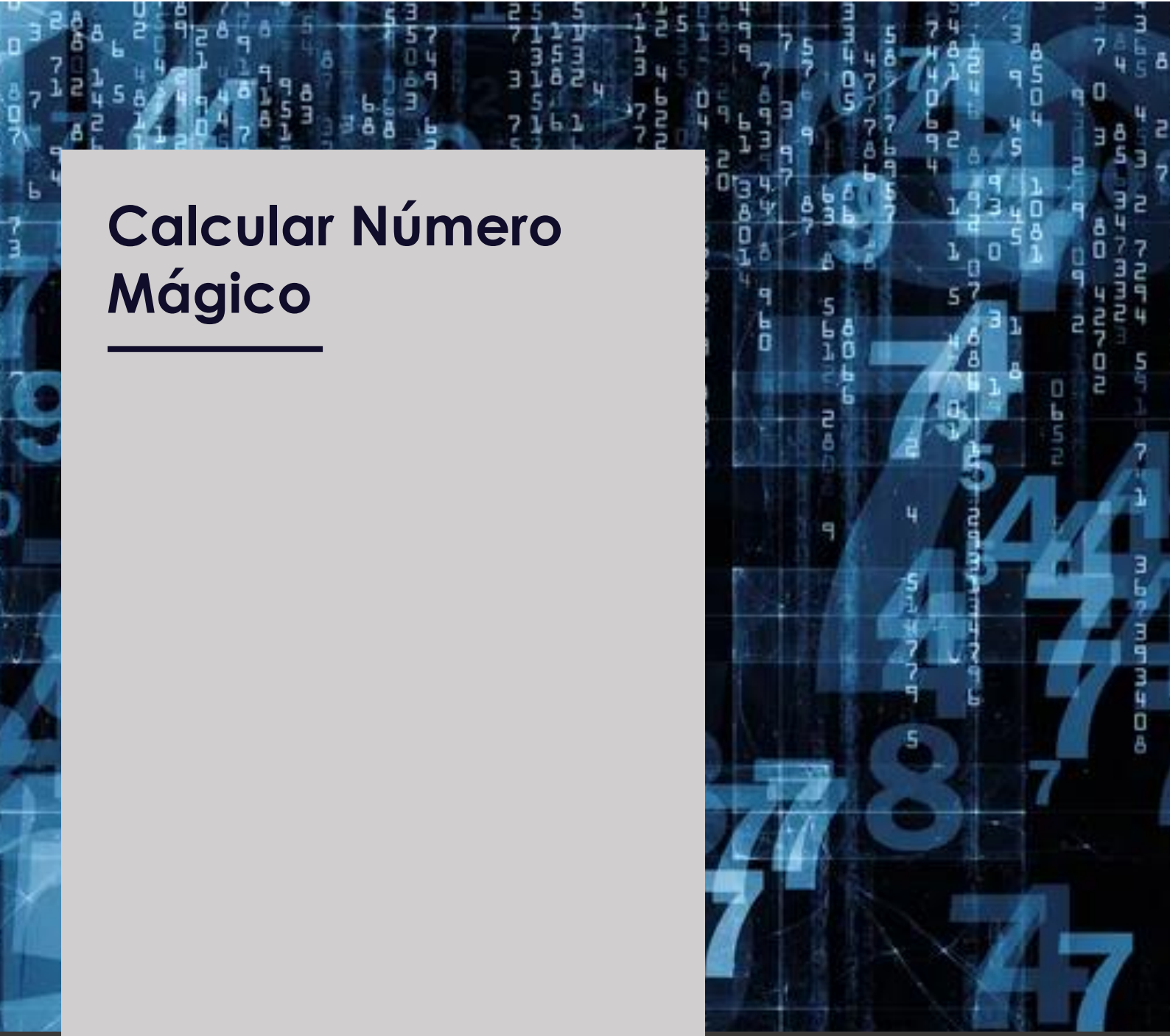


Calcular Número Mágico

12 ENERO

Asignatura: Big Data
Creadores: Jaime Gómez, Sebastián Huete, David Rodríguez, Samuel Sánchez, Víctor Vega y Álvaro Tuñón.

Indice

Introducción

Este documento describe el proceso de resolución del problema **“Cálculo de un número mágico Tuentístico”,** que implica incluir un número positivo en la suma de varios números Tuentísticos para aumentar el número de adiciones Un número tuentístico se define como aquel cuya representación en inglés incluye la palabra "twenty", como 20, 21 o 120. El objetivo principal de esta tarea es explorar algoritmos de optimización y análisis numérico en un lenguaje específico, agregando complejidad y creatividad adicionales al problema.

Para este proyecto, utilizamos Python 3.7 como lenguaje de programación y seguimos el estándar de codificación PEP8 para garantizar la claridad y calidad del código. Además, implementamos un conjunto de pruebas automatizadas utilizando pytest, garantizando solidez y precisión de la solución propuesta. Este enfoque nos permitió no solo resolver el problema, sino también asegurarnos que el código fuera escalable, mantenible y comprobable.

Planteamiento del problema

Un número tuentístico es cualquier número que, al escribirse en inglés, contenga la palabra "twenty" (por ejemplo, 20, 21 o 120000). Como en Tuenti amamos los números tuentísticos, nos gusta representar cualquier número no tuentístico como una suma de números tuentísticos positivos para aumentar su tuentísticidad (a esto lo llamamos una suma tuentística). Por ejemplo, el número no tuentístico 50 puede representarse como la suma tuentística 25 + 25. Los propios números tuentísticos también cuentan como sumas tuentísticas.

Dado un número positivo, queremos saber el número máximo de elementos que puede tener una de sus sumas tuentísticas.

Entrada

La primera línea contiene el número de casos, **C**. Luego, siguen **C** líneas, cada una con un número **N**.

Salida

Para cada caso, imprime Case #X: M, donde **X** es el número del caso (el primer caso tiene el número 1) y **M** es el número máximo de elementos en una suma tuentística de **N**. Si no es posible representar N como una suma tuentística, imprime Case #X: IMPOSSIBLE.

Límites

- $1 \leq C \leq 500$
- $1 \leq N \leq 262$

Ejemplo de Entrada:

3
20
80
35

Ejemplo de Salida:

Case #1: 1
Case #2: 4
Case #3: IMPOSSIBLE

Solución planteada

Metodología

1. **Identificar los números tuentísticos:**
   - Usamos el módulo inflect para convertir números en su representación en inglés y verificamos si contienen la palabra "twenty". Esto asegura que solo se consideren números válidos como tuentísticos.
2. **Generar una lista de números tuentísticos:**
   - Creamos una lista de números tuentísticos en el rango permitido (20≤N≤lı´mite20 \leq N \leq \text{límite}20≤N≤lı´mite), que será utilizada para calcular las descomposiciones.
3. **Búsqueda en profundidad para calcular la suma máxima:**
   - Empleamos un enfoque basado en grafos para explorar todas las posibles combinaciones de números tuentísticos que sumen hasta el número objetivo.
   - Usamos un algoritmo de búsqueda en profundidad (DFS) con una pila para rastrear las combinaciones posibles y determinar el máximo número de pasos necesarios.
4. **Paralelización con MapReduce:**
   - Implementamos la lógica de procesamiento usando multiprocessing.Pool para dividir el trabajo entre varios procesos, acelerando la resolución de múltiples casos.

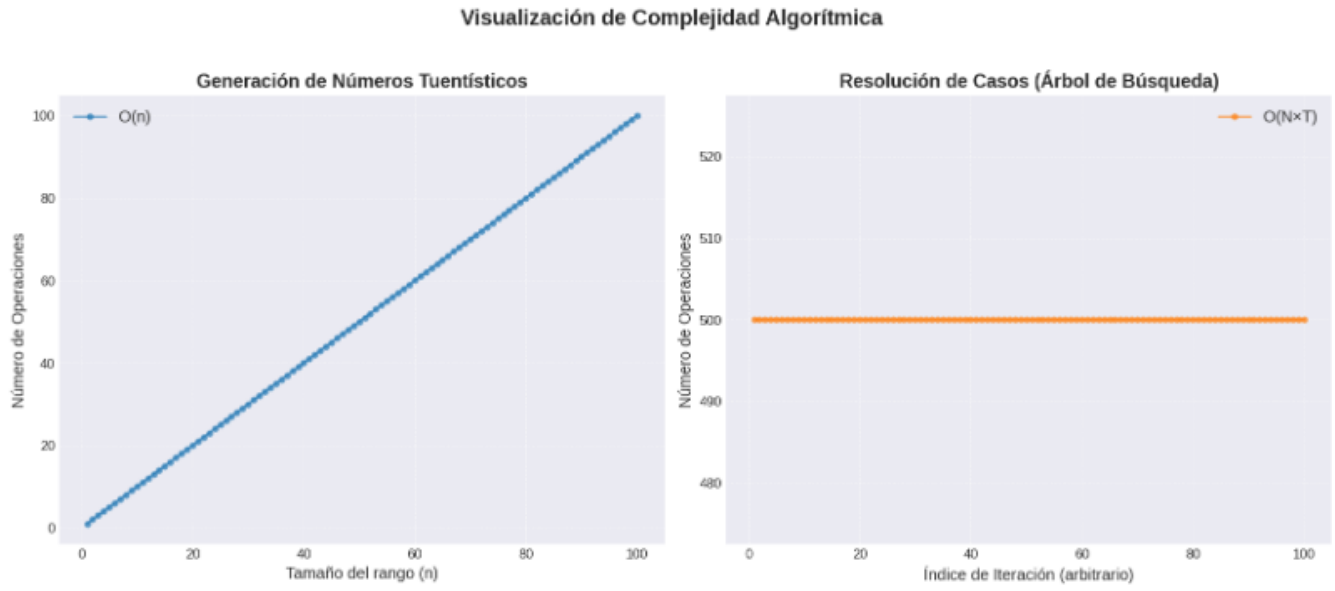
Complejidad

Generación de números tuentísticos:

- Complejidad: O(n)O(n)O(n), donde nnn es el límite superior del rango.
- Explicación: Recorremos cada número dentro del rango para determinar si es tuentístico.

Resolución de cada caso:

- Complejidad: O(N×T) O (N \times T) O (N×T), donde NNN es el número objetivo y TTT es el número de números tuentísticos.
- Explicación:
  - En el peor caso, exploramos todas las combinaciones posibles de números tuentísticos que sumen hasta NNN.
  - Esto implica restar repetidamente cada número tuentístico de NNN en un árbol de búsqueda.



Estructura del Código en Módulos

Para una mejor organización, el código se puede dividir en los siguientes archivos y carpetas:

numero_magico/

├── main.py

├── grafo_tuentistico.py

├── procesador.py

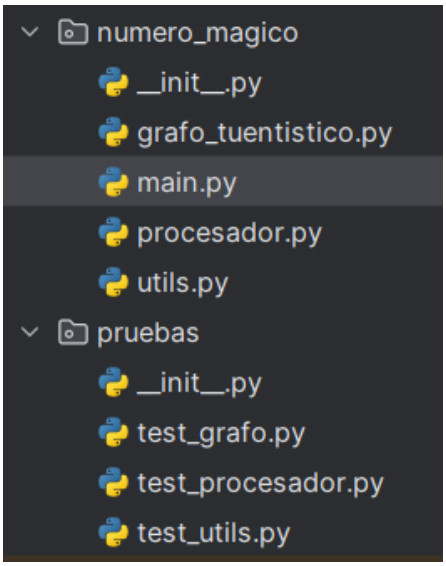
├── utils.py

├── **pruebas/**

│ ├── test_grafo.py

│ ├── test_procesador.py

│ └── test_utils.py



Detalles del Estructura:

- Directorio principal **(numero_magico/):** Contiene todos los archivos principales y subdirectorios relacionados con el proyecto.
- **Archivos principales:**
  - **main.py:** El punto de entrada que coordina la ejecución del programa.
  - **grafo_tuentistico.py**: Implementa las estructuras de datos y algoritmos principales relacionados con el grafo.
  - **procesador.py**: Contiene la lógica para descomponer los números y resolver los casos.
  - **utils.py:** Incluye funciones auxiliares como la generación de números tuentísticos.
- **Carpeta de pruebas (pruebas/):**
  - **test_grafo.py**: Evalúa la funcionalidad de las clases del grafo.
  - **test_procesador.py**: Pruebas específicas para la lógica del procesamiento.
  - **test_utils.py**: Comprueba las funciones auxiliares.

Implementación de los archivos principales

A continuación, se presenta el código desarrollado para resolver el problema:

main.py

```
from procesador import procesar_casos_mapreduce

if __name__ == "__main__":
    print("Introduce el número de casos entre 1 y 500:")
    valido = False
    cantidad_casos = 0

    while not valido:
        try:
            cantidad_casos = int(input().strip())
            if 1 <= cantidad_casos <= 500:
                valido = True
            else:
                print("El número de casos debe estar entre 1 y 500. Inténtalo
de nuevo.")
        except ValueError:
            print("Por favor, introduce un número válido.")

    casos = []

    print(f"Introduce los {cantidad_casos} números, uno por línea:")
    for _ in range(cantidad_casos):
        numero = int(input().strip())
        if 1 <= numero <= 262:
            casos.append(numero)
        else:
            print(f"El número {numero} no está en el rango permitido. Por
favor, inténtalo de nuevo.")

    resultados = procesar_casos_mapreduce(casos)

    print("\nResultados:")
    for resultado in resultados:
        print(resultado)
```

- **Propósito:** Punto de entrada del programa.
- **Contenido:**
  - Solicita al usuario el número de casos (CCC).
  - Valida que CCC esté entre 1≤C≤5001 \leq C \leq 5001≤C≤500.
  - Solicita al usuario los números de cada caso (1≤N≤2621 \leq N \leq 2621≤N≤262).
  - Llama a “**procesar_casos_mapreduce”** para procesar los casos
  - Imprime los resultados de cada caso.

grafo_tuentistico.py

```python
class GrafoTuentistico:
    def __init__(self, numeros_tuentisticos):
        """
        Inicializa el grafo con los nodos representados por números
tuentísticos.
        """
        self.numeros_tuentisticos = numeros_tuentisticos

    def generar_vecinos(self, nodo):
        """
        Genera los nodos vecinos restando números tuentísticos válidos.
        """
        return [nodo - t for t in self.numeros_tuentisticos if nodo - t >= 0]


class ExploradorGrafo:
    def __init__(self, grafo):
        """
        Inicializa el explorador del grafo.
        """
        self.grafo = grafo

    def buscar_maxima_suma(self, numero):
        """
        Realiza una búsqueda en profundidad (DFS) para encontrar el máximo
número de pasos.
        """
        pila = [(numero, 0)]  # (nodo actual, número de pasos)
        maximo_contador = 0

        while pila:
            actual, contador = pila.pop()
            for vecino in self.grafo.generar_vecinos(actual):
                if vecino == 0:
                    maximo_contador = max(maximo_contador, contador + 1)
                else:
                    pila.append((vecino, contador + 1))

        return maximo_contador if maximo_contador > 0 else "IMPOSSIBLE"
```

- **Propósito**: Define las clases que representan y operan sobre el grafo tuentístico.
- **Contenido**:
  - **class GrafoTuentistico:**
    - Representa los nodos y conexiones del grafo.
    - **“generar_vecinos”**: Genera nodos vecinos válidos restando números tuentísticos.
  - **class ExploradorGrafo:**
    - Implementa la búsqueda en el grafo.
    - **“buscar_maxima_suma”**: Encuentra el número máximo de pasos necesarios para descomponer un número en sumandos tuentísticos.

procesador.py

```python
from functools import partial
from multiprocessing import Pool
from grafo_tuentistico import GrafoTuentistico, ExploradorGrafo
from utils import generar_numeros_tuentisticos


def procesar_casos_mapreduce(casos):
    """
    Procesa una lista de casos utilizando MapReduce y devuelve los resultados para cada caso.
    """
    casos_validos = [n for n in casos if 1 <= n <= 262]

    if len(casos_validos) != len(casos):
        print("Algunos casos no están en el rango permitido (1 ≤ N ≤ 262) y serán ignorados.")

    limite = max(casos_validos) if casos_validos else 0
    numeros_tuentisticos = generar_numeros_tuentisticos(limite)

    grafo = GrafoTuentistico(numeros_tuentisticos)
    explorador = ExploradorGrafo(grafo)

    # Usar Pool para realizar el mapeo paralelo
    with Pool() as pool:
        resultados = pool.map(partial(procesar_numero, explorador=explorador), enumerate(casos, start=1))

    return resultados


def procesar_numero(caso, explorador):
    """
    Procesa un número individualmente y determina el resultado utilizando el explorador del grafo.
    """
    indice, numero = caso
    resultado = explorador.buscar_maxima_suma(numero)
    return f"Case #{indice}: {resultado}"
```

- **Propósito:** Lógica para procesar los casos utilizando MapReduce.
- **Contenido:**
  - **“procesar_casos_mapreduce”:**
    - Procesa una lista de casos en paralelo.
    - Filtra los casos inválidos (NNN fuera del rango).
    - Crea instancias de **GrafoTuentistico** y **ExploradorGrafo.**
    - Usa Pool para mapear la función **procesar_numero.**
  - “**procesar_numero**”:
    - Procesa un caso individual.
    - Llama a **buscar_maxima_suma** para determinar el resultado.

```
import inflect

def generar_numeros_tuentisticos(limite):
    """
    Genera una lista de todos los números tuentísticos menores o iguales al
límite dado.
    """
    motor = inflect.engine()
    return [n for n in range(20, limite + 1) if "twenty" in
motor.number_to_words(n)]
```

- **Propósito:** Funciones auxiliares.
- **Contenido:**
  - **generar_numeros_tuentisticos:**
    - Genera una lista de números tuentísticos (20≤N≤limite20 \leq N \leq limite20≤N≤limite).
  - **test_buscar_maxima_suma:** Comprueba que buscar_maxima_suma encuentre correctamente la suma máxima o devuelva "**IMPOSSIBLE**" si no es posible.

Implementación de las pruebas

test_grafo.py

```
import pytest
from numero_magico.grafo_tuentistico import GrafoTuentistico,
ExploradorGrafo

def test_generar_vecinos():
    numeros_tuentisticos = [20, 21, 22]
    grafo = GrafoTuentistico(numeros_tuentisticos)

    # Nodo con vecinos válidos
    assert grafo.generar_vecinos(50) == [30, 29, 28]

    # Nodo sin vecinos válidos
    assert grafo.generar_vecinos(19) == []


def test_buscar_maxima_suma():
    numeros_tuentisticos = [20, 21]
    grafo = GrafoTuentistico(numeros_tuentisticos)
    explorador = ExploradorGrafo(grafo)

    # Caso posible
    assert explorador.buscar_maxima_suma(80) == 4  # 20 + 20 + 20 + 20

    # Caso imposible
    assert explorador.buscar_maxima_suma(19) == "IMPOSSIBLE"
```

- **Objetivo**: Pruebas unitarias para las clases **GrafoTuentistico** y **ExploradorGrafo**.
- **Contenido**:
  - **“test_generar_vecinos”**: Verifica que “**generar_vecinos”** genere correctamente los nodos vecinos.
  - **“test_buscar_maxima_suma”**: Comprueba que “**buscar_maxima_suma”** encuentre correctamente la suma máxima o devuelva "**IMPOSSIBLE**" si no es posible.

test_procesador.py

```
import pytest
from numero_magico.tuentisticos import es_tuentistico,
generar_numeros_tuentisticos

def test_es_tuentistico():
    assert es_tuentistico(20) is True
    assert es_tuentistico(19) is False

def test_generar_numeros_tuentisticos():
    assert generar_numeros_tuentisticos(30) == [20, 21, 22, 23, 24, 25, 26,
27, 28, 29]import pytest
from numero_magico.procesador import procesar_casos_mapreduce,
procesar_numero
from numero_magico.grafo_tuentistico import GrafoTuentistico,
ExploradorGrafo
```

```python
def test_procesar_numero():
    numeros_tuentisticos = [20, 21]
    grafo = GrafoTuentistico(numeros_tuentisticos)
    explorador = ExploradorGrafo(grafo)

    # Caso válido
    caso = (1, 40)
    resultado = procesar_numero(caso, explorador)
    assert resultado == "Case #1: 2"  # 20 + 20

    # Caso imposible
    caso = (2, 19)
    resultado = procesar_numero(caso, explorador)
    assert resultado == "Case #2: IMPOSSIBLE"

def test_procesar_casos_mapreduce():
    casos = [40, 19, 80]
    resultados_esperados = [
        "Case #1: 2",  # 20 + 20
        "Case #2: IMPOSSIBLE",
        "Case #3: 4"   # 20 + 20 + 20 + 20
    ]
    resultados = procesar_casos_mapreduce(casos)
    assert resultados == resultados_esperados
```

- **Objetivo:** Pruebas para las funciones en procesador.py.
- **Contenido:**
  - **“test_procesar_numero”**: Verifica que un caso individual se procese correctamente.
  - **“test_procesar_casos_mapreduce”**: Prueba el procesamiento completo de múltiples casos usando MapReduce.

test_utils.py

```python
import pytest
from numero_magico.utils import generar_numeros_tuentisticos

def test_generar_numeros_tuentisticos():
    # Caso con límite bajo
    assert generar_numeros_tuentisticos(25) == [20, 21, 22, 23, 24, 25]

    # Caso con límite exacto
    assert generar_numeros_tuentisticos(30) == [20, 21, 22, 23, 24, 25, 26,
27, 28, 29]

    # Caso sin tuentísticos
    assert generar_numeros_tuentisticos(19) == []
```

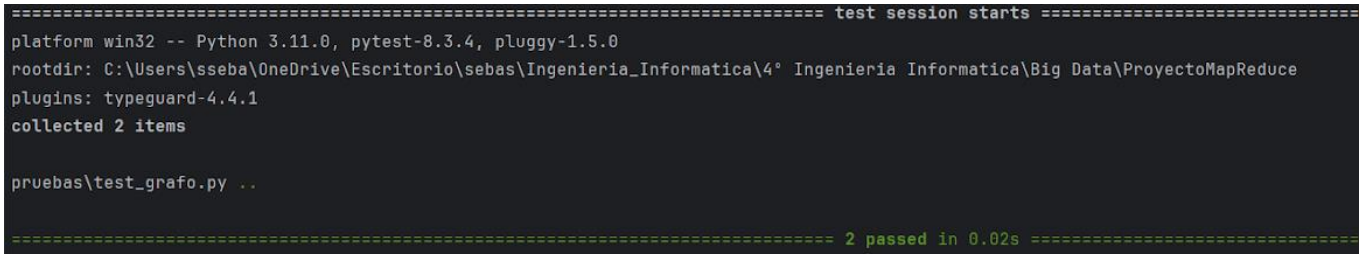
- **Objetivo:** Pruebas para las funciones auxiliares en utils.py.
- **Contenido:**
  - **“test_generar_numeros_tuentisticos”:** Verifica que se generen correctamente los números tuentísticos para diferentes límites.

Pruebas automáticas

Pruebas en test_grafo.py

1. **test_generar_vecinos:**
   - **Descripción**: Verificamos que la función generar_vecinos del grafo construya correctamente los nodos vecinos para un número dado.
   - **Pruebas**:
     - Para el nodo 50 con números tuentísticos [20, 21, 22], los vecinos esperados son [30, 29, 28].
     - Para el nodo 19, no debería haber vecinos válidos, ya que es menor que el menor número tuentístico.
   - **Resultado**: Todas las aserciones pasaron correctamente.

2. **test_buscar_maxima_suma:**
   - **Descripción**: Evaluamos el método de búsqueda del número máximo de sumandos tuentísticos que componen un número objetivo.
   - **Pruebas**:
     - Para 80 con números [20, 21], el resultado esperado es 4 (20 + 20 + 20 + 20).
     - Para 19, el resultado esperado es "IMPOSSIBLE", ya que no puede ser compuesto por números tuentísticos.
   - **Resultado**: Todas las pruebas pasaron sin errores.



Pruebas en test_procesador.py

1. **test_es_tuentistico:**
   - **Descripción**: Comprobamos si un número es identificado correctamente como tuentístico.
   - **Pruebas**:
     - 20 debe ser reconocido como tuentístico.
     - 19 no debe ser reconocido como tuentístico.
   - **Resultado**: Pruebas exitosas.
2. **test_generar_numeros_tuentisticos:**
   - **Descripción**: Validamos la generación de todos los números tuentísticos hasta un límite dado.
   - **Prueba**:
     - Hasta 30, los números generados deben ser [20, 21, 22, 23, 24, 25, 26, 27, 28, 29].
   - **Resultado**: Correcto.
3. **test_procesar_numero:**
   - **Descripción**: Probamos la función que procesa un caso individual y determina cuántos sumandos o si es imposible alcanzar un objetivo.
   - **Pruebas**:
     - Para el caso (1, 40), el resultado esperado es "Case #1: 2".
     - Para el caso (2, 19), el resultado esperado es "Case #2: IMPOSSIBLE".
   - **Resultado**: Pasaron todas las pruebas.

4. **test_procesar_casos_mapreduce:**
    - **Descripción**: Validamos la función que procesa múltiples casos y retorna los resultados en un formato estándar.
    - **Pruebas**:
        - Para los casos [40, 19, 80], los resultados deben ser:

          "Case #1: 2"
          "Case #2: IMPOSSIBLE"
          "Case #3: 4"

    - **Resultado**: Todos los resultados fueron correctos.

```
============================= test session starts =============================
platform win32 -- Python 3.11.0, pytest-8.3.4, pluggy-1.5.0
rootdir: C:\Users\sseba\OneDrive\Escritorio\sebas\Ingenieria_Informatica\4° Ingenieria Informatica\Big Data\ProyectoMapReduce
plugins: typeguard-4.4.1
collected 2 items

pruebas\test_procesador.py ..

============================== 2 passed in 3.14s ==============================
```

Pruebas en test_utils.py

1. **test_generar_numeros_tuentisticos:**
    - **Descripción:** Validamos la generación de números tuentísticos en distintos límites.
    - **Pruebas:**
        - Límite bajo (25): Deben generarse [20, 21, 22, 23, 24, 25].
        - Límite exacto (30): Deben generarse [20, 21, 22, 23, 24, 25, 26, 27, 28, 29].
        - Sin tuentísticos (19): Resultado esperado es una lista vacía.
    - **Resultado:** Todas las pruebas pasaron exitosamente.

```
============================= test session starts =============================
platform win32 -- Python 3.11.0, pytest-8.3.4, pluggy-1.5.0
rootdir: C:\Users\sseba\OneDrive\Escritorio\sebas\Ingenieria_Informatica\4° Ingenieria Informatica\Big Data\ProyectoMapReduce
plugins: typeguard-4.4.1
collected 1 item

pruebas\test_utils.py .

============================== 1 passed in 1.18s ==============================
```

Pruebas manuales

Caso base

1. **Número de casos:**
   - Se especifica cuántos cálculos o problemas diferentes se desean resolver.
   - Ejemplo: Si se elige "3", se procesarán tres números objetivo.
2. **Valores para calcular:**
   - Se ingresan los números para los cuales se desea realizar el cálculo del número mágico tuentístico.
   - Cada valor debe cumplir con los límites establecidos (entre 1 y 262).
   - Ejemplo: Si se introducen los valores 20, 33 y 35, el sistema calculará para cada uno la solución correspondiente.

```
Introduce el número de casos entre 1 y 500:
3
Introduce los 3 números, uno por línea:
20
33
35

Resultados:
Case #1: 1
Case #2: IMPOSSIBLE
Case #3: IMPOSSIBLE
```

Caso fallo 1

1. **Número de casos fuera del rango:**
   - Se especifica un número de casos mayor a 1 o menor a 500.
2. **Validación de entrada:**
   - El sistema detectará que el número ingresado no cumple con las restricciones y, por lo general, mostrará de nuevo el mensaje indicando que los valores deben estar entre 1 y 500.
   - Este proceso asegura que solo se procesen casos válidos dentro del rango establecido.

```
Introduce el número de casos entre 1 y 500:
-1
El número de casos debe estar entre 1 y 500. Inténtalo de nuevo.
501
El número de casos debe estar entre 1 y 500. Inténtalo de nuevo.
```

Caso fallo 2

1. **Selección de números tuentísticos fuera del rango:**
   - El usuario introduce números objetivo que no cumplen con los límites establecidos para los números tuentísticos.
2. **Validación de los valores:**
   - El sistema verifica cada número objetivo introducido para asegurarse de que esté dentro del rango válido.
   - Si un número está fuera del rango, el programa emite de nuevo el.

```
Introduce el número de casos entre 1 y 500:
5
Introduce los 5 números, uno por línea:
-1
El número -1 no está en el rango permitido. Por favor, inténtalo de nuevo.
300
El número 300 no está en el rango permitido. Por favor, inténtalo de nuevo.
263
```

Conclusión

Este proyecto ha permitido resolver el problema de descomposición en números tuentísticos mediante un enfoque eficiente basado en grafos y búsqueda en profundidad. El uso de pruebas unitarias ha sido fundamental para verificar su funcionamiento y precisión. En cuanto a los aprendizajes, ha reforzado habilidades clave en el diseño de algoritmos y el desarrollo estructurado. Además, ha facilitado el aprendizaje del trabajo colaborativo y la gestión de proyectos en equipo mediante el uso de GitHub.