Tecnología Digital 1: Introducción a la Programación - TP1

Autores: Pardo, Ignacio
Silvestri, Juan Ignacio
Waisman, Gonzalo

Justificación

Función misma_paridad:

Para evaluar si dos números enteros n, m mayores o iguales a 0 tienen la misma paridad, la función evalúa si el resto de la división entera por 2 de n es igual al resto de la división entera por 2 de m.

El resto de la división entera por 2 sólamente puede ser 1 o 0. Es 1 cuando el número es impar ya que tal número se escribe como $2a + 1$, donde a es un número entero, qué es exactamente la definición de número impar. El resto es 0 cuando el número es par ya que ese número se puede expresar como $2a$, la definición de un número par.

Si los restos de las divisiones enteras por 2 de n y m son iguales, quiere decir que n y m son ambos impares (en el caso de 1 y 1) o son ambos pares (en el caso de 0 y 0). En ambos casos, al evaluarse en la expresión `m%2 == n%2`, la función devuelve `True`. Si esta expresión al evaluarse no es igual, significa que alguna de las dos divisiones enteras tiene resto 1 y la otra resto 0, lo que implica que no tienen la misma paridad y para tal caso la función devuelve `False`.

Función alterna_paridad:

Para evaluar si en un número alterna la paridad entre sus dígitos, la función evalúa cada par de dígitos del número n en la función `misma_paridad`, devolviendo `True` al

final si no se encuentra un par de dígitos con la misma paridad, o `False` en caso que se encuentren dígitos con la misma paridad.

La función contiene un ciclo `while` con el siguiente enunciado invariante: $1 \leq i \leq len(n\_str)$, que vale en los siguientes 4 puntos del ciclo: al comienzo de éste (A), al comienzo del bloque de código (B), al final del bloque de código (C) y al final del ciclo (D). Al comenzar el ciclo, i vale 1 tanto en A como en B. Entre B y C, se evalúa en la función `misma_paridad` los dígitos i e $i-1$ que en este caso son 1 y 0. Si esta evaluación (o la evaluación de los dígitos i e $i-1$ en general) devuelve `True`, el ciclo termina con la expresión `return False` ya que se puede asegurar que n no alterna paridad entre sus dígitos. En cambio, si `misma_paridad` devuelve `False`, los primeros dos dígitos (o los dígitos i e $i-1$ en general) de n alternan paridad, por lo que el ciclo continúa y suma 1 a la variante i. El bloque de código se repite hasta que $i = len(n\_str) - 1$, y en este punto se evalúan los dígitos i e $i-1$ que valen $len(n\_str) - 1$ y $len(n\_str) - 2$, los cuales son último y anteúltimo dígitos de n, asegurando así que se evaluaron todos los dígitos del número. Cuando se evalúa en A nuevamente, la condición del ciclo ya no es verdadera, avanzando así hasta el punto D con el invariante aún válido ya que

$$1 \leq len(n\_str) - 1 \leq len(n\_str).$$

Si el ciclo llegó hasta este punto, se puede asegurar que ningún par de dígitos de n no alterna paridad, por lo que la función devuelve `True`.

Se puede asegurar que el ciclo `while` termina correctamente debido a que en la condición del mismo se evalúa `i < len(n_str)`, donde `len(n_str)` es un entero que no se modifica a lo largo del ciclo, mientras que `i` aumenta en 1 en cada vuelta de ejecución, asegurando así que en determinado momento `i == len(n_str)`, lo cual interrumpe el ciclo ya que la condición devuelve `False`.

Función es_peculiar:

Para determinar si n es o no peculiar, se evalúan dos condiciones obligatorias:

Por una parte se realiza la comparación de igualdad entre el resto de la división entre n y 22, y 0. Además se evalúa si en el número n sus dígitos alternan su paridad tomando provecho de la función `alterna_paridad`. La función finalmente realiza una operación `and` entre ambas condiciones para determinar el valor de retorno.

La función no realiza ciclos de por sí, aquellos que se utilizan en el trasfondo están explicados en la función `alterna_paridad`.

Función n_esimo_peculiar:

La función utiliza una variable de tipo `int` como contadora de los números peculiares. Dicha variable se incrementa a medida que surgen números peculiares durante la iteración por unidad, a partir de 0 hasta el enésimo número en sí.

El enunciado invariante $0 \leq cont \leq pos, n$ es válido a lo largo de todo el ciclo. Al comenzar el ciclo, todas las variables son mayores o iguales a 0. En todas las vueltas del ciclo, se evalúa si la variable pos es un número peculiar. Si lo es (como en el primer ciclo ya que *pos = 0* y 0 es un número peculiar), la variable `cont` se incrementa en 1; al mismo tiempo, en todas las vueltas se incrementa la variable `pos` de a 22. De esta manera, se evalúan en la función `es_peculiar` a través de la variable `pos` todos los números divisibles por 22 desde 0 hasta el n-ésimo número peculiar, cuando $cont = n + 1$. En la siguiente iteración, la condición es `False`, por lo que el ciclo termina con el valor $pos\text{-}22$ como el n-ésimo número peculiar (se resta 22 ya que al final de cada ciclo se suma 22 a la variante, como intento de dejarla en un posible siguiente n peculiar.).

La correcta finalización del ciclo de la función depende únicamente de si los números peculiares forman una serie infinita de números (Ver página 5). De lo contrario, se puede afirmar que si se pasase por parámetro un n mayor al de la posición del último número en la serie de números peculiares, la función `n_esimo_peculiar` continuaría sin final.

Función cant_peculiares_entre:

Similar a la función *n_esimo_peculiar,* la función *cant_peculiares_entre* utiliza un contador que, al iterar entre los números $n, m \in \mathbb{N}_0$ ingresados, se incrementa al hallarse un número peculiar.

El enunciado invariante $0 \leq n \leq m$ se cumple a lo largo del ciclo. Se evalúan todos los números entre n y m, representados por la variable i al momento de evaluarlos en la función `es_peculiar`. La variable i se incrementa en 1 en todas las iteraciones hasta que $i > m$, donde la condición del ciclo es False y este termina. Se puede asegurar que el ciclo termina ya que m es invariable a lo largo de la ejecución del ciclo mientras que i incrementa en 1 en cada iteración lo que conlleva necesariamente a que en determinado momento, $i > m$.

Aclaraciones adicionales

CLI

En el script tp1.py, escribimos además de la función *cli*, que hicimos con lo que vimos en el ejemplo de números amigables, una función *cli_* (cli') con otros conceptos que no vimos en clase y funcionalidad adicional.

De antemano tomamos provecho de la función incluida de python *globals* que retorna un diccionario que contiene una relación key-value entre el nombre en string de una función y el objeto de dicha función en sí. Esto nos permitió no tener que encadenar ifs para "decidir" a qué función se indicó a llamar.

Además notamos que al diferenciar las funciones mediante ifs se repetía código, principalmente en el parseo a int de los parámetros que se deben dar a cada función, para reducir esto y dar menos lugar a error, hacemos un unpack de los argumentos que se pasaron por consola pasado el nombre de la función (`sys.argv[2:]`). Esto solo es posible ya que todos los parámetros de las 5 funciones son números enteros mayores o iguales a 0. De no ser así habría que haber distinguido ciertas funciones, o delegar esta conversión y admisión de tipos a cada función.

Por último, tuvimos en cuenta los errores de input posibles del usuario, cómo ingresar números menores a 0, números con coma, inputs no numéricos o funciones desconocidas.

Función n_esimo_peculiar alternativa:

Al plantear el algoritmo de la función `n_esimo_peculiar`, planteamos también una función `n_esimo_peculiar_` (enesimo peculiar'), que utiliza la función `cant_peculiares_entre`, para segmentar el ciclo en el cual contabilizar los números peculiares como se suele hacer para hallar números primos y así intentar optimizar la función. Además notamos la similitud entre las funciones `n_esimo_peculiar` y `cant_peculiares_entre`, y que de alguna forma se podría pensar una unificación entre ambas.

Infinitos números peculiares:

Nuestra suposición era que los números peculiares son infinitos, para intentar demostrarlo partimos de lo siguiente:

Los números peculiares cumplen dos condiciones:

- Son divisibles por 22.
- Alternan su paridad

Intentamos entonces explayar más las implicaciones de esta definición de número peculiar.

Los números divisibles por 22 son infinitos ya que se puede obtener como el producto entre 22 y cualquier natural, y el conjunto de los naturales es infinito.

Los números que son divisibles por 22 son los números divisibles por 11 pares (esto pasa porque 11 y 2 son coprimos).

Tenemos el cuenta la regla de divisibilidad del 11, en la que un número es divisible por 11 si la suma de sus dígitos en las posiciones pares es igual a la suma de sus dígitos en las posiciones impares, o también cuando difieren por 11, por ejemplo:

11: 1 es igual a 1

121: 1+1 es igual a 2

638: 6+8-3 = 11

Para que sean pares, volviendolos divisibles por 22, solo podemos considerar los múltiplos de 11 que contengan cantidad par de números impares en las posiciones impares, y es indiferente la paridad de la cantidad de números pares en las posiciones pares, haciendo que ambas sumas den número par y que alternen paridad. Entonces la cantidad de dígitos pares será siempre 1 mayor o igual a la cantidad de dígitos impares.

Volviendo a la idea de comparar la suma de los dígitos pares y la suma de los dígitos impares, puedo entonces siempre obtener un número peculiar mayor, sumando 2 en algún dígito par y haciendo lo mismo en algún dígito impar o 1 a dos dígitos impares, pero siempre manteniendo la igualdad o diferencia por 11 entre la suma de sus dígitos en posiciones pares y la suma de sus dígitos en las posiciones impares. O de otra forma, solo si tengo un número de cantidad de dígitos par, puedo agregarle un 2 al comienzo y sumarle 2 a uno de sus dígitos impares.

Por ejemplo, el 50º peculiar es el número 5434, para formar un nuevo peculiar a partir de este, puedo sumarle 2 a sus últimos 2 dígitos, (en este caso coincide que el próximo peculiar es el próximo divisible por 22), resultando en 5456. O bien, agregarle un 2 al

comienzo y sumarle 2 a alguno de sus dígitos impares, obteniendo el número peculiar 25454.

Parece resultar en que los números peculiares son infinitos, haciendo que la función *n_esimo_peculiar* concluya correctamente. Con esta última forma de generar nuevos peculiares, se podría pensar un algoritmo que sume 2 a dos dígitos del previo peculiar, priorizando los dígitos más últimos, a diferencia de iterar por todos los múltiplos de 22.

Sobre legibilidad y "buenas" prácticas.

En algunas ocasiones nos vimos en una bifurcación de cómo optar por escribir nuestro código. Principalmente en la función *cli_* que describimos previamente, pero también en casos más simples. El siguiente caso notamos que ocurre en 2 de las 5 funciones de la consigna y sirve a modo de un debate mayor:

```
if condicion:
   var += 1
```

Optamos por dejar comentada una alternativa "branchless" que aprovecha las operaciones entre tipos de python `int_var += condicion`, particularmente en este caso la adición entre ints y bools como se sugiere en el Python Enhancement Proposal 285:

> "Should we strive to eliminate non-Boolean operations on bools in the future, through suitable warnings, so that for example True+1 would eventually (in Python 3000) be illegal?
> => No.
> There's a small but vocal minority that would prefer to see "textbook" bools that don't support arithmetic operations at all, but most reviewers agree with me that bools should always allow arithmetic operations."

Un ejemplo de esto sucede en la función *cant_peculiares_entre* donde el siguiente ciclo se podría reducir evitando el if.

```
while i <= m:
      if es_peculiar(i):
            cant += 1
      i += 1
```

```
while i <= m:
      cant += es_peculiar(i)
      i += 1
```

Conclusión

Al verse en dos posiciones conjuntas frente al mismo problema, siendo una la declaración y especificación de las funciones a ser abstraídas e implementadas del otro lado a la hora de escribir la CLI, estas prácticas se vieron bajo juicio muchas veces durante la realización de este trabajo.

Frente al Zen de Python, *“Flat is better than nested”* entró en conflicto con *“Sparse is better than dense”* a la hora de tomar muchas de las decisiones, aunque muchas de ellas se terminan viendo opacadas por la subjetividad de *“Beautiful is better than ugly”.*