 3, 3, 2]]

mat2 = [[2, 3, 3, 3],
        [0, 2, 3, 3],
        [0, 0, 2, 3],
        [0, 0, 0, 2]]


mat3 = [[1, 1, 1, 0],
        [1, 1, 0, 0],
        [1, 0, 0, 0],
        [0, 0, 0, 0]]

mat4 = [[0, 0, 0, 0],
        [0, 0, 0, 1],
        [0, 0, 1, 1],
        [0, 1, 1, 1]]


assert(es_transpuesta(mat1, mat2))
assert(not es_transpuesta(mat3, mat4))
```

Suma Matrices (1 punto)

Crea la función `suma_matrices(m1, m2)` que dadas dos matrices devuelva su suma. Para inicializar la matriz resultado podéis usar la función crear_matriz_zeros(n_fila, n_columnas) que devuelve una matriz del tamaño marcado con todos sus elementos a 0.

In [ ]:
```python
def crear_matriz_zeros(n_filas, n_columnas):
    m = list()
    for i in range(n_filas):
        m.append(list())
        for j in range(n_columnas):
            m[i].append([0])
    return m

def suma_matrices(m1, m2):
    pass

mat1 = [[2, 0, 0, 0],
        [3, 2, 0, 0],
        [3, 3, 2, 0],
        [3, 3, 3, 2]]

mat2 = [[2, 3, 3, 3],
        [0, 2, 3, 3],
        [0, 0, 2, 3],
        [0, 0, 0, 2]]

print(suma_matrices(mat1, mat2))
```

Suma bordes (1.5 puntos)

Crea una función `suma bordes` que dada una matriz sume todos los elementos de sus bordes exteriores. Por ejemplo, en la siguiente matriz 4x4 se marcan los elementos borde con la letra X.

$$\text{‘}\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}\text{‘}$$

$$\text{‘}\begin{bmatrix} X & X & X & X \\ X & f & g & X \\ X & j & k & X \\ X & X & X & X \end{bmatrix}\text{‘}$$

In [ ]:
```python
def suma_bordes(m):
    pass


mat1 = [[2, 0, 8, 0],
        [3, 2, 0, 6],
        [3, 3, 2, 0],
        [3, 3, 3, 2]]

assert (suma_bordes(mat1) == 33)


mat2 = [[1, 2, 3, 4],
        [3, 2, 0, 6],
        [3, 3, 2, 0],
        [3, 3, 3, 2],
        [3, 3, 3, 2]]
assert (suma_bordes(mat2) == 38)
```

Solo 1 (1.5 puntos)

Crea una función `solo_1(m)` que dada una matriz binaria compruebe que en cada fila y en cada columna tiene exactamente un 1.

In [4]:

```
def solo_1(m):
    pass



m1 = [[0, 0, 1],
      [1, 0, 0],
      [0, 1, 0]]
assert(solo_1(m1))

m2 = ([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1]])
assert(not solo_1(m2))


m3 = ([[0, 1],
       [1, 0]])

assert(solo_1(m3))

m4 = ([[0, 0],
       [1, 0]])

assert(not solo_1(m4))

m5 = ([[1, 1, 0],
       [0, 0, 0],
       [0, 0, 1]])
assert(not solo_1(m5))
```