Taller 1: Recursion

Jhorman Gomez[1], Esteban Gomez[2]

2326867[1], 2330197[2]

Universidad del Valle

Facultad de Ingeniería

Escuela de Ingeniería de Sistemas y Computación

Santiago de Cali, Septiembre de 2024

1 Informe de Procesos

1.1 Máximo Común Divisor

1.1.1 Utilizando el Teorema Fundamental de la Aritmética

La función denominada "mcdTFA" genera un proceso recursivo lineal. Esto se debe a que la función realiza exactamente $\underline{n}$ llamadas recursivas, siendo $\underline{n}$ la longitud de las listas proporcionadas como argumentos. Cada llamada recursiva procesa un elemento de las listas y luego pasa al siguiente, sin realizar llamadas adicionales dentro de cada recursión.

```
def mcdTFA(ln: List[Int], lm: List[Int], primes: List[Int]): Int = {
    def obtainLowerExponent(exponentN: Int, exponentM: Int): Int = {
      if (exponentN <= exponentM) exponentN else exponentM;
    }
    def exponentiation(base: Int, exponent: Int): Int = {
      if (exponent == 0) 1 else{
        base*exponentiation(base, exponent-1);
      }
    }
    if (primes.isEmpty) 1
    else {
      exponentiation(primes.head, obtainLowerExponent(ln.head, lm.head)) *
      mcdTFA(ln.tail, lm.tail, primes.tail);
    }
}
```

Fragmento de código 1. Función mcdTFA.

La función recibe como argumentos tres listas: *ln*, *lm* y *primes*. La lista *ln* contiene los exponentes de los factores primos del número $\underline{n}$, mientras que *lm* contiene los exponentes correspondientes al número $\underline{m}$. La lista *primes* incluye los factores primos que descomponen ambos números.

El funcionamiento de la función es el siguiente:

1. Verificación de la lista de primos: La función comienza verificando si la lista *primes* está vacía. Si es así, retorna 1, ya que no hay más factores primos que procesar.

2. Obtención del menor exponente: Si la lista *primes* no está vacía, la función llama a *obtainLowerExponent*, que evalúa los valores en la cabeza de las listas *ln* y *lm* para determinar el menor exponente entre ambos.

3. Exponenciación: El menor exponente obtenido se usa para elevar el número primo correspondiente al valor de la cabeza de la lista *primes*.

4. Llamada recursiva: La función se llama a sí misma (*mcdTFA*) con las colas de las listas *ln*, *lm* y *primes*, es decir, sin los valores en la cabeza de cada lista.

5. Este proceso se repite hasta que la lista *primes* esté vacía. En ese momento, la función habrá calculado el máximo común divisor (MCD) entre los números $\underline{n}$ y $\underline{m}$.

1.1.2 Utilizando el Algorimto de la División

La función denominada "mcdEBez" genera un proceso recursivo lineal. La función genera $\underline{c}$ llamados recursivos, siendo $\underline{c}$ el número de ciclos que le toma al algoritmo encontrar el MCD entre los números $\underline{n}$ y $\underline{m}$. Cada llamada recursiva procesa un par de valores y luego pasa al siguiente par, sin realizar llamadas adicionales dentro de cada recursión.

```
def mcdEBez(n: Int, m: Int): (Int, Int, Int) = {
    if (m == 0) (n, 1, 0)
    else {
        val (d, x1, y1) = mcdEBez(m, n%m);
        val x = y1;
        val y = x1 - n/m * y1;
        (d, x, y);
    }
}
```

Fragmento de código 2. Función mcdEBez.

La función recibe como argumentos dos números $\underline{n}$ y $\underline{m}$ de tal forma que $n \geq m$. El funcionamiento de la función es el siguiente:

1. La función verifica si $m == 0$, de ser así se concluye que estamos ante el caso base donde $\underline{n}$ es el mcd y los coeficientes de bezout para $\underline{x}$ e $\underline{y}$ son 1 y 0 respectivamente, de no ser así se llama recursivamente otra vez a la función *mcdEBez* con el argumento $\underline{n}$ en la nueva función ahora siendo $\underline{m}$ y el argumento $\underline{m}$ ahora siendo el residuo entre los n y m anteriores, es decir, $n\%m$, este proceso se repite hasta que se llega al caso base descrito anteriormente.

2. Una vez llegados al caso base, el programa comienza a regresarse hacia atrás y finalmente se tienen valores para los $\underline{x_1}$ e $\underline{y_1}$ iniciales, gracias a estos coeficientes iniciales se va calculando los nuevos coeficientes hasta finalmente regresar al primer llamado de la función donde ahora se nos entrega una tupla con los valores d, x, y correspondientes al mcd y a los coeficientes de bezout.

1.2 Números de Fibonacci

1.2.1 Fibonacci Árbol

La función denominada "fibonacciA" implementa un proceso recursivo para calcular el número de Fibonacci de un índice dado. El proceso se basa en la definición estándar de la serie de Fibonacci, donde cada número en la secuencia es la suma de los dos anteriores. Esta implementación, sin embargo, tiene un

rendimiento limitado debido a la naturaleza de la recursión no optimizada, lo que conduce a muchas llamadas repetidas.

```
def fibonacciA(n: Int): Int = {
    if (n == 0) 1
    else if (n == 1) 1
    else fibonacciA(n - 1) + fibonacciA(n - 2)
}
```

Fragmento de código 2. Función fibonacciA.

La función recibe como argumento un entero $\underline{n}$, que representa el índice en la secuencia de Fibonacci que se desea calcular. La función sigue el siguiente procedimiento:

1. Condición base 1: Si $\underline{n == 0}$, la función retorna 1. Esto se ajusta a una versión modificada de la serie de Fibonacci que comienza con 1 en lugar de 0.

2. Condición base 2: Si $\underline{n == 1}$, también retorna 1. Esto asegura que la secuencia comience correctamente, es decir, con 1 en el índice 1.

3. Llamadas recursivas: Si $\underline{n > 1}$, la función realiza dos llamadas recursivas, calculando el número Fibonacci para $\underline{n - 1}$ y para $\underline{n - 2}$, y luego sumando los resultados. Este paso sigue la regla fundamental de la serie de Fibonacci:
$$F(n) = F(n - 1) + F(n - 2)$$

1.2.2 Fibonacci Iterativo

La función denominada `fibonacciI` implementa un proceso iterativo para calcular los números en la secuencia de Fibonacci. A diferencia de un enfoque recursivo, la función no realiza múltiples llamadas recursivas, sino que utiliza un bucle para calcular el término deseado.

```
def fibonacciI(n: Int): Int = {
  def it(n: Int, a: Int, b: Int): Int = {
    if (n == 0) a
    else if (n == 1) b
    else it(n - 1, b, a + b)
  }
  it(n, 1, 1)
}
```

Fragmento de código 2. Función fibonacciI.

La función recibe como argumento un número entero $\underline{n}$ que indica el índice en la secuencia de Fibonacci. El funcionamiento de la función es el siguiente:

1. Definición de la función auxiliar `it`: La función fibonacciI comienza definiendo una función auxiliar it que realiza el cálculo de los términos de la secuencia de Fibonacci de manera iterativa. Esta función auxiliar recibe tres argumentos: n (el índice del término), a (el término actual) y b (el siguiente término).

2. Verificación del caso base: La función auxiliar `it` verifica si el índice $\underline{n}$ es 0 o 1. Si $\underline{n == 0}$, retorna el valor $\underline{a}$; si $\underline{n == 1}$, retorna el valor $\underline{b}$. Esto cubre los casos base de la secuencia de Fibonacci.

3. Cálculo del siguiente término: Si $\underline{n > 1}$, la función `it` calcula el siguiente término de la secuencia sumando $\underline{a}$ y $\underline{b}$, y luego llama a sí misma con los valores actualizados.

4. Proceso iterativo: En lugar de realizar llamadas adicionales, la función it mantiene los valores a y b actualizados hasta alcanzar el índice deseado. El término en la secuencia de Fibonacci se calcula iterativamente, sin realizar llamadas recursivas.

5. Inicio del proceso iterativo: La función fibonacciI inicia el proceso iterativo llamando a la función auxiliar it con los valores iniciales n, 1 y 1, que representan los dos primeros términos de la secuencia de Fibonacci.

Este proceso se repite hasta que $\underline{n}$ sea 0 o 1. En ese momento, la función habrá calculado el valor del término $\underline{n}$ de la secuencia de Fibonacci y lo devolverá como resultado.

2 Informe de Correción

2.1 Máximo Común Divisor

2.1.1 Utilizando el Teorema Fundamental de la Aritmética

obtainLowerExponent

Comencemos demostrando que la función obtainLowerExpontent que recibe como parámetros dos listas, efectivamente retorna el menor exponente entre las cabezas de ambas listas. Sea $f(n, m)$ la función que retorna el menor exponente entre dos valores, $\underline{n}$ y $\underline{m}$ respectivamente y $Pf(n, m)$ la función *obtainLowerExponent* que representa a $f(n, m)$ en código, se tiene que demostrar que:

$$\forall n, m \in A : Pf(n, m) == f(n, m)$$

veamos con el siguiente caso:

Caso Pf(0, 0)

$$Pf(0, 0) \rightarrow if\ (0 <= 0)\ 0\ else\ 0 \rightarrow 0$$

$$f(0, 0) = 0$$

$$Pf(0,0) == f(0,0)$$

Caso Pf(k, k)

$$Pf(k,k) \rightarrow if\ (k <= k)\ k\ else\ k \rightarrow k$$

$$f(k,k) = k$$

$$Pf(k,k) == f(k,k)$$

entonces ahora es momento de demostrar nuestra hipotesis inductiva:
Caso Pf(k+1, k)

$$Pf(k+1,k) \rightarrow if\ (k+1 <= k)\ k+1\ else\ k \rightarrow k$$

$$f(k+1,k) = k$$

$$Pf(k+1,k) == f(k+1,k)$$

Caso Pf(k, k+1)

$$Pf(k,k+1) \rightarrow if\ (k <= k+1)\ k\ else\ k+1 \rightarrow k$$

$$f(k,k+1) = k$$

$$Pf(k,k+1) == f(k,k+1)$$

por tanto tenemos que:

$$\forall n, m \in A : Pf(n,m) == f(n,m)$$

exponentiation

Sea $f(b,a)$ la función que retorna el resultado de elevar $\underline{b}$ a la potencia $\underline{a}$ y $Pf(b,a)$ la función *exponentiation* que la implementa en código, tenemos que demostrar que:

$$\forall b, a \in A : Pf(b,a) == f(b,a)$$

para lograr esto primero comencemos con algunos casos base:
Caso Pf(2, 2)

$$Pf(2,2) \rightarrow if\ (2 == 0)\ 1\ else\ 2 * Pf(2, 2-1)$$

$$Pf(2,1) \to if\ (1 == 0)\ 1\ else\ 2 * Pf(2, 1-1)$$
$$Pf(2,0) \to if\ (1 == 0)\ 1\ else\ 2 * Pf(2, 0-1) \to 1$$
$$Pf(2,2) \to 2 * (2 * (1))$$
$$Pf(2,2) = 2 * 2 * 1 = 4$$

$$f(2,2) = 4$$

$$Pf(2,2) = f(2,2)$$

Caso Pf(base, k)

$$Pf(base,k) \to if\ (k == 0)\ 1\ else\ base * Pf(base, k-1)$$

$$Pf(base,k-1) \to if\ (k == 0)\ 1\ else\ base * (base * Pf(base, k-2))$$

si seguimos el proceso hasta que $k == 0$ llegaremos a que

$$Pf(base,k) = base * (base * base * \ldots * base) = base^k$$

$$\underbrace{base * base * \ldots * base}_{k \text{ veces}}$$

$$f(base,k) = base^k$$

$$Pf(base,k) = f(base,k)$$

es hora de usar nuestra hipotesis inductiva:
Caso Pf(base, k+1

$$Pf(base,k+1) \to if\ (k+1 == 0)\ 1\ else\ base * Pf(base, (k+1)-1)$$

$$Pf(base,k) \to if\ (k == 0)\ 1\ else\ base * (base * Pf(base, k-1))$$

pero de el anterior caso ya sabemos que $Pf(base,k) = base^k$, por lo tanto:

$$Pf(base,k+1) \to base * (base^k)$$
$$Pf(base,k+1) = base^{k+1}$$

$$f(base,k+1) = base^{k+1}$$

$$Pf(base,k+1) = f(base,k+1)$$

por tanto tenemos que:

$$\forall b, a \in A : Pf(b, a) == f(b, a)$$

mcdTFA

Sea $f(lista_1, lista_2, lista_3)$ la función que retorna el mcd entre dos valores a partir de tres listas:

- $lista_3$ siendo los factores primos en orden ascendente que comparten y descomponen a los números $\underline{n}$ y $\underline{m}$ a los cuales se les va a calcular el mcd.
- $lista_1$ y $lista_2$ los exponentes de los factores primos que componen a los números n y m respectivamente.

Tenemos la función $mcdTFA(lista_1, lista_2, lista_3)$ que implementa $f(lista_1, lista_2, lista_3)$ mediante código. Para valores de esta demostración, nos referiremos a $mcdTFA(lista_1, lista_2, lista_3)$ como $Pf(a)$ y a $f(lista_1, lista_2, lista_3)$ como $f(a)$ donde $\underline{a}$ es una tripla de la forma:

$$a = exponentesN, exponentesM, primos$$

Por el teorema fundamental de la aritmética tenemos que todo numero natural mayor que 1, se puede expresar de manera unica como producto de numeros primos. Si los primos se presentan en orden ascendente, esta descomposicion es unica. En otras palabras se tiene que:

$$n > 1 \rightarrow n = p_1 p_2 p_3 ... p_k$$

$$k \in \mathbb{N}$$

Ejemplo:

$$1000 = 2 * 2 * 2 * 5 * 5 * 5$$

esto se puede simplificar a

$$1000 = 2^3 * 5^3$$

en otras palabras, la definición anterior se puede generalizar de la siguiente forma:

$$(n = p_1^{i_1} p_2^{i_2} p_3^{i_3} p_k^{i_k}) \wedge (\forall j | 1 \leq j \leq k : i_j \geq 0)$$

entonces, dados

$$n = p_1^{i_1} p_2^{i_2} p_3^{i_3} p_k^{i_k} \wedge m = p_1^{j_1} p_2^{j_2} p_3^{j_3} p_k^{j_k}$$

se tiene que

$$mcd(n, m) = p_1^{min(i_1, j_1)} p_2^{min(i_2, j_2)} p_3^{min(i_3, j_3)} \dots p_k^{min(i_k, j_k)}$$

Ejemplo:

$$1000 = 2^3 * 5^3$$

$$10 = 2^1 * 5^1$$

entonces

$$mcd(1000, 10) = 2^1 * 5^1 = 10$$

Teniendo lo anterior en cuenta, para demostrar que efectivamente

$$Pf(a) == f(a)$$

tendremos que demostrar que

$$\forall a \in A : Pf(a) == f(a)$$

Caso a_1

Comencemos con un caso donde la longitud de la lista de factores primos que descomponen a ambos números sea 1 es decir:

$$a_1 = (2^2),\ (2^1),\ (2)$$

esto representando los argumentos necesarios para el cálculo del mcd entre 4 y 2. Entonces:

$$Pf(a_1) \rightarrow if\ (false)\ 1\ else\ exponentiation(2, obtainLowerExponent(2,1)) * Pf([],[],[])$$

$$Pf((),(),()) \rightarrow if\ (true)\ 1\ else \ldots \rightarrow 1$$

$$Pf(a_1) \rightarrow exponentiation(2, obtainLowerExponent(2,1)) * 1$$

$$Pf(a_1) \rightarrow exponentiation(2,1) * 1$$

$$Pf(a_1) = 2^1 * 1 = 2$$

$$f(a_1) = 2$$

Caso a_k

probemos ahora con el siguiente caso con listas de longitud k:

$$a_k = (i_1, i_2, \ldots, i_k), (j_1, j_2, \ldots, j_k), (p_1, p_2, \ldots, p_k)$$

esto representa los argumentos necesarios para el cálculo entre números $\underline{n}$ y $\underline{m}$

$Pf(a_k) \rightarrow if\ (primes.isEmpty)\ 1\ else\ exponentiation(p_1, obtainLowerExponent(i_1, j_1)) * Pf(a_{k-1})$

siendo $a_{k_1} = (i_2, \ldots, i_k), (j_2, \ldots, j_k), (p_2, \ldots, p_k)$

$Pf(a_{k-1}) \rightarrow if\ (primes.isEmpty)\ 1\ else\ exponentiation(p_2, obtainLowerExponent(i_2, j_2)) * Pf(a_{k-2})$

siendo $a_{k-2} = (\ldots, i_k), (\ldots, j_k), (\ldots, p_k)$

$Pf(a_{k-2} \to if\ (primes.isEmpty)\ 1\ else\ exponentiation(p_3, obtainLowerExponent(i_3, j_3)) * Pf(a_{k-3})$

si repetimos este proceso hasta que primes.isEmpty == true obtendriamos lo siguiente:

$$Pf(a_k) = p_1^{min(i_1,j_1)} p_2^{min(i_2,j_2)} p_3^{min(i_3,j_3)} \ldots p_k^{min(i_1 k, j_k)}$$

$$f(a_k) = p_1^{min(i_1,j_1)} p_2^{min(i_2,j_2)} p_3^{min(i_3,j_3)} \ldots p_k^{min(i_1 k, j_k)}$$

$$Pf(a_k) = f(a_k)$$

es hora de usar nuestra hipotesis inductiva:
Caso a_{k+1}

$$a_{k+1} = (i_1, i_2, \ldots, i_k, i_{k+1}), (j_1, j_2, \ldots, j_k, j_{k+1}), (p_1, p_2, \ldots, p_k, p_{k+1})$$

$Pf(a_{k+1}) \to if\ (primes.isEmpty)\ 1\ else\ exponentiation(p_1, obtainLowerExponent(i_1, j_1)) *$
$Pf(a_{(k+1)-1})$

$$a_k = (i_1, i_2, \ldots, i_k), (j_1, j_2, \ldots, j_k), (p_1, p_2, \ldots, p_k)$$

gracias al caso anterior sabemos que:

$$Pf(a_k) = p_1^{min(i_1,j_1)} p_2^{min(i_2,j_2)} p_3^{min(i_3,j_3)} \ldots p_k^{min(i_k,j_k)}$$

por tanto

$$Pf(a_{k+1}) = p_1^{min(i_1,j_1)} p_2^{min(i_2,j_2)} p_3^{min(i_3,j_3)} \ldots p_k^{min(i_1 k, j_k)} p_{k+1}^{min(i_{k+1}, j_{k+1})}$$

$$f(a_{k+1}) = p_1^{min(i_1,j_1)} p_2^{min(i_2,j_2)} p_3^{min(i_3,j_3)} \ldots p_k^{min(i_1 k, j_k)} p_{k+1}^{min(i_{k+1}, j_{k+1})})$$

$$Pf(a_{k+1}) = f(a_{k+1})$$

entonces

$$\forall a \in A : Pf(a) == f(a)$$

2.1.2 Utilizando el Algorimto de la División

El algoritmo de la división establece que: Dados $n \in \mathbb{N}, d \in \mathbb{N}^+$.

$$\exists q, r | 0 \leq r < d : n = qd + r$$

donde q e r son únicos.

Basados en lo anterior y en la siguiente propiedad, (Suponiendo $n \geq m$):

$$n = mq + r \Rightarrow mcd(n, m) = mcd(m, r)$$

se tiene que:

$$mcd(n, m) = \begin{cases} n & \text{Si } m = 0 \\ mcd(m, r) & \text{Si, por algoritmo de la división } n = mq + r \end{cases}$$

En otras palabras, para calcular el MCD entre dos números $\underline{n}$ y $\underline{m}$, solo hay que ver cuántas veces cabe $\underline{m}$ en $\underline{n}$. A esto lo llamaremos cociente ($\underline{q}$). Luego, calculamos la diferencia entre $\underline{n}$ y $\underline{mq}$, a esto lo llamaremos residuo ($\underline{r}$). Repetimos este proceso ahora con $n = m$ y $m = r$, la cantidad de veces necesarias hasta llegar al caso base donde $m = 0$ en el cual el mcd entre ambos números será $\underline{n}$.

Una propiedad del mcd es:

$$mcd(n, m) = d \Rightarrow \exists x, y | : d = xn + my$$

El algoritmo de Euclides nos permite calcular $\underline{x}$ e $\underline{y}$ tales que $mcd(n, m) = nx + my$. Ejemplo: $mcd(28, 10)$

Paso	n	m	q	r
0	28	10	2	8
1	10	8	1	2
2	8	2	4	0
3	2	0		

Tabla 1. Calculo de $mcd(28, 10)$.

ahora nos regresamos:

Paso	r
1	$2 = 10 - 8$
0	$8 = 28 - 10 * 2$

Tabla 2. Calculo de x e y.

entonces tenemos:

$$2 = 10 - (28 - 10 * 2) = 28(-1) + 10(3)$$

$$x = -1 \wedge y = 3$$

Supongamos que encontramos los coeficientes x_1, y_1 para $(n, n\%m$:

$$n * x_1 + m * y_1 = d$$

y queremos encontrar el par:

$$n * x + m * y = d$$

podemos representar $n\%m$ como:

$$n\%m = n - \lfloor \frac{n}{m} \rfloor * m$$

reemplazando en la ecuación para x_1, y_1:

$$d = m * x_1 + (n - \lfloor \frac{n}{m} \rfloor * m) * y_1$$

organizando:

$$d = n * y_1 + m(x_1 - y_1 * \lfloor \frac{n}{m} \rfloor)$$

entonces:

$$x = y_1 \wedge y = x_1 - y_1 * \lfloor \frac{n}{m} \rfloor$$

mcdEBez

Sea $f : \mathbb{N}^2 \to \mathbb{N}^3$ una función que recibe dos números naturales $\underline{n}$ y $\underline{m}$ y retorna una tupla de tres números naturales (d, x, y) donde $\underline{d}$ es el mcd entre $\underline{n}$ y $\underline{m}$, y $\underline{x}$ e $\underline{y}$ los coeficientes de Bezout y sea $Pf(n, m)$ la función que la implementa en código, tenemos que demostrar que:

$$\forall n, m \in \mathbb{N} : Pf(n, m) == f(n, m)$$

para lograrlo, nuestra hipotesis inductiva será que asumiendo como verdadero:

$$Pf(n, m) = (d, x, y)$$

entonces

$$Pf(m, n\%m) = (d, x_1, y_1)$$

comencemos con algunos casos.

Caso $Pf(n, 0)$

$$Pf(n, 0) \to if\ (true)\ (n, 1, 0)\ else \ldots \to (n, 1, 0)$$

$$f(n, 0) = (n, 1, 0)$$

$$Pf(n, 0) == f(n, 0)$$

Caso $Pf(k, n\%k$

Supongamos que $Pf(n, m)$ funciona correctamente para todos los $m < k$, entonces, queremos demostrar que de igual forma funciona para $m = k$.

Sabemos que:

$$Pf(k, n\%k) = (d, x_1, y_1)$$

donde:

$$d = mcd(k, n\%k)$$

$$d = k * x_1 + (n\%k) * y_1$$

Queremos verificar que:

$$Pf(n, k) = (d, x, y)$$

donde:

$$x = y_1$$

$$y = x_1 - y_1 * \lfloor \frac{n}{k} \rfloor$$

sabemos que:

$$n\%k = n - \lfloor \frac{n}{k} \rfloor * k$$

reemplazamos en $d = k * x_1 + (n\%k) * y_1$:

$$d = k * x_1 + (n - \lfloor \frac{n}{k} \rfloor * k) * y_1$$

simplificamos:

$$d = n * y_1 + k * (x_1 - \lfloor \frac{n}{k} \rfloor * y_1)$$

por tanto:

$$x = y_1 \wedge y = x_1 - y_1 * \lfloor \frac{n}{k} \rfloor$$

entonces:

$$d = n * x + m * y$$

y la relación se mantiene. Por lo tanto la función $Pf(n, m)$ devuelve la tupla (d, x, y) para todos los pares ordenados (n, m), entonces:

$$\forall n, m \in \mathbb{N} : Pf(n, m) == f(n, m)$$

2.2 Fibonacci

2.2.1 Fibonacci Árbol

$$\forall n \in \mathbb{N} : P\ fibonacciA(n) == F(n)$$

Caso base:

Para $n = 0$ y $n = 1$.

Para $n = 0$:

Calculamos lo que devuelve `fibonacciA` usando el modelo de sustitución:

$$fibonacciA(0) \rightarrow \texttt{if (0 == 0) 1}$$

Por lo tanto:

$$fibonacciA(0) \rightarrow 1$$

Por otro lado, $F(0) = 1$.
Podemos concluir que:

$$fibonacciA(0) == F(0)$$

Para $n = 1$:
Calculamos lo que devuelve `fibonacciA` usando el modelo de sustitución:

$$fibonacciA(1) \rightarrow \texttt{if (1 == 0) 1 else if (1 == 1) 1}$$

Por lo tanto:

$$fibonacciA(1) \rightarrow 1$$

Por otro lado, $F(1) = 1$.
Podemos concluir que:

$$fibonacciA(1) == F(1)$$

Caso de inducción:
Para $n = k+1$, donde $k \geq 1$. Hay que demostrar que: $P\ fibonacciA(k) == F(k) \rightarrow P\ fibonacciA(k+1) == F(k+1)$.
Empecemos por calcular lo que devuelve `fibonacciA` usando el modelo de sustitución:

$fibonacciA(k+1) \rightarrow$ `if (k + 1 == 0) 1 else if (k + 1 == 1) 1 else fibonacciA((k + 1) - 1) + fi`

Esto se simplifica a:

$$fibonacciA(k+1) \rightarrow fibonacciA(k) + fibonacciA(k-1)$$

Aplicando la hipótesis de inducción (HI):

$$\rightarrow F(k) + F(k-1) = F(k+1)$$

Por otro lado, $F(k+1) = F(k) + F(k-1)$.
Podemos concluir que:

$$fibonacciA(k+1) == F(k+1)$$

Por lo tanto, por inducción, podemos concluir que:

$$\forall n \in \mathbb{N} : P\ fibonacciA(n) == F(n)$$

Esto demuestra que la función `fibonacciA` implementada en Scala calcula correctamente los números de la secuencia de Fibonacci para cualquier número natural n.

2.2.2 Fibonacci Iterativo

Precondición

Dado un estado $s = (n, a, b)$:
La precondición inv(s) establece que:

$$a = \text{fib}(N - n + 1) \text{ y } b = \text{fib}(N - n).$$

Respuesta

La función `fibonacciI(n)` utiliza una función auxiliar `it(n, a, b)` donde:
Si $s = (n, a, b)$ es un estado final, entonces respuesta(s) = b.

Demostración

1. Probar que la precondición se cumple para el estado inicial
El estado inicial es $s_0 = (N, 1, 1)$:

$$\text{inv}(s_0) = \text{inv}(N, 1, 1) : \ a = \text{fib}(N - N + 1) \text{ y } b = \text{fib}(N - N).$$

Calculamos:

$$\text{fib}(N - N + 1) = \text{fib}(1) = 1$$
$$\text{fib}(N - N) = \text{fib}(0) = 1$$

Por lo tanto, inv(s_0) $\rightarrow$ True.

2. Probar que si s es un estado final, entonces la respuesta es correcta
Sea $s_f = (0, a, b)$ un estado final.
Por ser estado, se tiene:

$$a = \text{fib}(N - 0 + 1) = \text{fib}(N + 1)$$
$$b = \text{fib}(N - 0) = \text{fib}(N)$$

Por ser estado final, sabemos que $n = 0$.
Por lo tanto, si $s_f = (0, a, b)$ es un estado final, se tiene que $b = \text{fib}(N)$. Como respuesta(s) = b, se concluye que respuesta(s) = fib(N).

3. Probar que si no es un estado final, entonces (esFinal(s) $\wedge$ inv(s)) $\rightarrow$ inv(trans(s))
Sea $s = (n, a, b)$ un estado no final.

Si $n \neq 0$, entonces el estado $s' = (n-1, b, a+b)$ es un nuevo estado generado por la transformación `it(n - 1, b, a + b)`.

Necesitamos probar que este nuevo estado s' cumple la precondición $\text{inv}(s')$.

Por hipótesis de inducción sabemos que:

$$a = \text{fib}(N - n + 1)$$

$$b = \text{fib}(N - n)$$

Queremos probar que en el nuevo estado $s' = (n-1, b, a+b)$:

$$b = \text{fib}(N - (n-1) + 1) = \text{fib}(N - n + 2)$$

$$a + b = \text{fib}(N - (n-1))$$

Por la definición de Fibonacci:

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

Como $a = \text{fib}(N - n + 1)$ y $b = \text{fib}(N - n)$, entonces:

$$a + b = \text{fib}(N - n + 1) + \text{fib}(N - n) = \text{fib}(n - 1)$$

Por lo tanto, $\text{inv}(s')$ se cumple.

4. Demostrar que el algoritmo termina

Si $s = (n, a, b)$ no es un estado final, el nuevo estado $s' = (n-1, b, a+b)$ se genera. Cada vez que se transforma, n disminuye en 1. Dado que n es un número natural (entero no negativo), eventualmente n alcanzará 0. Por lo tanto, en un número finito de pasos, se llegará a un estado final $s_f = (0, a, b)$.

3 Casos de Prueba

3.1 Máximo Común Divisor

3.1.1 mcdTFA

Para este algoritmo se prepararon las siguientes pruebas:

1. $mcd(30, 12)$ resultado esperado: 6
2. $mcd(28, 12)$ resultado esperado: 4
3. $mcd(56, 7)$ resultado esperado: 7
4. $mcd(57, 15)$ resultado esperado: 3
5. $mcd(44, 8)$ resultado esperado: 4

Resultados Obtenidos

```
val res0: Int = 6
val res1: Int = 4
val res2: Int = 7
val res3: Int = 3
val res4: Int = 4
```

Resultados de pruebas 1. Función mcdTFA.

En base a los resultados y gracias a nuestra demostración que efectivamente

$$\forall a \in A : Pf(a) == f(a)$$

lo anterior de forma teórica. Creemos que las pruebas aquí expuestas son suficientes para junto a la demostración realizada, probar la fiabilidad de nuestra función.

3.1.2 mcdEBez

Para este algoritmo se prepararon las siguientes pruebas:

1. $mcdEBez(28, 10)$
2. $mcdEBez(35, 15)$
3. $mcdEBez(48, 18)$
4. $mcdEBez(56, 21)$
5. $mcdEBez(84, 30)$

Prueba 1

Paso	n	m	q	r
0	28	10	2	8
1	10	8	1	2
2	8	2	4	0
3	2	0		

Tabla 3. Calculo de $mcd(28, 10)$.

ahora nos regresamos:

Paso	r
1	$2 = 10 - 8$
0	$8 = 28 - 10 * 2$

Tabla 4. Calculo de x e y para la prueba 1.

entonces tenemos:

$$2 = 10 - (28 - 10 * 2) = 28(-1) + 10(3)$$

$$x = -1 \wedge y = 3$$

Prueba 2

Paso	n	m	q	r
0	35	15	2	5
1	15	5	3	0
2	5	0		

Tabla 5. Cálculo de $mcd(35, 15)$.

Ahora nos regresamos:

Paso	r
1	$5 = 15 - 3 * 5$
0	$5 = 35 - 2 * 15$

Tabla 6. Cálculo de x e y para la Prueba 2.

Entonces tenemos:

$$5 = 35(1) + 35(-2)$$

$$x = 1 \wedge y = -2$$

Prueba 3

Paso	n	m	q	r
0	48	18	2	12
1	18	12	1	6
2	12	6	2	0
3	6	0		

Tabla 7. Cálculo de $mcd(48, 18)$.

Ahora nos regresamos:

Paso	r
2	$6 = 18 - 1 * 12$
1	$12 = 48 - 2 * 18$

Tabla 8. Cálculo de x e y para la Prueba 3.

Entonces tenemos:

$$6 = 18 - 1 * (48 - 2 * 18)$$

$$6 = 48(-1) + 18(3)$$

$$x = -1 \wedge y = 3$$

Prueba 4

Paso	n	m	q	r
0	56	21	2	14
1	21	14	1	7
2	14	7	2	0
3	7	0		

Tabla 9. Cálculo de $mcd(56, 21)$.

Ahora nos regresamos:

Paso	r
2	$7 = 21 - 1 * 14$
1	$14 = 56 - 2 * 21$

Tabla 10. Cálculo de x e y para la Prueba 4.

Entonces tenemos:

$$7 = 21 - 1 * (56 - 2 * 21)$$
$$7 = 56(-1) + 21(3)$$
$$x = - \wedge y = 3$$

Prueba 5

Paso	n	m	q	r
0	84	30	2	24
1	30	24	1	6
2	24	6	4	0
3	6	0		

Tabla 1. Cálculo de $mcd(84, 30)$.

Ahora nos regresamos:

Paso	r
2	$6 = 30 - 1 * 24$
1	$24 = 84 - 2 * 30$

Tabla 12. Cálculo de x e y para la Prueba 5.

Entonces tenemos:

$$6 = 30 - 1 * (84 - 2 * 30)$$
$$6 = 84(-1) + 30(3)$$
$$x = -1 \wedge y = 3$$

Resultados Obtenidos

```
val res5: (Int, Int, Int) = (2,-1,3)
val res6: (Int, Int, Int) = (5,1,-2)
val res7: (Int, Int, Int) = (6,-1,3)
val res8: (Int, Int, Int) = (7,-1,3)
val res9: (Int, Int, Int) = (6,-1,3)
```

Resultados de pruebas 2. Función mcdEBez.

En base a los resultados y gracias a nuestra demostración que efectivamente

$$\forall a \in A : Pf(a) == f(a)$$

las pruebas aquí expuestas para esta función dan fiabilidad de su capacidad para lograr lo planteado en el enunciado del problema.

3.2 Fibonacci

3.2.1 Fibonacci Árbol

Para este algoritmo recursivo se prepararon las siguientes pruebas:

1. `fibonacciA(5)` resultado esperado: 8
2. `fibonacciA(10)` resultado esperado: 89
3. `fibonacciA(20)` resultado esperado: 10946
4. fibonacciA(3) resultado esperado: 3
5. fibonacciA(19) resultado esperado: 6765

Resultados Obtenidos:

```
val res0: Int = 8
val res1: Int = 89
val res2: Int = 10946
val res3: Int = 3
val res4: Int = 6765
```

Resultados de pruebas para la función `fibonacciA`.

En base a los resultados obtenidos y nuestra demostración se evidencia que

$$\forall n \in \mathbb{N} : \text{fibonacciA}(n) == F(n)$$

Dado que $F(n)$ representa el enésimo número en la secuencia de Fibonacci, podemos afirmar que los resultados de las pruebas presentadas, junto con la demostración teórica, son suficientes para confirmar la fiabilidad de nuestra función.

3.2.2 Fibonacci Iterativo

Para este algoritmo iterativo se prepararon las siguientes pruebas:

1. `fibonacciI(7)` resultado esperado: 21
2. `fibonacciI(9)` resultado esperado: 55
3. `fibonacciI(21)` resultado esperado: 17711
4. fibonacciI(17) resultado esperado: 2584
5. fibonacciI(30) resultado esperado: 1346269

Resultados Obtenidos:

```
val res0: Int = 21
val res1: Int = 55
val res2: Int = 17711
val res3: Int = 2584
val res4: Int = 1346269
```

Resultados de pruebas para la función `fibonacciI`.

En base a los resultados obtenidos y gracias a nuestra demostración se prueba que efectivamente

$$\forall n \in \mathbb{N} : \text{fibonacciI}(n) == F(n)$$

Donde $F(n)$ es el enésimo número en la secuencia de Fibonacci. Podemos concluir que las pruebas aquí expuestas son suficientes para, junto a la demostración realizada, probar la fiabilidad de nuestra función.

4 Conclusiones

En este taller, hemos abordado la resolución de problemas mediante la implementación de algoritmos en un lenguaje de programación, explorando diversas técnicas recursivas e iterativas. Este taller ha sido especialmente revelador, ya que ha demostrado el poder y la elegancia de la recursión, una herramienta que, aunque introducida en cursos posteriores, nunca se había considerado en profundidad en nuestro trabajo previo con estructuras de control más convencionales.

La experiencia al resolver este taller ha demostrado que la recursión puede reemplazar estructuras de control comunes, como los ciclos for y while, así como incluso algunas estructuras condicionales complejas. Esta capacidad de resolver problemas mediante la descomposición en subproblemas más simples ha ampliado significativamente nuestra comprensión de la programación.

Además, al comparar la implementación recursiva con la iterativa, hemos observado que la recursión puede ofrecer soluciones elegantes y claras. Sin embargo, también hemos aprendido que, en ciertos casos, puede ser menos eficiente debido a la sobrecarga de llamadas recursivas y el consumo de memoria. La capacidad de expresar soluciones de manera más concisa es un gran beneficio, pero es crucial considerar la eficiencia y el rendimiento, especialmente en problemas con grandes entradas.

En resumen, este taller no solo ha mejorado nuestra habilidad para aplicar técnicas recursivas, sino que también ha subrayado la importancia de elegir el enfoque adecuado para cada problema, equilibrando la elegancia de la solución con la eficiencia y el rendimiento necesarios.