Soluciones de Ejercicios Seleccionados de la Guía 2

Algoritmos y Estructuras de Datos II, DC, UBA.

Segundo cuatrimestre 2020

Índice

1. Complejidad

Ejercicio 1:

Sean $f, g\ : \mathbb{N} \to \mathbb{N}$ y supongamos que está definido el límite $\lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} = l \in \mathbb{R}_{\geq 0} \cup \{+\infty\}$. Probar que:

a) $0 < l < +\infty$ si y sólo si $f \in \Theta(g)$

b) $l = +\infty$ si y sólo si $f \in \Omega(g)$ y $f \notin O(g)$

c) $l = 0$ si y sólo si $f \in O(g)$ y $f \notin \Omega(g)$

Vayamos en el orden de los items.

Ítem a)

Veamos las dos implicaciones.

($\Rightarrow$) Repasemos que, por definición, $\lim\limits_{n\to+\infty} f(n) = L$ significa que para todo $\epsilon > 0$ existe un $n_0 \in \mathbb{N}$ tal que para todo $n > n_0$ se tiene que $|f(n) - \mathrm{L}| < \epsilon$.

Sabemos que el límite es $0 < l < +\infty$. Luego, a partir de algún n_0 que depende de ε, se tiene que:

$$\|\frac{f(n)}{g(n)} - l\| < \varepsilon \iff -\varepsilon < \frac{f(n)}{g(n)} - l < \varepsilon \iff (-\varepsilon + l)g(n) < f(n) < (\varepsilon + l)g(n)$$

Como esta desigualdad es para un ε arbitrario, y como $l > 0$, podemos tomar $0 < \varepsilon_0 < l$, por ejemplo $\varepsilon_0 := l/2$, y definir $k_1 := (-\varepsilon_0 + l)$, $k_2 := (\varepsilon_0 + l)$. De esta forma $k_1, k_2 > 0$. Entonces la definición de límite asegura que existe un n_0 tal que para todo $n > n_0$ se verifica:

$$k_1 g(n) < f(n) < k_2 g(n) \quad \forall n > n_0$$

Probando así que $f(n) \in \Theta(g(n))$

($\Leftarrow$) Sabemos que $f(n) \in \Theta(g(n))$, entonces existe un n_0 y $0 < k_1, k_2 < +\infty$ tal que:

$$k_1 g(n) < f(n) < k_2 g(n) \iff k_1 < \frac{f(n)}{g(n)} < k_2 \quad \forall n > n_0$$

Como vale para todo $n > n_0$ podemos tomar límite en las tres partes de la desigualdad, resultando en:

$$\lim_{n\to+\infty} k_1 < \lim_{n\to+\infty} \frac{f(n)}{g(n)} < \lim_{n\to+\infty} k_2$$

Cómo k_1, k_2 son constantes respecto a n y $0 < k_1, k_2 < +\infty$ entonces el $0 < \lim\limits_{n\to+\infty} \frac{f(n)}{g(n)} < +\infty$

Ítem b)

Veamos las dos implicaciones:

($\Rightarrow$) Sabemos que el límite se va a $+\infty$, entonces para todo $M > 0$ existe un n_0 tal que para todo $n > n_0$ se verifica que $\frac{f(n)}{g(n)} > M$. Equivalentemente:

$$f(n) > Mg(n) \quad \text{para todo } n > n_0.$$

Es fácil concluir por definición entonces que $f(n) \in \Omega(g)$. Además por el item a) podemos ver que f no puede ser $O(g)$, pues si lo fuese entonces $f(n) \in \Theta(g(n))$ y el límite de f/g sería un número, contradiciendo la hipótesis de $l = +\infty$.

($\Leftarrow$) Sabemos que $f \notin O(g)$, y por lo tanto $f \notin \Theta(g)$. Como el ítem a) es una equivalencia lógica ("si y sólo si"), eso nos dice que en este caso l debe ser 0 o $+\infty$. Notar que el límite no puede ser negativo, pues las funciones son estrictamente positivas. Además, por definición de $f \in \Omega(g)$ existen una constante $k > 0$ y un n_0 tal que para todo $n > n_0$ se tiene que $0 < kg(n) < f(n)$. Equivalentemente:

$$0 < k < \frac{f(n)}{g(n)} \quad \text{para todo } n > n_0.$$

Tomando límite concluimos que $\lim_{n\to+\infty} \frac{f(n)}{g(n)} \geq k > 0$. Además, ya sabíamos que el límite de $\frac{f(n)}{g(n)}$ sólo podía ser 0 o $+\infty$, de modo que necesariamente deber ser $+\infty$.

Ítem c)

Veamos las dos implicaciones:

($\Rightarrow$) Sabemos que el límite es 0, entonces para todo $\varepsilon > 0$ existe un n_0 tal que para todo $n > n_0$ se tiene que $\|\frac{f(n)}{g(n)}\| < \varepsilon$. Como f, g no toman valores negativos, esto implica:

$$f(n) < \varepsilon g(n) \quad \text{para todo } n > n_0.$$

Consideremos algún $\varepsilon_0 > 0$ fijo, por ejemplo $\varepsilon_0 := 1$. Tomando $k_1 := \varepsilon_0$ concluimos que $f(n) \in O(g(n))$ y con un razonamiento análogo al de la ida en el ítem b) concluimos que $f(n) \notin \Omega(g(n))$

($\Leftarrow$) Utilizando lo que ya probamos en a) y b) facilmente podemos ver esta implicación. Sabemos que $f \in O(g)$ y $f \notin \Omega(g)$. Por una parte esto nos dice en particular que $f \notin \Theta(g)$ y por la equivalencia del ítem a) sabemos entonces que el límite de f/g no puede ser una constante en $(0, +\infty)$. El límite tampoco puede ser $+\infty$ pues el ítem b) da una condición necesaria y suficiente para esto. Notar que el límite tampoco puede ser negativo, pues $f, g > 0$. Así, la única alternativa es que el límite sea 0.

Ejercicio 2:

Determinar el orden de complejidad temporal de peor caso de los siguientes algoritmos, asumiendo que todas las operaciones sobre arreglos y matrices toman tiempo $O(1)$.

La complejidad se debe calcular en función de una medida de los parámetros de entrada, por ejemplo, la cantidad de elementos en el caso de los arreglos y matrices y el valor en el caso de parámetros naturales.

Vamos a llamar $n = long(A)$.

Sumatoria, que calcula la sumatoria de un arreglo de enteros:

function Sumatoria(arreglo A)
  int i, total; ▷ $\mathcal{O}(1)$
  total := 0; ▷ $\mathcal{O}(1)$
  for i := 0 . . . Long(A) - 1 **do** ▷ Este `for` se ejecutará n veces por lo que la complejidad será la sumatoria de 0 a $n-1$ de la complejidad de las operaciones internas del ciclo
    total := total + A[i]; ▷ $\mathcal{O}(1)$
  end for ▷ Complejidad del ciclo: $\sum_{i=0}^{n-1} \mathcal{O}(1) = \mathcal{O}(n)$
end function

Más formalmente, tenemos que la complejidad total es:

$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=0}^{n-1} \mathcal{O}(1) = \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$

Complejidad total en el peor caso: $\mathcal{O}(n)$

SumatoriaLenta, que calcula la sumatoria de n, definida como la suma de todos los enteros entre 1 y n, de forma poco eficiente:

function SumatoriaLenta(natural N)
  int i, total; ▷ $\mathcal{O}(1)$
  total := 0; ▷ $\mathcal{O}(1)$
  for i := 1 . . . n **do** ▷ Este `for` se ejecutará n veces por lo que la complejidad va a ser $\mathcal{O}(n)$ la sumatoria de n veces la complejidad de las operaciones internas del ciclo
    for j := 1 . . . i **do** ▷ Este `for` se ejecutará **en el peor caso** n veces[1] por lo que la complejidad será la sumatoria de n veces la complejidad de las operaciones internas del ciclo
      total := total + 1; ▷ $\mathcal{O}(1)$
    end for ▷ Complejidad del ciclo interno (informal): $\mathcal{O}(n) * \mathcal{O}(1) = \mathcal{O}(n)$
  end for ▷ Complejidad del ciclo externo (informal): $\mathcal{O}(n) * \mathcal{O}(n) = \mathcal{O}(n^2)$
end function

Más formalmente, tenemos que la complejidad total es:

$$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{n}(\sum_{j=1}^{i}(\mathcal{O}(1))$$

$$\mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{n}(\sum_{j=1}^{n}\mathcal{O}(1) = \mathcal{O}(1) + \mathcal{O}(1) + \sum_{i=1}^{n}\mathcal{O}(n) = \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$$

Complejidad total: $\mathcal{O}(n^2)$

InsertionSort, que ordena un arreglo pasado como parámetro:

function InsertionSort(arreglo A)
  int i, j, valor; ▷ $\mathcal{O}(1)$
  for i := 0 . . . Long(A) - 1 **do** ▷ Este `for` se ejecutará n veces por lo que la complejidad será la sumatoria de 0 a $n-1$ de la complejidad de las operaciones internas del ciclo
    valor := A[i]; ▷ $\mathcal{O}(1)$
    j := i - 1; ▷ $\mathcal{O}(1)$
    while j $\geq$ 0 $\land$ a[j] > valor **do** ▷ En el peor caso, j tendrá un valor del orden de n, por lo que el `while` se ejecutará n veces. Que $a[j] > valor$ no va a cambiar la complejidad en *peor caso*[2] mientras no haya ninguna precondición al respecto ya que entonces podría suceder que $a[j] > valor$ para todo j entre 0 e i (o sea, que el arreglo original esté ordenado de mayor a menor).
      A[j+1] := A[j]; ▷ $\mathcal{O}(1)$
      j := j - 1; ▷ $\mathcal{O}(1)$
    end while ▷ Complejidad del ciclo interno: $\mathcal{O}(n)$ (ver explicación abajo)
    A[j+1] := valor; ▷ $\mathcal{O}(1)$
  end for ▷ Complejidad del ciclo externo: $\mathcal{O}(n^2)$ (ver explicación abajo)
end function

Complejidad total: $\mathcal{O}(n^2)$

La complejidad del ciclo interno es la sumatoria de las ejecuciones de $j = i - 1$ hasta 0 inclusive. Esto es lo mismo que $\sum_{j=0}^{i}$. Pero en el peor caso, $i = n - 1$. Entonces nos queda:

$$\sum_{j=0}^{i}(\mathcal{O}(1) + \mathcal{O}(1)) = \sum_{j=0}^{n-1}\mathcal{O}(1) = \mathcal{O}(n)$$

Por otro lado, la complejidad del ciclo externo es la sumatoria de las ejecuciones internas de $i = 0$ hasta $n - 1$. Esto es:

$$\sum_{i=0}^{n-1}(\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(1)) = \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n)$$

Pero por álgebra de órdenes, esto es: $\mathcal{O}(n^2)$.

Finalmente, tenemos $\mathcal{O}(1) + \mathcal{O}(n^2)$ que es trivialmente $\mathcal{O}(n^2)$

BúsquedaBinaria, que determina si un elemento se encuentra en un arreglo, que debe estar ordenado:

[1] notar que en el peor caso $i = n$
[2] notar que si nos piden el mejor caso, ahí sí influye.

```
function BúsquedaBinaria(arreglo A, elem valor)
    int izq := 0, der := Long(A) - 1;                    ▷ O(1)
    while izq < der do                                   ▷ La diferencia entre los valores de la guarda va disminuyendo a la mitad cada vez. En otras palabras, estamos dividiendo por 2 su valor en cada iteración, por lo que el ciclo se ejecutará una cantidad de veces del orden de log(n).
        int medio := (izq + der) / 2;                    ▷ O(1)
        if valor < A[medio] then
            der := medio;                                ▷ O(1)
        else
            izq := medio;                                ▷ O(1)
        end if
    end while                                            ▷ Complejidad del ciclo: Σ_{i=0}^{log(n)} O(1) + O(1) + O(1)= 3 * O(log(n)) = O(log(n))
    return A[izq] = valor;                               ▷ O(1)
end function
```

Complejidad total: $\mathcal{O}(1) + \mathcal{O}(log(n)) + \mathcal{O}(1) = \mathcal{O}(log(n))$

ProductoMat, que dadas dos matrices A (de $p \times q$) y B (de $q \times r$) devuelve su producto AB (de $p \times r$):

```
function ProductoMat(matriz A, matriz B)
    int fil, col, val, colAFilB;                         ▷ O(1)
    matriz res(Filas(A), Columnas(B));                   ▷ O(1)
    for fil := 0 ... Filas(A) - 1 do                     ▷ Este for se ejecutará p veces por lo que la complejidad será la sumatoria de p veces la complejidad de las operaciones internas del ciclo
        for col := 0 ... Columnas(B) - 1 do              ▷ Este for será del orden de r
            val := 0;                                    ▷ O(1)
            for colAFilB := 0 ... Columnas(A) - 1 do     ▷ Este for será del orden de q
                val := val + (A[fil][colAFilB] * B[colAFilB][col]);   ▷ O(1) (no importa todas las operaciones que haga sobre matrices. Si son constantes, la complejidad resultante es constante)
            end for                                      ▷ Complejidad del ciclo interno: Σ_{i=0}^{q-1} O(1) = O(q)
            res[fil][col] := val;                        ▷ O(1)
        end for                                          ▷ Complejidad del ciclo intermedio: Σ_{i=0}^{r-1} (O(1) + O(q) + O(1)) = O(r * q)
    end for                                              ▷ Complejidad del ciclo externo: Σ_{i=0}^{p-1} O(r * q) = O(p * r * q)
    return res;                                          ▷ O(1)
end function
```

Complejidad total: $\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(p * q * r) + \mathcal{O}(1) = \mathcal{O}(p * q * r)$

Ejercicio 3:

Para cada una de las siguientes afirmaciones, decida si son verdaderas o falsas y justifique su decisión.

1. $\mathcal{O}(n^2) \cap \Omega(n) = \Theta(n^2)$
2. $\Theta(n) \cup \Theta(n \log n) = \Omega(n \log n) \cap \mathcal{O}(n)$

   Erratas: $\Theta(n) \cup \Theta(n \log n) = \mathcal{O}(n \log n) \cap \Omega(n)$
3. Existe una $f : \mathbb{N} \to \mathbb{N}$ tal que para toda $g : \mathbb{N} \to \mathbb{N}$ se cumple que $g \in \mathcal{O}(f)$.
4. Sean $f, g : \mathbb{N} \to \mathbb{N}$, entonces se cumple que $\mathcal{O}(f) \subseteq \mathcal{O}(g)$ o $\mathcal{O}(g) \subseteq \mathcal{O}(f)$ (es decir, el orden sobre funciones dado por la inclusión de la O es total).

1. FALSO - Consideremos el contraejemplo de $\mathcal{O}(n \cdot log(n))$. Es fácil ver que pertenece a $\mathcal{O}(n^2)$ y a $\Omega(n)$. Sin embargo, no pertenece a $\Theta(n^2)$.

2. FALSO - El conjunto $\Omega(n \log n) \cap \mathcal{O}(n)$ es $\emptyset$.

   Erratas: también FALSO. Consideremos una función $f \in \Theta(n \log(\log(n)))$. Esto significa que es $\Omega(n \log(\log(n)))$ y $\mathcal{O}(n \log(\log(n)))$. Es claro, entonces que pertenece a $\mathcal{O}(n \log(n))$ y a $\Omega(n)$ pero no a $\Omega(n \log(n))$ ni a $\mathcal{O}(n)$. Por lo tanto, $f \notin \Theta(n \log(n))$ ni $f \notin \Theta(n)$. En otras palabras, $\Theta(n \log(\log(n))) \not\subset \Theta(n) \cup \Theta(n \log n)$ pero está incluido en $\mathcal{O}(n \log n)$ y en $\Omega(n)$.

3. FALSO - Supongamos que efectivamente $\exists f : \mathbb{N} \to \mathbb{N} / \forall g : \mathbb{N} \to \mathbb{N}, g \in \mathcal{O}(f)$. Considero $h : \mathbb{N} \to \mathbb{N} / h(x) = f(x)^2$. Es claro que $h \in \mathcal{O}(f^2)$ pero $h \notin \mathcal{O}(f)$. ¡Absurdo!

4. FALSO - Definamos $f(x) = x$ y

   $$g(x) = \begin{cases} 0 & \text{si x es par} \\ \\ x^2 & \text{sino} \end{cases}$$

   Veamos que $f \notin \mathcal{O}(g)$. Supongamos que sí, ie, $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ / f(x) \leq g(x) \forall n > n_0$. Ahora bien, tomo $m = 2n$. Como m es par, $g(m) = 0$. Sin embargo, $f(m) = m > 0$. ¡Absurdo!

   Ahora veamos que $g \notin \mathcal{O}(f)$. Nuevamente, supongamos que sí y análogamente, tomemos $m = 2n + 1$. Ahora, $g(m) = m^2 > m = f(m)$. Por lo tanto, tampoco se cumple que $g \in \mathcal{O}(f)$.

   Ahora bien, por la propiedad vista en clase, $f \in \mathcal{O}(f)$ y $g \in \mathcal{O}(g)$. Pero en este caso ni $\mathcal{O}(f) \not\subset \mathcal{O}(g)$ ni $\mathcal{O}(g) \not\subset \mathcal{O}(f)$, por lo que la afirmación es falsa.

2. Invariante de representación y función de abstracción

Ejercicio 1:

Primero, el invariante de representación en español:

1. El grado debe ser igual a n
2. El elemento n-ésimo de coef. debe ser distinto a cero

Notar que como son `nat` no hay números negativos. Entonces, el invariante de representación en lógica de primer orden queda:

$$rep(e) = e.grado = n \wedge_{\text{L}} e.coef[e.grado] > 0$$

Lo siguiente es la función de abstracción. Recordemos que podemos hacerla tanto a partir de los observadores como de los generadores. En este caso, lo hacemos a partir de los generadores por comodidad, pero ambas formas son posibles (pista para hacerlo con observadores, hacer otra operación que se llame *evaluar'* que reciba el arreglo de la estructura, un natural, y devuelva el resultado de aplicar la evaluación sobre n):

$$abs(e) = ConstruirPolinomio(e, 0)$$

ConstruirPolinomio(e, i) $\equiv$ **if** $i = e.grado$ **then**
    Cte($e.coef[i]$)
else
    Cte($e.coef[i]$) + X * ConstruirPolinomio(e, $i+1$)
fi

La función de abstracción nos dice **como interpretamos la estructura** (junto con el invariante). Puede observarse en *ConstruirPolinomio* que lo que decidimos es que el coeficiente constante se encuentre en la posición 0, y que los exponentes de cada coeficiente vayan incrementando a medida que nos movemos hacia la derecha en el arreglo. Podría haber sido al revés.

Por último, la segunda parte del ejercicio, que consiste en plantear la interfaz, y realizar el algoritmo de Evaluar:

Interfaz

se explica con: Polinomio

géneros: `polinomio`

Operaciones básicas

Cte(**in** k: `nat`) $\rightarrow res$: `polinomio`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} Cte(k)\}$

X() $\rightarrow res$: `polinomio`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} X()\}$

Suma(**in** a: `polinomio`, **in** b: `polinomio`) $\rightarrow res$: `polinomio`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} a + b\}$

Producto(**in** a: `polinomio`, **in** b: `polinomio`) $\rightarrow res$: `polinomio`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} a * b\}$

Evaluar(**in** p: `polinomio`, **in** n: `nat`) $\rightarrow res$: `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{\text{obs}} Evaluar(p, n)\}$

iEvaluar(**in** p: `polinomio`, **in** n: `nat`) $\rightarrow res$: nat

```
res ← 0
xn ← 1
for i = 0 to p.grado do                    ▷ Lo tomo inclusive
    res ← res + xn * p.coef[i]
    xn ← x * n
end for
```

Ejercicio 2:

Problema original

Primero, el invariante de representación en español:

1. La longitud debe ser mayor a 0 (observar que no hay ningún generador *vacio*)
2. La longitud de la palabra debe ser igual a *long*
3. La palabra debe ser palíndromo

Pasemos estos predicados a lógica de primer orden:

$$
\begin{array}{ll}
(1) & e.long > 0 \\
(2) & long(e.palabra) = e.long \\
(3) & (\forall i : nat)\ i < long(e.palabra) \Rightarrow_{\text{L}} iesimo(e.palabra, i) = iesimo(e.palabra, long(e.palabra) - 1 - i)
\end{array}
$$

Notar que cuando hacemos *long(e.palabra)*, ese *long* es el del TAD Secuencia.

Con estos predicados, definimos nuestro invariante de representación:

$$rep(e) = (1) \wedge (2) \wedge (3)$$

La función de abstracción es la siguiente:

$$abs(e) = p : Palindromo \mid ver(p) = e.palabra$$

La interfaz:

Interfaz

se explica con: PALINDROMO

géneros: `palindromo`

Operaciones básicas

MEDIO(**in** $a : \alpha$) $\rightarrow$ res : `palindromo`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} medio(a)\}$

MEDIODOBLE(**in** $a : \alpha$) $\rightarrow$ res : `palindromo`
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} medioDoble(a)\}$

AGREGAR(**in** $a : \alpha$, **in/out** p : `palindromo`)
Pre $\equiv$ $\{p = p_0\}$
Post $\equiv$ $\{p =_{\text{obs}} agregar(a, p_0)\}$

VER(**in** p : `polinomio`) $\rightarrow$ res : `secu`(α)
Pre $\equiv$ {true}
Post $\equiv$ $\{res =_{\text{obs}} ver(p)\}$

---

iVer(**in** p : `palindromo`) $\rightarrow$ res : secu(α)

```
res ← p.palabra        ▷ No aclaré en la signatura, pero hay aliasing
```

Problema modificado

Lo que nos pide el ejercicio es considerar la representación donde *palabra* solo guarda la **mitad** del palíndromo. Planteemos los predicados en español del invariante nuevamente:

1. La longitud debe ser mayor a 0
2. *long* debe ser el doble de la longitud de *palabra* (si el palíndromo tiene longitud par) o el doble menos uno (si el palíndromo tiene longitud impar).

Pasemos estos predicados a lógica de primer orden:

$$\begin{aligned}&(1) \quad e.long > 0\\ &(2) \quad e.long = 2 * long(e.palabra) \lor e.long = 2 * long(e.palabra) - 1\end{aligned}$$

Con estos predicados, definimos nuestro invariante de representación:

$$rep(e) = (1) \land (2)$$

La función de abstracción es la siguiente:

$$\begin{aligned}abs(e) = p : Palindromo \mid\\ &(e.long \,\%2 = 0 \Rightarrow ver(p) = e.palabra \,\&\, reverso(e.palabra)) \land\\ &(e.long \,\%2 = 1 \Rightarrow ver(p) = fin(e.palabra) \,\&\, reverso(e.palabra))\end{aligned}$$

reverso(s) $\equiv$ **if** vacia?(s) **then** <> **else** reverso(fin(s)) $\circ$ prim(s) **fi**

Como el cambio se encuentra solo en la estructura de representación, la interfaz en general va a ser igual a la de antes. Lo que sí cambia es el aliasing y la complejidad de las operaciones. Mientras que en la versión anterior habia aliasing y el algoritmo era $O(1)$; en esta ocasión no va a haber aliasing, y la complejidad va a depender del largo de la palabra.

Como los algoritmos lidian con la estructura de representación, debemos redefinir el algoritmo para Ver:

iVer(**in** p : `palindromo`) $\rightarrow res$: secu(α)

```
res ← <>
for i = 0 to (p.long/2) − 1 do
    res ← res • iesimo(p.palabra, i)
end for
for i = long(p.palabra) − 1 downto 0 do
    res ← res • iesimo(p.palabra, i)
end for
```

Ejercicio 3:

¡En ejercicios complicados se vuelve muy importante analizar metódicamente la estructura de representación a la hora de elegir los predicados del invariante!

Una forma de organizarse es empezar mirando cada campo de la estructura por si misma, y **sin mirar las demás**, ver si hay alguna condición que deben cumplir.

- *invitados*: Puede ser cualquier conjunto, no hace falta pedirle nada
- *presentes*: Puede ser cualquier cola, no hace falta pedirle nada
- *grupoDe*: Mirando grupoDe sí hay algo que no debe pasar, ¡una persona no puede estar en más de un grupo!
- *regaloDeGrupo*: De manera similar, dos grupos no pueden traer el mismo regalo (puede verse en la restricción de *lleganInvitados*).
- *grupoMasNumeroso*: Puede ser cualquier grupo.

Entonces, tenemos:

1. La intersección entre cualquier par de conjuntos de personas en *grupoDe* debe ser vacia
2. No puede haber ningún par de grupos en *regaloDeGrupo* cuyo regalo sea el mismo

Una vez analizado esto, podemos continuar tomando campos de a pares y ver que condiciones deben cumplirse para que su información sea consistente:

3. Los *presentes* deben estar incluidos en *invitados*
4. Las personas en *presentes* deben ser exactamente las mismas que en *grupoDe*
5. Las claves de *grupoDe* deben ser las mismas que las de *regaloDeGrupo*
6. *grupoMasNumeroso* es un grupo válido
7. *grupoMasNumeroso* debe ser efectivamente el grupo más numeroso (y de haber varios, debe ser el lexicográficamente mínimo)

Notar que mientras vamos incluyendo y pensando condiciones, nos podemos encontrar con algunos predicados redundantes. Por ejemplo, también podríamos haber querido incluir 'Toda persona en *grupoDe* tiene que estar en *invitados*'. Lo que ocurre es que esto ya está implicado por algunos de los predicados ya incluidos: Si las personas de *grupoDe* son las mismas que las de *presentes* (por (5)), y todo presente está en *invitados* (por (3)), entonces nos queda el predicado que enunciamos antes.

Repasando todo fijamente, no parece faltarnos nada. Así que pasemos todo a lógica de primer orden:

(1) y (2) $(\forall g, g' : grupo)(\{g, g'\} \subset claves(e.grupoDe) \land g \neq g') \Rightarrow_{\text{L}}$
$((obtener(g, e.grupoDe) \cap obtener(g', e.grupoDe) = \emptyset) \land$
$(obtener(g, e.regaloDeGrupo) \neq obtener(g', e.regaloDeGrupo)))$

(3) $(\forall p : persona)\ esta?(e.presentes, p) \Rightarrow p \in e.invitados$

(4) $(\forall p : persona)\ esta?(e.presentes, p) \Leftrightarrow (\exists g : grupo)\ g \in claves(e.grupoDe) \land_{\text{L}} p \in obtener(g, e.grupoDe)$

(5) $claves(e.grupoDe) = claves(e.regaloDeGrupo)$

(6) $e.grupoMasNumeroso \in claves(e.grupoDe)$

(7) $(\forall g : grupo)\ g \in claves(e.grupoDe) \Rightarrow_{\text{L}}$
$(\#obtener(g, e.grupoDe) < \#obtener(e.grupoMasNum, e.grupoDe)) \lor$
$(\#obtener(g, e.grupoDe) = \#obtener(e.grupoMasNum, e.grupoDe) \land e.grupoMasNum \leq_{string} g)$

Donde nuestro invariante de representación es:

$$rep(e) = (5) \wedge_{\text{L}} (1) \wedge (2) \wedge (3) \wedge (4) \wedge (6) \wedge_{\text{L}} (7)$$

Algunos comentarios:

- Observar los *yluego* en el rep. En el predicado (1) asumo que las claves de *e.grupoDe* y *e.regaloDeGrupo* son las mismas. En el (7) asumo que *e.grupoMasNumeroso* está en las claves de *e.grupoDe*
- El hecho de haber combinado los puntos 1 y 2 es simplemente para no repetir el para todo. La claridad es una buena practica a la hora de plantear estos invariantes en lógica de primer orden.

La función de abstracción es la siguiente:

$$
\begin{aligned}
abs(e) = a : altaFiesta \mid & \\
& invitadosPendientes(a) = e.invitados \\
& \wedge\ grupoMasNumeroso(a) = e.grupoMasNumeroso \\
& \wedge\ (\ (\forall r : regalo) \\
& \qquad ((\exists g : grupo)\ g \in claves(e.regaloDeGrupo) \wedge_{\text{L}} r = obtener(g, e.regaloDeGrupo)) \Leftrightarrow r \in regalos(a)\ ) \\
& \wedge\ (\ (\forall r : regalo) \\
& \qquad (\ r \in regalos(a) \Rightarrow \\
& \qquad\quad (\exists g : grupo)\ g \in claves(e.regaloDeGrupo) \wedge_{\text{L}} r = obtener(g, e.regaloDeGrupo)\ \wedge \\
& \qquad\quad personasPorRegalo(a, r) = g\ ))
\end{aligned}
$$

Nos queda implementar la operación *llegaGrupo*:

iLlegaGrupo(**in/out** a: `altaFiesta`, **in** $personas$: `conj(persona)`, **in** $grupo$: `string`, **in** $regalo$: `string`)

$itPersonas \leftarrow CrearIt(personas)$
while $HaySiguiente(itPersonas)$ **do**
  $Encolar(a.presentes, Siguiente(itPersona))$ ▷ Se rompe (4)
  $Avanzar(itPersonas)$
end while
$Definir(a.grupoDe, grupo, Copiar(personas))$ ▷ Se reestablece (4), pero se rompe (5) y quizás (7)
$Definir(a.regaloDeGrupo, grupo, regalo)$ ▷ Se reestablece (5)
$personasNumerosas \leftarrow Obtener(a.grupoMasNumeroso, a.grupoDe)$
if (
  $(\#personas > \#personasNumerosas) \vee$
  $(\#personas = \#personasNumerosas \wedge grupo < e.grupoMasNumeroso)$
) {
  $a.grupoMasNumeroso \leftarrow grupo$ ▷ Se reestablece (7) si se habia roto antes
}