Correctitud

Ejercicio 1

Para verificar la correctitud del algoritmo que calcula la suma de los primeros n números naturales, comenzaremos definiendo la invariante. Esta invariante establece que, en cada iteración del bucle `while`, donde $k = i$, el valor de `resultado` es igual a la suma de todos los números desde 1 hasta k, es decir, $\sum_{i=1}^{k} i$.

Inicialización

Consideramos el caso base antes de entrar en el bucle `while`. En este estado, `resultado` es igual a 0, lo que indica que no se ha procesado ninguna suma. Según la definición de la invariante, la suma de no considerar ningún elemento es 0, lo que respeta la invariante.

Mantenimiento

Asumimos que en la iteración $i = k$, la invariante se cumple y, por lo tanto, `resultado` es igual a $\sum_{i=1}^{k} i$.

Ahora probaremos que la invariante se mantiene para $i = k + 1$. Sabemos que, en la iteración k, `resultado` contiene el valor $\sum_{i=1}^{k} i$. Al procesar la siguiente operación, `resultado` se actualizará de la siguiente manera:

$$\text{resultado} = \text{resultado} + (k + 1)$$

Esto se traduce a:

$$\text{resultado} = \sum_{i=1}^{k} i + (k + 1) = \sum_{i=1}^{k+1} i$$

Por lo tanto, hemos demostrado que si la invariante es cierta para k, también lo es para $k + 1$.

Terminación

El bucle `while` se ejecuta hasta que $k = n$. Cuando esto ocurre, se habrá sumado cada valor desde 1 hasta n. En este punto, la invariante garantiza que `resultado` es igual a $\sum_{i=1}^{n} i$. Dado que esta es precisamente la suma de los primeros n números naturales, el algoritmo termina correctamente con el valor deseado. Esto concluye la prueba de que el algoritmo es correcto.

Ejercicio 2

Queremos verificar la correctitud de la función que calcula el factorial de un número entero positivo. Para ello, utilizaremos el principio de inducción débil.

Caso Base

Consideremos el caso base cuando $n = 0$. Según la definición del factorial, factorial$(0) = 1$, lo cual coincide con el valor esperado, ya que $0! = 1$.

Hipótesis Inductiva

Asumimos que la función es correcta para $n = k$, es decir, que factorial$(k) = k!$.

Paso Inductivo

Ahora demostraremos que la función también es correcta para $n = k + 1$. Según la definición de la función, se tiene:

$$\text{factorial}(k + 1) = (k + 1) \cdot \text{factorial}(k)$$

Por la hipótesis inductiva, sabemos que factorial$(k) = k!$. Sustituyendo en la ecuación anterior, obtenemos:

$$\text{factorial}(k + 1) = (k + 1) \cdot k!$$

Esto es precisamente la definición del factorial:

$$(k + 1) \cdot k! = (k + 1)!$$

Por lo tanto, hemos demostrado que factorial$(k + 1) = (k + 1)!$.

Hemos verificado el caso base y completado el paso inductivo, lo que confirma que la función que calcula el factorial de un número entero positivo n es correcta para todos los $n \geq 0$.

Ejercicio 3

Definimos la secuencia de Fibonacci como sigue:

0, 1, 1, 2, 3, 5, 8, ...

Para demostrar que el código es correcto, utilizaremos inducción fuerte.

Casos base:

1. Si $n = 0$, entonces fibonacci$(n) = 0$ (lo cual es correcto).
2. Si $n = 1$, entonces fibonacci$(n) = 1$ (lo cual es correcto).

Hipótesis inductiva:

Asumimos que para todas las $n \leq k$, la función fibonacci es correcta, es decir, fibonacci(n) es igual al n-ésimo número de la secuencia.

Paso inductivo:

Demostraremos que la afirmación es válida para $n = k + 1$. Dado que n no es un caso base, para calcular fibonacci(n) usaremos la relación:

$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

Reemplazando con $k + 1$, obtenemos:

$$\text{fibonacci}(k+1) = \text{fibonacci}(k) + \text{fibonacci}(k-1)$$

Sabemos que fibonacci(k) y fibonacci($k - 1$) son correctas por la hipótesis inductiva. Por lo tanto, podemos concluir que fibonacci(n) es correcto.

Ejercicio 4

Para demostrar la correctitud del algoritmo Bubble Sort, analizaremos los bucles interno y externo utilizando invariantes de ciclo.

Invariante de ciclo para el bucle interno

Invariante: Al final de cada iteración del bucle interno (índice j), el mayor elemento entre $arr[0..j]$ está en la posición j.

Inicialización: En la iteración $j = 0$, $arr[0]$ es el número mayor en el intervalo $arr[0...0]$. Por ende, se cumple el invariante.

Mantenimiento: Supongamos que al final de la k-ésima iteración del bucle interno, el mayor elemento entre $arr[0..k]$ está en la posición k. En la $ (k + 1) $-ésima iteración del bucle interno, el algoritmo compara $arr[j]$ y $arr[j + 1]$. Si $arr[j] > arr[j + 1]$, se realiza un intercambio. Esto garantiza que al final de esta iteración, el mayor elemento entre $arr[0..k + 1]$ se ubica en la posición $k + 1$. Por lo tanto, la invariante se mantiene.

Terminación: El bucle interno se ejecuta hasta que j alcanza su valor máximo, lo que corresponde al último elemento no ordenado de la parte no procesada de la lista. En este punto, el mayor elemento en el rango $arr[0..j]$ estará en su posición correcta j, lo que asegura que el bucle ha ordenado correctamente todos los elementos dentro de ese rango.

Invariante de ciclo para el bucle externo

Invariante: Al final de cada iteración del bucle externo (índice i), los $i+1$ elementos más grandes de la lista están en las últimas posiciones.

Inicialización: Antes de la primera iteración ($i = 0$), no se han realizado intercambios y no hay elementos ordenados. La invariante se cumple trivialmente porque no hay elementos en la parte ordenada de la lista.

Mantenimiento: Supongamos que al final de la k-ésima iteración del bucle externo, los k elementos más grandes están en sus posiciones correctas al final de la lista. En la $(k+1)$-ésima iteración, el bucle interno se ejecuta y compara pares de elementos adyacentes. El elemento más grande entre los no ordenados se mueve a la última posición de la parte no ordenada de la lista. Por lo tanto, al final de la $(k+1)$-ésima iteración, $k+1$ elementos más grandes estarán en la posición $n-k-1$. Esto significa que la invariante se mantiene.

Terminación: El bucle externo termina cuando i alcanza $n-1$, es decir, cuando todos los elementos han sido ubicados en sus posiciones correctas. En este punto, la lista completa estará ordenada, cumpliéndose la invariante final de que los n elementos más grandes están en sus posiciones finales, de mayor a menor en las últimas posiciones de la lista.

Ejercicio 5

La invariante será que, en la iteración i, el número x no es divisible por ningún número en el rango de 2 a i. Esto implica que para cualquier d en $[2, i]$, $x \bmod d \neq 0$ o x es compuesto (donde $i < n$).

Inicialización:

Antes de la primera iteración, el intervalo de números es vacío, por lo que x no es divisible por ninguno de esos números. Esto cumple la invariante porque no hay divisores en el rango $[2, 1]$ que puedan dividir a x.

Mantenimiento:

Supongamos que en la k-ésima iteración la invariante se cumple, es decir, x no es divisible por ningún número en el intervalo $[2, k]$ o que x es compuesto. Demostraremos que la invariante se mantiene para la iteración $(k+1)$.

- Si x es divisible por algún número en $[2, k]$, entonces ya se ha detectado que x es compuesto, y el algoritmo habría retornado `false` en una iteración anterior.
- Si x no es divisible por ningún número en $[2, k]$, entonces llegamos a la iteración $k+1$, donde se presentan dos posibilidades:
    1. **Si x es divisible por $k+1$** (donde $k+1 < x$): En este caso, se concluye que x es compuesto, y el algoritmo retornará `false`.

2. **Si x no es divisible por ningún número en** $[2, k+1]$: Esto significa que para cada número d en este intervalo, también se cumple que $x \mod d \neq 0$, manteniéndose la invariante.

Terminación:

El bucle finaliza cuando se ha verificado que x no es divisible por ningún número en el rango $[2, \sqrt{x}]$. En este punto, si no se ha encontrado ningún divisor, se concluye que x es primo y el algoritmo retorna `true`. De esta manera, la invariante se cumple hasta la terminación, garantizando que el algoritmo ha evaluado correctamente si x es primo o compuesto.

Demostración de la necesidad de revisar solo hasta $\sqrt{x}$:

Es importante notar que si x tiene un divisor distinto de 1 y x, al menos uno de esos divisores debe ser menor o igual a $\sqrt{x}$. Esto se debe a que, si ambos divisores son mayores que $\sqrt{x}$, su producto sería mayor que x, lo cual es una contradicción. Por lo tanto, basta con verificar la divisibilidad de x solo con los números en el rango de 2 a $\sqrt{x}$. Si x no es divisible por ningún número en este rango, se puede concluir que x es primo.

Con esto, la invariante se mantiene, y podemos concluir que el algoritmo verifica correctamente si un número entero x es primo.

Ejercicio 6

Demostraremos la correctitud del algoritmo utilizando inducción fuerte.

Casos base:

Para el caso de $n = 0$, la función `diceCombinations(0)` es correcta, ya que hay una única forma de obtener la suma, que consiste en no lanzar ningún dado.

Hipótesis inductiva:

Supongamos que `diceCombinations(n)` es correcta para todos los n menores que k, es decir, $n \leq k$.

Paso inductivo:

Consideremos $n = k + 1$:

1. **Caso 1:** $k + 1 < 6$
   En esta situación, los posibles valores que se pueden lanzar con el dado son $\{1, 2, \ldots, k+1\}$. Si lanzamos un dado y obtenemos un valor x perteneciente a este conjunto, esto es equivalente a calcular `diceCombinations(k + 1 - x)`. Por lo tanto, para todo x en $\{1, 2, \ldots, k+1\}$, se tiene que:

$$\text{diceCombinations}(k+1) = \sum_{x=1}^{k+1} \text{diceCombinations}(k+1-x)$$

2. **Caso 2:** $k+1 \geq 6$
   En este caso, los valores posibles que se pueden lanzar son $\{1, 2, \ldots, 6\}$. Similar al caso anterior, al lanzar un dado y obtener un valor x perteneciente a este conjunto, podemos expresar esto como `diceCombinations(k + 1 - x)`. Así, para todo x en $\{1, 2, \ldots, 6\}$:

$$\text{diceCombinations}(k+1) = \sum_{x=1}^{6} \text{diceCombinations}(k+1-x)$$

En ambos casos, hemos mostrado que `diceCombinations(k + 1)` puede ser expresado como una suma de combinaciones de las formas de alcanzar las sumas correspondientes. Además, podemos escribir `diceCombinations(k + 1)` en función de los casos correctos según la hipótesis inductiva. Esto confirma que la función se comporta de manera correcta según la hipótesis inductiva.

Por lo tanto, mediante inducción fuerte, podemos concluir que el algoritmo es correcto para cualquier n no negativo.