Victoria Stefania Schenone Fernández (Steffy)

Francisco Serravalle

Santiago Chiarini Podesta

Tecnología Digital 1: Introducción a la Programación

Trabajo Práctico 1

Justificación de los casos de tests elegidos: 32 tests totales.

Para todas las funciones, **Test_EjemplosEnunciado**:
Se hizo la prueba para justificar lo solicitado en las pautas del TP.

Función Tests_decimal_a_binario:

MásDeUnDígito(self): dos números con más de una cifra.
NumerosRepetidos(self): secuencia de los mismos números.
NumerosPares(self): grupos definidos diferentes números.
NumerosImpares(self): idem punto anterior.
PotenciaDeDos(self): cálculos de números más complejos. Caso borde, no habitual, impacto significativo en la experiencia del usuario al ver el programa.

Función Tests_es_binario_balanceado:

SoloUnDígito(self): solamente una cifra difiere variación algunos son balanceados y otros no.
SoloUnos(self): número 1 repetidos.
CasiCompletos(self): lo que "casi pasaría", tener más 1's que 0's o más 0's que 1's.
NumerosPares(self): grupos definidos diferentes números.
NumerosImpares(self): idem punto anterior.

Función Tests_cantidad_binarios_balanceados_entre:

Varios(self): números diversos al azar para incluir integridad con varias opciones.
NumerosFactoriales(self): producto de los números enteros positivos desde 1 hasta n o m, caso especial y no tan habitual, visto en la materia.
NumerosCompuestos(self): números naturales (cumple con el Requiere de la función) no primos, diferentes tipos de números divisibles.
NumerosPrimos(self): número natural, tiene divisor el 1 y él mismo.
IntervaloMínimo(self): si n y m son iguales (cumple Requiere n<=m), en un rango muy pequeño y nos devolvería la cantidad de un balanceado solo.

Función Tests_siguiente_binario_balanceado:

MuchosDígitos(self): cifra más alta, escala más elevada de números.
MismosDígitosRepetidos(self): número con todos sus dígitos iguales.
NumerosImpares(self): grupos definidos diferentes números
Primeros10(self): primeros 10 números naturales, no siempre el siguiente binario balanceado es el número sucesivo al parámetro n, saltea hasta la cantidad idéntica de 0's y 1's.

Función Tests_anterior_binario_balanceado:

Cercanos_y_Alejados(self): n más bajo cerca y n más alto menor alejado.
Primeros8(self): primeros 8 números naturales, desde el 3, y difiere el resultado en 10.

Función Tests_binario_balanceado_mas_cercano:

CuandoEsBinarioBalanceado(self): 1er condicional, n misma cantidad 0's y 1's, res es = n.
BinariosPrevios(self): 3er condicional, si n-previo < sucesivo-n = número anterior.
Binarios_Sucesivos(self): 2do condicional, si n-previo > sucesivo-n = número siguiente.
SucesivoAnteriorIguales(self): 2do condicional n-previo sea = a sucesivo-n = num siguiente

Demostración de Correctitud y Terminación:

Función es_binario_balanceado:

Terminación:
Antes de empezar el ciclo, la variable i empieza valiendo 0. (línea 21)
En cada ejecución del cuerpo del ciclo, i se incrementa en 1. (línea 33)
Por la cláusula Requiere sabemos que n debe pertenecer a los números naturales, es decir que n debe ser > 0. Y, además, en el cuerpo del ciclo no se modifica el valor de n.
Entonces es inevitable que en algún momento i llegue a valer la longitud de conversión_n.
En ese momento, la condición i<len(conversión_n) será False, por lo que el ciclo terminará.

Correctitud: Primero identifiquemos qué cosas podemos afirmar sobre las variables del programa (i, contador_ceros y contador_unos) que sean verdaderas en los puntos (A), (B), (C) y (D), indicados en el código junto a la función es_binario_balanceado. Cuando n = 2 veamos:

i	contador_ceros	contador_unos
0	0	0
1	0	1
2	1	1

Viendo esta tabla, podemos afirmar 2 cosas que son verdaderas en cada iteración:

La variable i se encuentra entre 0 y el len de conversión_n (recordemos que len de conversión_n es 2) pero, ¿Por qué? Nosotros sabemos que i empieza valiendo 0 antes de entrar al ciclo, pero tal como dijimos en la terminación i va llegar a valer el len de conversión_n, es decir **0<=i<=len(conversión_n)**

La variable contador_ceros vale la cantidad de ceros, si en las primeras i posiciones sin incluir de conversión_n hay un '0'. Y la variable contador_unos vale la cantidad de unos, si en las primeras i posiciones sin incluir de conversión_n hay un '1'.

Notar que proponemos este **Invariante** para cualquier valor de n.
Sigamos ahora el siguiente razonamiento que nos lleva desde el principio hasta el final de la ejecución de es_binario_balanceado:

#(A) Nuestro invariante es verdadero en el punto (A), donde i vale 0 **(esto me verifica 0<=i<=len(conversión_n))** y las variables contador_ceros y contador_unos valen 0.

Analicemos si contador_ceros = 0 y contador_unos = 0, me verifica la segunda parte del invariante, contador_ceros vale la cantidad de ceros, si en las primeras i posiciones sin incluir de conversión_n hay un 'o' y contador_unos vale la cantidad de unos, si en las primeras i posiciones sin incluir de conversión_n hay un '1'. Decir 'hasta las primeras i posiciones sin incluir', es lo mismo que decir 'hasta la pos i-1', entonces nos queda:

- **contador_ceros vale la cantidad de ceros, si hasta la pos i-1, hay un 'o' y contador_unos vale la cantidad de unos, si hasta la pos i-1, hay un '1".**

Pero, antes de entrar al ciclo, i vale 0, entonces nos queda:

- **contador_ceros vale la cantidad de ceros, si hasta la pos i-1, hay un 'o' y contador_unos vale la cantidad de unos, si hasta la pos i-1, hay un '1".**

Pero ¿Qué posición podemos comparar hasta la pos i-1? → Ninguna porque i vale 0, entonces contador_ceros = 0 y contador_unos = 0, siendo cero ambas y sin poder comparar ninguna posición de conversión_n y eso es verificado por el contador_ceros = 0 y contador_unos = 0 del principio.

#(B) Notemos que la primera vez que llegamos a (B), el invariante vale por lo mismo del punto (A) (i=0, contador_ceros = 0 y contador_unos = 0) porque esto sucede justo después de que la guarda del while resulte True, entonces i, contador_ceros y contador_unos todavía no fueron afectados.

Luego de ejecutar todo el cuerpo llegamos a #(C) i = 1, contador_ceros se le suma un uno, porque vemos que si se compara la posición de la variable conversión_n con un 'o' y supongamos que esto fuera cierto es porque coincide con un 'o'. Lo mismo sucede para contador_unos, si se compara la pos de conversión_n con un '1' y eso resulta True es porque encajan y a contador_unos se le suma 1. Por consiguiente, i se incrementa en 1 y por eso se cumple: **0<=i<=len(conversión_n) y que contador_ceros vale la cantidad de ceros, si hasta la pos i-1, hay un 'o' y contador_unos vale la cantidad de unos, si hasta la pos i-1, hay un '1'.**

(Nuestro invariante siempre va ser verdadero al terminar el cuerpo del ciclo, cabe destacar que entre las 2 instrucciones cuando se ejecute la primera se va romper el invariante, pero luego en #C se reincorpora).

Esto se mantiene ciclo a ciclo, hasta que **i = len(conversión_n)**, es decir mientras que la guarda del while resulte True, mi ciclo "es como si fuera un ping pong" a lo largo de sucesivas iteraciones, tendremos que el invariante propuesto vale en (B), vale en (C), vale en (B), vale en (C), vale en (B), vale en (C), etc.

Cuando i llega a valer el len(conversión_n), ahí deja de valer la guarda del while, por lo tanto, la condición resulta Falsa, salimos del ciclo y llegamos al punto #D.

#(D) Dado que al evaluar la condición i<len(conversión_n) da False, no modifica el valor de las variables, por ende, el invariante sigue siendo verdadero en el punto (D) y es fácil ver que i = len(conversión_n), hace valer la primera parte del invariante: **0<=i<=len(conversión_n)**

Ahora analicemos la segunda parte:

contador_ceros vale la cantidad de ceros, si hasta la pos i-1, hay un 'o' y contador_unos vale la cantidad de unos, si hasta la pos i-1, hay un '1".

Pero dijimos que i = len(conversión_n)

Entonces:

contador_ceros vale la cantidad de ceros, hasta el len(conversión_n)-1, si hay un 'o' y contador_unos vale la cantidad de unos, hasta el len(conversión_n)-1, si hay un '1".

¿Y que significa esto? → Tengamos en cuenta que len(conversión_n)-1 es la última posición de conversión_n, entonces ¿si ya llega a la última posición es porque ya recorrió las primeras? Sí.

Entonces:

contador_ceros vale la cantidad de ceros y contador_unos vale la cantidad de unos para cada carácter de conversión_n, si hay un 'o' o si hay un '1'.

#comentario: Si sucede lo explicado y además al terminar al ciclo se cumple que contador_ceros y contador_unos tienen la misma cantidad de 0's o de 1´s (cada uno por separado y los comparas entre sí), la variable vr vale True. Si no sucede esto la variable vr vale False es decir que dejarían de ser iguales.
Y vemos que esta parte del invariante cumple con "mi devuelve" de la especificación de la función es_binario_balanceado. Así queda finalizada la demostración de la correctitud. □

Función siguiente_binario_balanceado:

Terminación:

Antes de empezar el ciclo, variable binario_balanceado_posterior empieza valiendo n+1 (l.63)
En cada ejecución del cuerpo del ciclo, binario_balanceado_posterior se incrementa en 1 (l.67).
Por la cláusula Requiere sabemos que n debe ser mayor a 0, es decir debe pertenecer a los números naturales. Y además en el cuerpo del ciclo no se modifica el valor de n.
Entonces es inevitable que en algún momento es_binario_balanceado(binario_balanceado_posterior) llegue a ser balanceado, sucede esto porque hay infinitos números balanceados, si no hubiese infinitos no podrían existir próximos números balanceados.
En ese momento, la condición not es_binario_balanceado(binario_balanceado_posterior) será False, por lo que el ciclo terminará porque encontrará el binario sucesivo.

Correctitud: Primero identifiquemos qué cosas podemos afirmar sobre las variables del programa (binario_balanceado_posterior) que sean verdaderas en los puntos (A), (B), (C) y (D), indicados en el código junto a la función siguiente_binario_balanceado, cuando n = 2 veamos:

Binario_balanceado_posterior
3
4
5
6
7
8
9

Viendo esta tabla, podemos afirmar 2 cosas que son verdaderas en cada iteración:

La variable binario_balanceado_posterior se encuentra entre n+1 (recordemos que n=2, pero antes de entrar al ciclo se le suma uno) y el binario_balanceado_posterior inclusive, pero ¿Por qué? Nosotros sabemos que binario_balanceado_posterior empieza valiendo n+1 antes de ingresar al bucle, pero tal como dije en la terminación binario_balanceado_posterior va llegar a valer el siguiente binario balanceado, cuando se encuentre, entonces:
n+1<=binario_balanceado_posterior<=binario_balanceado_posterior

Para todo j (es decir número) entre n+1 y binario_balanceado_posterior incluido en el extremo izquierdo (todos esos j no son balanceados), pero no en el extremo derecho.

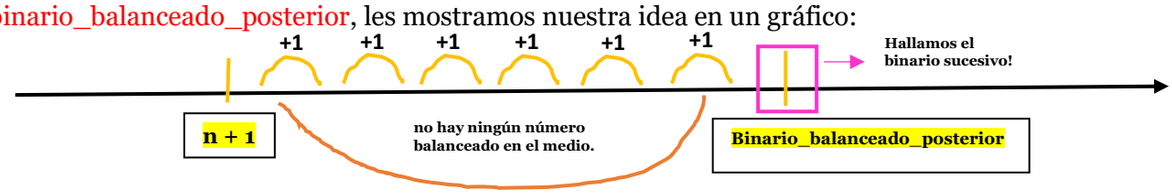
Notar que proponemos este **Invariante** para cualquier valor de n.
Sigamos ahora el siguiente razonamiento que nos lleva desde el principio hasta el final de la ejecución de binario_balanceado_posterior:

#(A) Nuestro invariante es verdadero en el punto (A), donde binario_balanceado_posterior vale n+1 (esto me verifica n+1<=binario_balanceado_posterior<=binario_balanceado_posterior) y para verificar la segunda parte del invariante, supongamos que:

Podría llegar a ser cierto a que no haya binarios balanceados sucesivos, al momento de sumarle a binario_balanceado_posterior n+1 (en el caso de no entrar al ciclo, sin pasar por los puntos #(B) y #(C) y llegar directamente a #(D), significa que hallamos el próximo binario balanceado).

#(B) Notemos que la primera vez que llegamos a (B), el invariante se rompe porque como entra al ciclo, significa que binario_balancedo_posterior no es binario balanceado, por lo tanto, la guarda del while = not False, quiere decir True e ingresamos. Mientras que la variable, binario_balanceado_posterior = n + 1 sigue valiendo lo mismo, porque todavía no fue alterada.

Luego de ejecutar todo el cuerpo del ciclo, llegamos a #C: (se reincorpora el invariante) binario_balanceado_posterior = binario_balanceado_posterior + 1, se incrementa por primera vez, pero esta instrucción no te brinda mucha información, porque todavía no sabemos si es el siguiente binario balanceado o no. Pensemos que no sea el binario sucesivo, va ir preguntándose con la guarda iteración a iteración si binario_balanceado_posterior ¿es balanceado o no? Y vas a ir sumando de a 1, hasta hallarlo porque los números son infinitos, por lo tanto, todos los números que no sean son aquellos que están en el medio entre n+1 y binario_balanceado_posterior, les mostramos nuestra idea en un gráfico:



Esto se mantiene ciclo a ciclo, hasta hallar el binario_balanceado_posterior, es decir mientras que la guarda del while resulte True.

Cuando binario_balanceado_posterior llegue a valer el siguiente binario balanceado, ahí deja de valer la guarda del while, por lo tanto, la condición resulta Falsa, salimos del ciclo y llegamos al punto #D.

#(D) Al evaluar la condición not es_binario_balanceado(binario_balanceado_posterior) es False y eso no modifica el valor de las variables, por ende, el invariante sigue siendo verdadero en (D) y es fácil ver que binario_balanceado_posterior = binario_balanceado_posterior.
Hace valer la primera parte del invariante:
n+1<=binario_balanceado_posterior<=binario_balanceado_posterior

Ahora analicemos la segunda parte:

Para todo j (es decir número) entre n+1 y binario_balanceado_posterior incluido en el extremo izquierdo (todos esos j no son balanceados), pero no en el extremo derecho.
Pero dijimos que binario_balanceado_posterior = binario_balanceado_posterior

Entonces:
Para todo j (es decir un número) entre n+1 hasta que binario_balanceado_posterior sea el binario_balanceado_posterior que se quiera encontrar hacia la dirección de +infinito.
¿Y qué significa esto? → binario_balanceado_posterior = binario_balanceado_posterior es dicho número sucesivo que queríamos obtener entonces ¿si va sumando de a 1 en 1 y todos esos no son los balanceados que le siguen? Este, sí lo es.

Entonces:

Para j tal que n+1<=binario_balanceado_posterior<=binario_balanceado_posterior, binario_balanceado_posterior = binario_balanceado_posterior si j es el n sucesivo y si j no lo es binario_balanceado_posterior = binario_balanceado_posterior +1 (hasta encontrarlo)

Y vemos que esta parte del invariante cumple con “mi devuelve” de la especificación de la función es_binario_balanceado. Así queda finalizada la demostración de la correctitud. □