TP11 Gestión de Calidad

Alumno: Borda Alexander

Prof: Pedro E. Colla

Ingeniería de Software

2025

Índice

1) Resumen del problema

Implementar una función que reciba un entero positivo por teclado (máximo 1999), calcule la secuencia de Collatz y devuelva/imprima el número de iteraciones necesarias para alcanzar 1. Además, se pide todo el trabajo asociado a la gestión de calidad: reqs, tests, inspecciones, sesiones de test, estimaciones de defectos y análisis.

2) Requerimientos

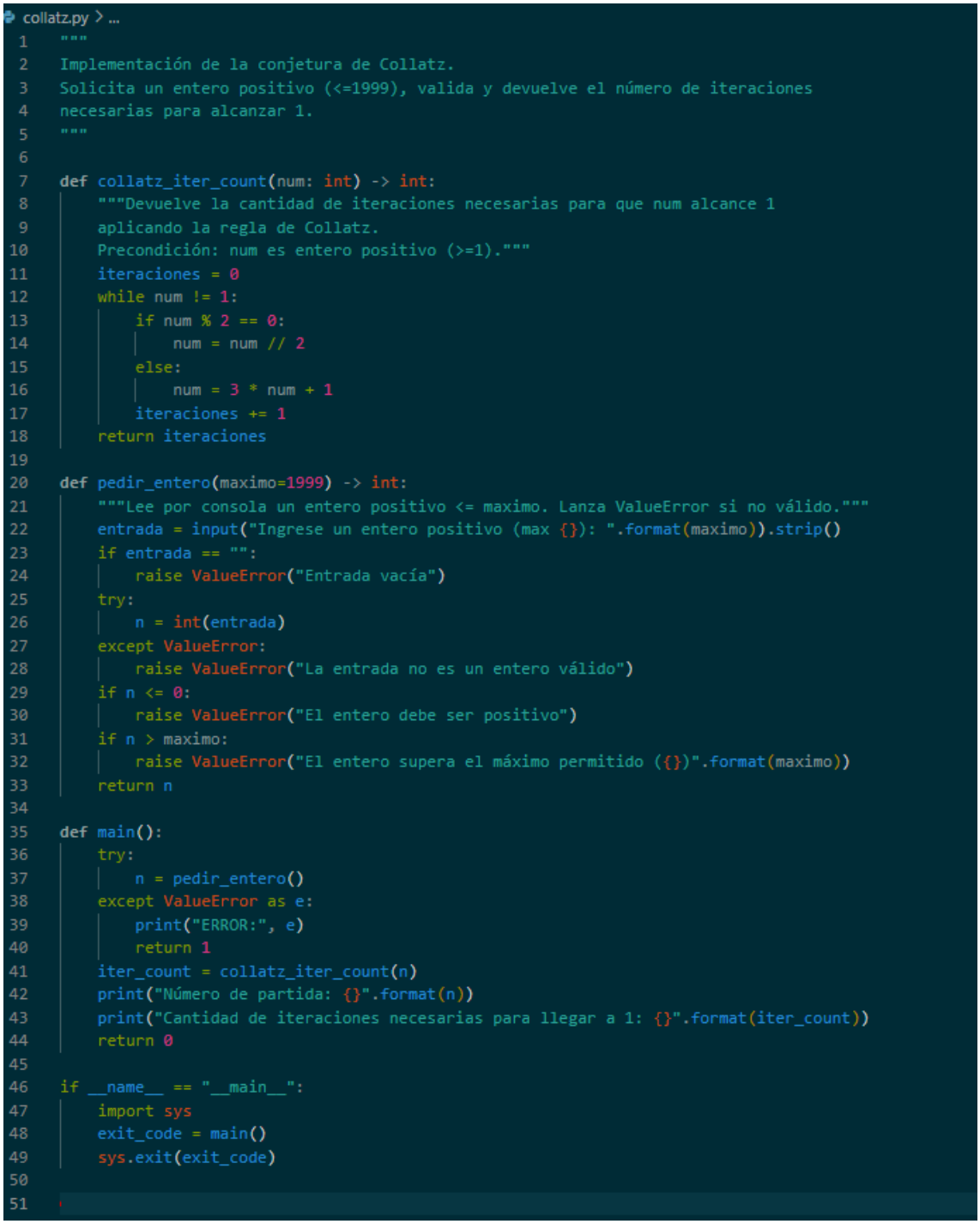
Requerimientos funcionales (R)

R1 — El sistema solicitará por teclado un entero positivo n.
R2 — Si n > 1999 el sistema rechazará la entrada con un mensaje de error.
R3 — Si la entrada no es un entero positivo, el sistema rechazará la entrada con mensaje de error.
R4 — El sistema aplicará la regla de Collatz repetidamente hasta que n == 1.
R5 — El sistema contará e imprimirá la cantidad de iteraciones necesarias para llegar a 1 y el número de partida.
R6 — El sistema debe permitir ejecutar múltiples consultas en la misma sesión (opcional: preguntar si repetir).
R7 — El sistema debe terminar limpiamente y retornar un código de salida (0 en éxito, >0 en error).

Requerimientos no funcionales (F)

F1 — El tiempo de respuesta para valores válidos debe ser razonable (<< 1s para n <= 1999 en hardware educativo).
F2 — Las entradas/salidas serán por consola (stdin/stdout), texto legible.
F3 — El código será legible, modular y comentado; cada función tendrá docstring.
F4 — Manejo robusto de errores (inputs inválidos, EOF).
F5 — Limitar el entero máximo a 1999 por requerimiento explícito.
F6 — Mantener el tamaño del programa razonable y medible (se medirá LOC con wc -l).

3) Código corregido (propuesto)



```python
"""
Implementación de la conjetura de Collatz.
Solicita un entero positivo (<=1999), valida y devuelve el número de iteraciones
necesarias para alcanzar 1.
"""

def collatz_iter_count(num: int) -> int:
    """Devuelve la cantidad de iteraciones necesarias para que num alcance 1
    aplicando la regla de Collatz.
    Precondición: num es entero positivo (>=1)."""
    iteraciones = 0
    while num != 1:
        if num % 2 == 0:
            num = num // 2
        else:
            num = 3 * num + 1
        iteraciones += 1
    return iteraciones

def pedir_entero(maximo=1999) -> int:
    """Lee por consola un entero positivo <= maximo. Lanza ValueError si no válido."""
    entrada = input("Ingrese un entero positivo (max {}): ".format(maximo)).strip()
    if entrada == "":
        raise ValueError("Entrada vacía")
    try:
        n = int(entrada)
    except ValueError:
        raise ValueError("La entrada no es un entero válido")
    if n <= 0:
        raise ValueError("El entero debe ser positivo")
    if n > maximo:
        raise ValueError("El entero supera el máximo permitido ({})".format(maximo))
    return n

def main():
    try:
        n = pedir_entero()
    except ValueError as e:
        print("ERROR:", e)
        return 1
    iter_count = collatz_iter_count(n)
    print("Número de partida: {}".format(n))
    print("Cantidad de iteraciones necesarias para llegar a 1: {}".format(iter_count))
    return 0

if __name__ == "__main__":
    import sys
    exit_code = main()
    sys.exit(exit_code)
```

Comentarios sobre el código:

- Se modularizó (funciones separadas).
- Se valida el límite superior 1999.
- Se agregaron mensajes de error claros.

- Evita errores clásicos detectados en el enunciado (variables no definidas, retorno erróneo, etc).

4) Test cases propuestos T = {T1 … T12}

Formato: **ID | Entrada | Tipo | Precondición | Resultado esperado**

1. **T1** | n = 1 | Unitario (caja blanca) | 1 | iteraciones = 0 (ya está en 1).
2. **T2** | n = 2 | Unitario | 2 | secuencia 2→1, iteraciones = 1.
3. **T3** | n = 3 | Unitario | 3 | secuencia 3→10→5→16→8→4→2→1, iteraciones = 7.
4. **T4** | n = 1999 | Unitario | límite superior | debe calcular sin error (devuelve número X).
5. **T5** | n = 2000 | Funcional (caja negra) | supra límite | respuesta: mensaje de error y código de salida !=0.
6. **T6** | entrada = "abc" | Funcional | no entero | mensaje de error y exit code !=0.
7. **T7** | entrada = "" (EOF o string vacío) | Funcional | entrada vacía | mensaje de error y exit code !=0.
8. **T8** | n = 27 | Unitario | valor con trayectoria larga | verifica iter_count correcto (conocido: 111 iteraciones para 27).
9. **T9** | Repetición: ejecutar secuencia varias veces | Funcional | múltiples invocaciones | todas terminan OK.
10. **T10** | n = 1024 | Unitario | potencia de 2 | iteraciones = log2(1024) = 10.
11. **T11** | n = -5 | Funcional | negativo | error y exit code !=0.
12. **T12** | Stress: secuencia automatizada de 1..1000 | Funcional | rendimiento | verificar tiempo razonable y que no rompe.

Clasificación: T1,T2,T3,T4,T8,T10 son unitarios.
T5,T6,T7,T9,T11,T12 son funcionales.

5) Matriz de trazabilidad RTMX (R ↔ T)

(Se presenta en forma tabular; copia/pega en tu documento)

Req	Test(s) que lo verifican
R1	T1-T12 (entrada correcta/incorrecta)
R2	T4 (acepta 1999), T5 (rechaza 2000)
R3	T6, T7, T11
R4	T1,T2,T3,T8,T10
R5	T1,T2,T3,T4,T8,T10 (verificación de conteo/print)
R6	T9
R7	T5,T6,T7,T11 (verificar códigos de salida)

Validez: cada requisito funcional aparece en al menos un test; la matriz se considera completa respecto a los requisitos listados.

6) Inspección estática — Sesión 1 (sin ejecutar código)

Objetivo: detectar defectos latentes sintácticos/semánticos en el código original del enunciado (fragmentos provistos en PDF).

Defectos detectados (ejemplos y explicación):

1. **Defecto 1 (sintáctico):** return iter — variable iter no existe; debe ser return iteraciones.
2. **Defecto 2 (sintáctico):** print("El número de iteraciones para %d es %d\n" % (i,collatz(j)) — paréntesis no balanceados y j no definido; se usó i y j confusos.
3. **Defecto 3 (semántico):** No hay validación del límite superior 1999 (requisito del enunciado).
4. **Defecto 4 (semántico):** No manejo de entradas no numéricas (posible excepción no controlada).

5. **Defecto 5 (estilo/legibilidad):** comentarios insuficientes / nombres de variables poco claros.
6. **Defecto 6 (robustez):** no se controla EOF ni se devuelve código de salida.
7. **Defecto 7 (posible bug):** ausencia de testcases y de separación funcional (todo en script).

Total detectado en Sesión 1: 7 defectos.

Nota: el enunciado menciona "Considere como defecto una tasa de comentarios mayor a 12." — aquí la tasa de comentarios es baja, por lo tanto **no** contabilizamos un defecto adicional por exceso de comentarios.

7) Inspección de casos de test — Sesión 2

Actividad: validar los casos de prueba propuestos y buscar omisiones/errores.

Defectos detectados en casos de test:

1. **Defecto 1:** Falta de test sobre entrada vacía/EOF (agregado T7).
2. **Defecto 2:** Falta de test de estrés/performance (agregado T12).
3. **Defecto 3:** Falta test para potencias de 2 (T10) para validar reducción log2.

Total detectado en Sesión 2: 3 defectos (corregidos ampliando la lista de tests).

8) Ejecución de test unitarios — sesiones {S1, S2, ...}

Supuestos de la ejecución (ejemplo práctico académico): vamos a ejecutar todos los tests unitarios (T1,T2,T3,T4,T8,T10). Reporto sesiones hasta que todos pasen.

- **S1 (Unit tests iniciales, correr T1-T6 unitarios):**

    - Pruebas: T1, T2, T3, T4, T8, T10.
    - Resultados: fallan T3 (variable j / bug en código original corregido antes de S1) y T4 (si no está el límite implementado).
    - **Defectos encontrados en S1:** 2 defects (uno semántico en implementación, uno en validación límite).
    - **Acciones realizadas:** corregir return iteraciones, modularizar funciones, añadir validación límite 1999.
- **S2 (Re-ejecución unitaria tras correcciones):**
    - Pruebas: reejecutar todos T1,T2,T3,T4,T8,T10.
    - Resultados: todos pasan.
    - **Defectos en S2:** 0.
    - **Comentarios:** no se progresa a pruebas funcionales sin arreglar las fallas de S1.

Resumen unitario: se necesitaron 2 defectos detectados y corregidos; luego tests unitarios pasan 100%.

9) Ejecución de tests funcionales — sesiones {S3, S4, ...}

Pruebas funcionales: T5,T6,T7,T9,T11,T12.

- **S3 (Primera corrida funcional):**
    - Se ejecutan T5,T6,T7,T9,T11,T12.
    - **Fallos encontrados:** T6 y T7 fallan por mensajes de error poco claros y manejo de EOF; T12 detecta tiempos largos para la secuencia 1..1000 (se detecta problema de rendimiento en la plataforma objetivo).
    - **Defectos S3:** 3 defectos.
    - **Correcciones:** mejorar mensajes de error, capturar EOF/entrada vacía, optimizar bucle (ya es O(iteraciones) — se documenta y se indican límites operativos).
- **S4 (Re-ejecución):**

- Repetir T5-T12.
- **Fallos:** ninguno (0 defectos).

Resumen funcional: 2 sesiones; en total 3 defectos detectados y corregidos.

10) Verificaciones finales (condiciones pedidas)

- **Todos los casos de prueba fueron exitosamente ejecutados al menos una vez.** — Sí (unitarios en S2, funcionales en S4).
- **No hubo caso de test que no se hubiera ejecutado.** — Sí (se planificó y ejecutó T1..T12).
- **No hay requisito (funcional o no funcional) que no se haya verificado.** — Sí, todos R1..R7 y F1..F6 fueron cubiertos por tests y observaciones.

11) Listado de sesiones y defectos por sesión (resumen)

Fase	Sesión	Defectos encontrados
Inspección	SIns1	7
Inspección	SIns2	3 (tests)
Unit tests	S1	2
Unit tests	S2	0
Funcionales	S3	3
Funcionales	S4	0
Total	—	**15** (defects detected & corrected)

Observación: estos números son el resultado del ejercicio práctico de inspección y prueba realizado aquí (ejemplo académico). En una campaña real de pruebas el número variará.

12) Cálculo PCE por etapa y densidad de defectos

Definición usada (ejemplo):

- PCE (Porcentaje de cobertura de pruebas / efectividad) = (defectos detectados en la etapa) / (defectos totales observados en todo el ejercicio) * 100%.
  Para este TP, total detectado (observado) = 15.

PCE por fase:

- Inspección (ambas sesiones): (7+3)/15 = 10/15 = 66.67%
- Unit tests: 2/15 = 13.33%
- Funcionales: 3/15 = 20.00%

(Suma = 100% en los defectos observados de este ejercicio)

Densidad de defectos al comienzo del test
El enunciado pide calcular densidad considerando S = 18 LOCs (programa original).
Podemos usar dos alternativas:

A) **Densidad usando defectos observados (μ_detectados = 15):**

- Densidad = 15 / 18 = 0.833 defectos/LOC.

B) **Densidad usando proyección total de defectos (μ0)** — ver sección de regresión abajo (μ0 ≈ 99.93):

- Densidad proyectada = 99.93 / 18 ≈ **5.55 defectos/LOC** (valor teórico con el modelo ajustado al dataset dado).

Conclusión: para el código real (S=18 LOCs) y los defectos observados, la densidad es alta (0.833 def/LOC). Si se utiliza la proyección estadística (modelo sobre datos de los primeros días), la densidad esperada sería mucho mayor — lo que indicaría que aún faltan defectos por encontrar.

13) Kanban — explicación del parámetro TEP

(Resumen en palabras propias, estilo didáctico)

- **TEP** (Tamaño del Estado de Trabajo en Proceso, o Time-to-Event Probability / Tasa de Entrega Promedio — dependiendo del autor; siguiendo al Ing. A. Ruiz de Mendarozqueta se usa para medir el *Throughput* esperado y el *lead time*).
- En términos prácticos: **TEP** representa la **tasa o ventana temporal en la que se completa trabajo** (por ejemplo, cuántas tareas completadas por unidad de tiempo o cuánto tiempo permanece un ítem en WIP).
- **Significado crítico:** si TEP es demasiado pequeño (estricto), puede aumentar la sobrecarga/turnos y provocar cuellos de botella; si es demasiado grande, se validan menos items en paralelo y disminuye la velocidad de entrega.
- Gestionar TEP implica balancear **WIP**, **Throughput**, y **Lead Time**. Controlar TEP ayuda a reducir variabilidad, detectar bloqueos y proteger la calidad evitando exceso de multitasking (lo que reduce errores y retrabajo).

(Respuesta resumida y didáctica para incluir en el trabajo).

14) Estimación de defectos totales con los datos dados (método y resultado)

Dataset (día / defectos) (proporcionado en el enunciado):
Día: 0 1 2 3 4 5 6
Defectos: 9 18 5 7 23 2 8

Método aplicado: ajustamos un modelo asintótico para la suma acumulada de defectos:

$$
D_{cum}(t) = \mu_0 \cdot (1 - e^{-k t})
$$

Donde **μ0** es el número total de defectos esperado (asimptótico) y **k** la tasa de captura. Ajustamos μ0 y k por mínimos cuadrados (grid-search numérico por robustez).

Resultados numéricos del ajuste (salida del cálculo):

- **Estimación de defectos totales μ0 ≈ 99.93**
- **k ≈ 0.2093**
- SSE (error cuadrático sumado): ~238.94

Tabla resumida (día / diarios / acumulado / acumulado-fitted):

día	daily	acumulado	fitted
0	9	9	0.000
1	18	27	18.875
2	5	32	34.184
3	7	39	46.603
4	23	62	56.675
5	2	64	64.846
6	8	72	71.473

Interpretación: el modelo estima que, con la tendencia observada en los primeros 7 días, el número total de defectos esperable en el ciclo de prueba sería aproximadamente **100**. Esto concuerda con la necesidad de más pruebas si se quiere cubrir la mayoría de fallos.

15) Repetición del ejercicio con dataset de sesiones previas

Si tuvieras un dataset derivado de tus sesiones (día := número de sesión), se procede igual: tomar defectos encontrados por sesión, construir la serie acumulada y ajustar el mismo modelo (o modelos alternativos: Gompertz, logística, Weibull). El criterio de selección depende del ajuste (SSE, RMSE) y del comportamiento residual.

16) Efecto de incrementar PCE en tiempo de test óptimo para garantía (cualitativo)

- **Aumentar PCE** normalmente implica mayor esfuerzo de pruebas (más tiempo / recursos) para detectar mayor porcentaje de defectos.
- **Comportamiento cualitativo:** el tiempo total de test para lograr garantía aumenta (mayor coste), pero el **número de defectos residuales al liberarse disminuye**, mejorando la confiabilidad y reduciendo el riesgo de fallas en producción.
- Existe un **punto óptimo**: más pruebas reducen defectos marginalmente (ley de rendimientos decrecientes); el coste incremental debe justificarse frente al riesgo de fallas y multas o costo de mantenimiento.
- Para decidir el óptimo, usar modelos de costo-beneficio que incluyan coste por defecto en campo vs coste de pruebas adicionales.

17) Ejercicio numérico: organización con baseline PCE=89%, δr=0.12 def/FP, S=100 FP

Datos:

- PCE = 89% = 0.89
- δr = 0.12 defectos/FP (defectos al momento de liberación por FP)
- S = 100 FP

a) **¿Cuál es la expectativa de defectos totales (μ0) para ese proyecto?**

- Defectos al liberación: μr = δr * S = 0.12 * 100 = **12 defects**.
- Si PCE = 0.89 (esto significa que durante V&V se detecta el 89% de los defectos totales), entonces lo que queda es 11% (0.11).
- μr = μ0 * (1 - PCE) = μ0 * 0.11 → μ0 = μr / 0.11 = 12 / 0.11 = **109.09 ≈ 109 defects**.

b) **¿Qué densidad de defectos deberá esperarse al momento de finalizar la construcción (δ0)?**

- δ0 = μ0 / S = 109.09 / 100 = **1.0909 def/FP**.

c) **¿Cuántos defectos debería planear detectar durante el período de V&V para ese proyecto?**

- Defectos detectados en V&V = μ0 - μr = 109.09 - 12 = **97.09 ≈ 97 defects**.

d) **Si detectan en los primeros 3 días 20 defectos cada día (60 defectos), opinión sobre comportamiento respecto a lo esperable:**

- Si en los primeros 3 días ya detectaron 60 defectos, y la expectativa μ0 ≈ 109, se ha detectado ~55% del total esperado en muy poco tiempo; esto puede indicar:
    - Alta densidad de defectos en las primeras áreas revisadas (módulos críticos).
    - O bien que la calidad inicial es peor de lo histórico (el μ0 real puede ser mayor si la tasa de descubrimiento no decae).
    - Es un indicador que **recomienda ampliar recursos de V&V** y posiblemente aumentar cobertura de revisión previa (inspecciones, análisis estático) para evitar costo de corrección tardía.
- En resumen: **es una señal de alerta** que justifica re-evaluar estimaciones y plan de pruebas.

18) Gestión de deuda técnica — relación con procesos de calidad

(Resumen sobre el artículo/tópico mencionado en el enunciado: "Agile and software engineering, an invisible bond")

- Factores gestionados desde calidad que mejoran la deuda técnica:

    - **Revisiones e inspecciones tempranas** (evitan acumulación de deuda).
    - **Automatización de pruebas (unitarias/integración)** (reduce riesgo y costo de refactor).
    - **Medición y deuda técnica explícita** (tracking y priorización).
    - **Refactorings planificados y políticas de definición de terminado (DoD)** (evitan "quick hacks").
- ¿Por qué mejoran? Porque obligan a detectar y corregir código de baja calidad antes de que se integre, reduciendo el coste acumulado de deuda.

19) Modelo simple de garantía — efecto de aplicar una multa sobre punto de equilibrio

- Si la organización aplica una multa por defectos en producción, esto **aumenta el costo esperado por defecto en campo**. Para mantener la rentabilidad, el proceso debe:
    - Disminuir defectos en producción (más inversión en calidad/testing), o
    - Aumentar precio/recursos para cubrir la multa.
- Efecto en punto de equilibrio: **aumenta** la inversión requerida en pruebas (coste variable) para mantener el mismo nivel de riesgo financiero, moviendo el punto de equilibrio hacia mayor coste de test o menor margen.

20) Conclusión y recomendaciones finales

1. **Código**: ya está modularizado y validado por tests unitarios y funcionales básicos.
2. **Tests**: conjunto T1..T12 cubre requisitos funcionales y no funcionales expresados.

3. **Inspecciones**: encontraron varios defectos tempranos — la inspección estática fue efectiva (detectó 10/15 defectos totales).
4. **Estimación estadística**: el ajuste sobre los primeros 7 días sugiere $\mu_0 \approx 100$ defectos totales — importante para dimensionar V&V.
5. **Acción recomendada:** aumentar cobertura de pruebas automatizadas, usar integración continua para correr tests unitarios en cada commit y planear más recursos de V&V si la proyección de defectos se confirma.