Tarea 1:

Autor: Miguel Angel Citarella

Correo: mcitarella@unal.edu.co

CC: 1021632687

Ejercicio 1:

"Implemente un notebook donde dado un n particular indexando los números primos se provean ejemplos que cumplan/o no cumplan la conjetura de Brocard's"

Para solucionar este problema primero hay saber que un número primo es aquel que solo se puede dividir entre sí mismo y 1 para resultar en un número entero. También hay que entender de qué se trata la conjetura de Brocard's. Esta propone que, si se enumeran los números primos, y se escoge el n-esimo, para n>1, y el n+1-esimo, y se elevan ambos al cuadrado, entre ellos se van a encontrar al menos otros 4 números primos.

La primera parte de esta solución consistía en realizar una función que pudiera identificar un número primo. Para esto, se construyó una función que tomaba un entero n, y ciclaba entre números desde el 2 hasta la raíz de n y verificaba si el módulo de n con algunos de esos enteros era 0. Si tal era el caso, significaba que n tenía un divisor en algún número probado, y por lo tanto no era primo. Se buscaban números hasta la raíz de n porque un número se puede expresar como el producto de 2 números: x*w = n. Si se comienzan a tomar números desde 1 hasta n para x, entonces w irá disminuyendo su valor. En el momento que x es igual a la raíz de n, entonces w es igual a x, y si desde este punto x continúa aumentando, entonces el valor de w ira disminuyendo, como ocurría anteriormente. Esto significa que se realizarían operaciones redundantes, porque se están realizando pruebas para números x>raíz(n), pero cuando x era menor a la raíz de n, realizar n/x (división intrínseca necesaria para obtener el módulo de n) habría resultado en un número mayora esta marca (y si llegaba a ser entero entonces se probaría que n no es primo), y sin embargo ahora x se está moviendo por ese rango de valores mayores a la marca. Con este conocimiento en mente, la función resultante fue:

```
def is_Prime(n: int)-> bool:
        '''
            Checks if n is prime.
        '''
        for i in range(2, int(sqrt(n))):
            if n % i == 0:
                return False
        else:
            return True
```

También era necesario crear una función que pueda encontrar el n-esimo primo. Para lograr tal objetivo se comenzó creando una sub-clase llamada Prime_Num, derivada de la clase int, que tenía la capacidad de realizar operaciones aritméticas como un entero normal, pero que le agregaba al objeto un atributo llamado n, en el que se contenía el índice de tal número primo. Esto facilitaba el funcionamiento de la función que se busca crear, porque ella se envía el índice del n-esimo primo que desee el usuario, después se calculaban uno por uno los números primos,

con su respectivo índice, hasta que se encontraba el que se encontraba bajo el índice determinado por el usuario:

```python
class Prime_Num(int):
    def __new__(cls, *, value: int, nth: int):
        num = super().__new__(cls, value)
        num.n = nth
        return  num

    def __add__(self, other: int):
        res = super(Prime_Num, self).__add__(other)
        return self.__class__(value=res, nth=self.n)

    def __divmod__(self, value: int) -> tuple[int, int]:
        return super().__divmod__(value)

    def __pow__(self, pow):
        return super().__pow__(pow)
```

```python
def nth_Prime(n: int) -> Prime_Num:
        '''
            Finds the nth prime number.
        '''
        num = Prime_Num(value=2, nth=1)
        while True:
            num+=1
            if is_Prime(num):
                num.n+=1
                if num.n == n:
                    return num
```

La última función que quedaba por crear era aquella que recibía un rango y buscaba los números primos intermedios, mediante el apoyo de las dos funciones anteriores:

```python
def find_Primes(a: int, b: int) -> Generator[int, None, int]:
    '''
        Return primes between a and b.
    '''
    for n in range(a+1, b):
        if is_Prime(n):
            yield n
```

Ahora, con estas herramientas en mente, se construyó la función principal, con la que se tomaba el índice del primo desde el que se quería probar el teorema (mediante una entrada por consola o por argumentos en la línea de comando). Se comenzaba calculando tales primos, después se partía a elevarlos al cuadrado y entre esos dos resultados encontrar los valores primos.

Al final, por conveniencia, se optó por mostrar en la consola solamente 4 números primos encontrados, si es que llegan a haber más de 10, para facilitar la visualización del usuario y demostrando con igual autoridad el teorema. Sin embargo, los otros valores quedan guardados en la lista de primos resultantes, por si más adelante se busca extender las capacidades del programa y se requiera el uso de tales números calculados. El resultado final fue:

```python
def main() -> None:
    try:
        n = argv[1]
    except IndexError:
        n = input('n>> ')
    finally:
        if n.isdigit() and not n == '1':
            n = int(n)
            p1, p2 = nth_Prime(n), nth_Prime(n+1)

            print(f"Your primes are: {p1}->{p2} ; {p1**2}->{p2**2}")

            primes = list()

            for prime in find_Primes(p1**2, p2**2):
                primes.append(prime)

            sumarize = lambda p_list: print("Proof:", *p_list[:4])

            print("List:", *primes) if 3<len(primes)<10 else sumarize(primes)


if __name__ == "__main__":
    main()
```

Ejercicio 2:

“Verifique utilizando un programa que la conjetura de la suma de potencias de Euler es falsa”

Así como para el primer ejercicio, en primer lugar es necesario entender en qué consiste la conjetura de la suma de potencias de Euler: esta propone que, si se toma un k y un n enteros, mayores a 1, y llega a ocurrir que un número entero positivo b elevado a la k, se puede expresar como la suma de n enteros positivos también elevados a la k, entonces n es mayor o igual a k.

La parte central para la construcción de un contra ejemplo, era crear una función capaz de construir el escenario planteado y verificar que se cumpla la igualdad. Para esto se construyó una lista, que contenía los n números que se querían probar, se procedía a elevarlos todos a la k y sumarlos, para después sacar la k-esima raíz de tal valor y guardarlo en una variable “resultado”. Si los n números de la lista cumplen con la igualdad, entonces el “resultado” sería un número entero positivo. Con esto en mente, se construyó la siguiente función:

```
def check_Euler(k: int, p_ints:list) -> None:
        '''Tests for the given exponent k if the coonjecture

        stands with the n<k numbers in the 'p_ints' list.
        '''
        sum_ints = sum(map(lambda a_n:pow(a_n,k), p_ints))
        result = pow(sum_ints, (1/k))
        if int(result) <= result <= int(result)+0.000000000001:
            counter_ex.append({f'Exponent: {k}':f'{p_ints} -> {int(result)}'})
```

Antes de continuar, es importante denotar que se importó la librearía estándar “math” para poder acceder realizar algunas operaciones, como la exponenciación, de forma más veloz. También hay que mencionar que la comparación realizada no fue con el signo de igualdad “==” para evitar sobrepasar las inexactitudes relacionadas a cálculos de números de punto flotante.

Ahora, con la función completa, se pasaron los 4 números a probar como contraejemplos de la conjetura. Si en efecto lo eran, entonces el programa guardaba los n números, así como el “resultado” en una lista dónde se buscaba almacenar todo el conjunto de valores que sirvan de contraejemplos,y ser presentados antes de terminar el programa.

```
def disproof(k=4):
    check_Euler(k=k, p_ints=[95800, 217519, 414560])

global counter_ex
counter_ex = list()
disproof()
```

```
if __name__ == "__main__":
    main()
    [print(example) for example in counter_ex]
```

Adicionalmente se implementaron una serie de funciones que buscaban encontrar por fuerza bruta todos los contraejemplos posibles para la conjetura. El usuario en este caso podía escoger para que casos de k podía buscar un contra ejemplo, y el resto de las funciones se iban a encargar de encontrar la suma n números elevados a la k, tal que n<k, y que la k-esima raíz del resultado sea también un número entero positivo.

La primera función “Euler” buscaba encontrar todas las combinaciones de n números, tal que n<k y cada número tomaba un valor máximo de 50 (para reducir el tiempo de ejecución del programa, aunque para alcanzar su máxima funcionalidad habría que aumentar este valor lo que más se pueda). Esta función trabaja bajo recursividad, dónde en el caso base es que se utiliza la función previamente establecida para evaluar los contenidos de la lista.

```
def euler(first_call:bool, k=1, n=1) -> None:
    '''Get all the possible combinations of numbers

    between 1 and 'max'.
    '''
    i = n-1
    if n == 1:
        if len(p_ints) > 1:
            for _ in range(p_ints[n], max+1):
                check_Euler(k, p_ints)
                p_ints[0]+=1
        return
    elif first_call:
        euler(first_call=True, k=k, n=i)
        p_ints.append(1)
        p_ints[:i] = [p_ints[i]] * i

    for _ in range(max):
        euler(first_call=False, k=k, n=i)
        p_ints[i]+=1
        p_ints[:i] = [p_ints[i]] * i
```

Los programas completos se encuentran en:

https://drive.google.com/drive/folders/11Jc8eFxm1n4KpX6-2MN-BCjNxR7FlCkD?usp=sharing