METODOLOGÍA DE LA PROGRAMACIÓN
EJERCICIOS DEL TEMA 6
TRANSFORMACIÓN DE PROGRAMAS RECURSIVOS
Método de Burstall

A) Funciones recursivas elementales (un caso simple y uno recursivo)

1. longitud (longitud de una lista)

Dada una lista de enteros, la función *longitud* calcula el número de elementos de la lista:

```
longitud :: ([Int]) -> Int

Precondición: {true}

longitud([]) = 0
longitud(x : s) = 1 + longitud(s)
```

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

longitud_re([8, 9, 1])

2. ultimo (último elemento de una lista)

Dada una lista no vacía, la función *ultimo* devuelve el último elemento de la lista, es decir, el que está más a la derecha:

```
ultimo :: ([Int]) -> Int

Precondición: {¬es_vacia(s)}

ultimo(s)
      | es_vacia(resto(s))        = primero(s)
      | otherwise                 = ultimo(resto(s))
```

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

ultimo_re([8, 9, 1])

3. inversa (inversa de una lista)

Dada una lista, la función *inversa* calcula la inversa de la lista:

```
inversa :: ([Int]) -> [Int]

Precondición: {true}

inversa([]) = []
inversa(x : s) = inversa(s) ++ [x]
```

Obtener una función iterativa utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

inversa_re([8, 9, 1])

4. basediez (para pasar a base 10)

Dados una base b (donde $2 \leq b \leq 9$) y un número $n \geq 0$ en base b, la función *basediez* calcula el valor de n en base diez:

```
basediez :: (Int, Int) -> Int

Precondición: {2 ≤ b ≤ 9 ∧ n ≥ 0}

basediez(b, n)
  | n ≤ b              = n
  | otherwise          = n mod 10 + basediez(b, n / 10) * b
```

Ejemplos:

basediez(2, 1010) = 10 **(Utilizar este ejemplo para calcular el invariante)**
basediez(2, 110011) = 51
basediez(2, 11011) = 27
basediez(8, 35) = 29

Obtener una función iterativa utilizando el método de Burstall.

5. logaprox (aproximación del logaritmo)

Dados dos enteros $b \geq 2$ y $x \geq 1$, la función *logaprox* calcula la aproximación entera por defecto del logaritmo en base *b* de *x*:

```
logaprox :: (Int, Int) -> Int

Precondición: {b ≥ 2 ∧ x ≥ 1}

logaprox(b, x)
  | x < b              = 0
  | otherwise          = logaprox(b, x / b) + 1
```

Ejemplos:

logaprox(2, 8) = 3
logaprox (2, 11) = 3
logaprox(10,100) = 2
logaprox(10, 503) = 2 **(Utilizar este ejemplo para calcular el invariante)**

Obtener una función iterativa utilizando el método de Burstall.

6. prodesc (producto escalar)

El producto escalar de dos listas de la misma longitud (que representan vectores) de enteros, pongamos s = [s_1, s_2, ..., s_k] y r = [r_1, r_2, ..., r_k], viene dado por:

$$\sum_{i=1}^{k} r_i * s_i$$

Por ejemplo, prodesc([3, 2, 1], [2, 5, 4]) = (3∗2) + (2∗5) + (1∗4) = 20.

```
prodesc :: ([Int], [Int]) -> Int

Precondición: {longitud(s) = longitud(r)}

prodesc (s, r)
  | es_vacia(s)          = 0
  | otherwise            = primero(s) * primero(r) + prodesc (resto(s), resto(r))
```

Obtener una función iterativa utilizando el método de Burstall.

Utilizar el ejemplo de arriba para calcular el invariante.

7. cambiar (sumar 1 en las posiciones pares y restar 1 en las impares)

Sea la siguiente función que dada una lista devuelve la lista que se obtiene decrementando en uno el valor de los elementos de las posiciones impares e incrementando en uno el valor de los elementos de las posiciones pares, con la salvedad de que si la lista tiene un número impar de elementos, el último elemento se deja tal cual:

```
cambiar :: ([Int]) → [Int]

Precondición: {true}

cambiar(s)
  | longitud(s) <= 1      = s
  | longitud(s) > 1       = (primero(s) – 1) : ((primero(resto(s)) +1) : cambiar(resto(resto(s))))
```

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule dicha lista cambiada para una lista dada.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

cambiar_re([3, 1, 8, 9])

8. acumular (lista con suma acumulada de los elementos) (Septiembre 2010)

Sea la siguiente función *acumular* que, dada una lista de enteros, devuelve la lista que contiene en cada posición la suma de los elementos que van desde la primera posición hasta esa posición. Si la lista es vacía devuelve la lista vacía.

```
acumular:: ([Int]) → [Int]
Precondición: {true}

acumular(s)
 | longitud(s) ≤ 1     = s                                                              (#1)
 | otherwise           = [primero(s)] ++ acumular((primero(s) + primero(resto(s))):resto(resto(s)))   (#2)
```

Dada una lista, la función *longitud* devuelve el número de elementos de la lista, la función *primero* devuelve el primer elemento de la lista y la función *resto* devuelve la lista que se obtiene eliminando el primer elemento de la lista.

La precondición viene a decir que se admite como lista de entrada cualquier lista, es decir, que la lista de entrada no ha de cumplir ningún requisito.

Ejemplos para la función *acumular*:

- acumular([10, 8, 15, 4]) = [10, 18, 33, 37]
  En la primera posición la suma acumulada es 10, en la segunda posición es 10 + 8, en la tercera 10 + 8 + 15 y en la cuarta 10 + 8 + 15 + 4.

- acumular([10, 2, 6]) = [10, 12, 18]
  Utilizar este ejemplo para calcular el invariante.
  En la primera posición la suma acumulada es 10, en la segunda posición es 10 + 2 y en la tercera es 10 + 2 + 6.

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule acumular(s) para una lista de enteros s.

B) Funciones recursivas con operación no asociativa

9. sin_ultimo (eliminar último elemento de la lista)

Dada una lista no vacía, la función *sin_ultimo* devuelve la lista que se obtiene eliminando el último elemento:

```
sin_ultimo :: ([Int]) -> [Int]

Precondición: {¬es_vacia(s)}

sin_ultimo(s)
      | es_vacia(resto(s))          = []
      | otherwise                   = primero(s) : sin_ultimo(resto(s))
```

Obtener una función iterativa utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

sin_ultimo_re([5, 8, 1])

C) Funciones con más de un caso simple

10. sublista (decide si una lista es sublista de otra)

Dadas dos listas, la función *sublista* decide si la primera es sublista de la segunda o no.

```
sublista :: ([Int], [Int]) -> Bool

Precondición: {true}

sublista(s, r)
        | longitud(s) > longitud(r)        = False
        | es_prefijo(s, r)                 = True
        | otherwise                        = sublista(s, resto(r))
```

Una lista s es sublista de r si existen otras dos listas h1 y h2 tal que:

r = h1 ++ s ++ h2

Ejemplos:

sublista([4, 8], [2, 7, **4**, **8**, 5]) = True	sublista([], [2, 7, 4, 8, 5]) = True
sublista([2, 7, 4, 8, 5], [4, 8]) = False	sublista([4, 8], []) = False
sublista([4, 8], [2, 4, 7, 8, 5]) = False	sublista([4, 8], [4, 8]) = True

La función *es_prefijo(s, r)* devuelve True si s es sublista de r y está justo al principio. Consideraremos que la función *es_prefijo* ya está definida tanto en Haskell como en ADA*, sólo hay que utilizarla.

Una lista s es prefijo de r si existe otra lista h tal que:

r = s ++ h

Ejemplos:

es_prefijo([4, 8], [2, 7, **4**, **8**, 5]) = False	es_prefijo([], [2, 7, 4, 8, 5])= True
es_prefijo([4, 8], [**4**, **8**, 2, 7, 5]) = True	es_prefijo([4, 8], [])= False
es_prefijo([4, 8], [4, 7, 2, 8, 5])= False	es_prefijo([4, 8], [4, 8]) = True

Se pide:

Obtener una función iterativa para sublista utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

sublista_re([4, 8], [2, 7, **4**, **8**, 5])

11. sufijo (decide si una lista es sufijo de otra)

Dadas dos listas, la función *sufijo* decide si la primera es sufijo de la segunda o no:

```
sufijo :: ([Int], [Int]) -> Bool

Precondición: {true}

sufijo(s, r)
      | longitud(s) < longitud(r)        = False
      | s == r                           = True
      | otherwise                        = sufijo(s, resto(r))
```

Una lista s es sufijo de r si existe otra lista h tal que:

$$r = h \mathbin{+\!\!+} s$$

Ejemplos:

sufijo([4, 8], [2, 7, 5, **4**, **8**]) = True	sufijo([], [2, 7, 4, 8, 5]) = True
sufijo([2, 7, 4, 8, 5], [4, 8]) = False	sufijo([4, 8], []) = False
sufijo([4, 8], [2, 4, 7, 8, 5]) = False	sufijo([4, 8], [4, 8]) = True

Obtener una función iterativa para sufijo utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

sufijo_re([4, 8], [5, **4**, **8**])

12. elimparpos (eliminar el elemento de la posición pos si es par) (Septiembre 2009)

Sea la siguiente función *elimparpos* que dados una lista no vacía de enteros s y un entero pos que representa una posición dentro de la lista s, devuelve la lista que se obtiene al eliminar el elemento de s que ocupa dicha posición en caso de que ese elemento sea par. Si el elemento que ocupa la posición indicada no es par, no se elimina:

```
elimparpos:: ([Int], Int) → [Int]
Precondición: {¬es_vacia(s) ∧ 1 ≤ pos ≤ longitud(s)}

elimparpos(s, pos)
  | pos == 1 && primero(s) mod 2 == 0          = resto(s)
  | pos == 1 && primero(s) mod 2 /= 0          = s
  | pos /= 1                                   = [primero(s)] ++ elimparpos(resto(s), pos – 1)
```

Ejemplos:

elimparpos([8, 5, **9**, 7], **3**) = [8, 5, **9**, 7] Aunque el 9 está en la posición 3 no se elimina porque el 9 es impar.

elimparpos([8, 5, **16**, 7], **3**) = [8, 5, 7] ← **Utilizar este ejemplo para calcular el invariante**

En este segundo ejemplo se elimina el 16 porque está en la posición 3 y el 16 es par.

Utilizando el **método de Burstall**, transformar esa función obteniendo una función iterativa que calcule lo mismo.

D) Funciones con más de un caso recursivo

13. nveces (número de apariciones de un elemento en una lista)

Dados un elemento x y una lista s, la función *nveces* cuenta las apariciones del elemento x en la lista s:

```
nveces :: (Int, [Int]) -> Int

Precondición: {true}

nveces(x, []) = 0
nveces(x, y : s)
  | x /= y                = nveces(x, s)
  | otherwise             = 1 + nveces(x, s)
```

Obtener una función iterativa utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

nveces_re(5, [5, 4, 5, 6])

14. minimo (mínimo de la lista)

Sea la siguiente función que dada una lista no vacía de enteros devuelve el mínimo de la lista:

```
minimo :: [Int] → Int

Precondición: {¬es_vacia(s)}

minimo(s)
  | es_vacia(resto(s))                  = primero(s)
  | primero(s) <= primero(resto(s))     = minimo(primero(s) : resto(resto(s)))
  | primero(s) > primero(resto(s))      = minimo(resto(s))
```

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule el mínimo para una lista no vacía.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

minimo_re([4, 6, 3, 8])

15. bin (calcular representación binaria de un número)

Dado un número natural x (x ≥ 0), la función *bin* calcula el valor en binario de x:

```
bin :: (Int) -> Int
Precondición: {x ≥ 0}

bin(x)
  | x ≤ 1            = x
  | par(x)           = 10 ∗ bin(x / 2)
  | impar(x)         = 10 ∗ bin(x / 2) + 1
```

Ejemplo:

Esta función no calcula la lista de bits correspondiente a x sino que el número decimal que coincide con la secuencia de bits correspondiente a x.

- bin(14) = 1110
  La función *bin* para 14 devuelve el número "mil ciento diez".
- bin(22) = 10110
  La función bin para 22 devuelve el número "diez mil ciento diez".

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

bin_re(14)

16. cifrasim (número de cifras impares de un número)

Sea la siguiente función que, dado un número entero x, tal que x ≥ 0, calcula el número de cifras impares de x:

```
cifrasim :: (Int) -> Int
Precondición: {x ≥ 0}

cifrasim (x)
  | x <= 9                  = x mod 2
  | (x mod 10) mod 2 == 0   = cifrasim (x div 10)
  | (x mod 10) mod 2 ≠ 0    = 1 + cifrasim (x div 10)

donde div es la división entera
```

Ejemplos:

cifrasim(45881) = 2
cifrasim(6800) = 0

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule el número de cifras impares para un valor x dado.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

cifrasim_re(4538)

17. mcd3 (máximo común divisor de tres números)

Dados tres números enteros positivos a, b y c (a ≥ 1, b ≥ 1, c ≥ 1), la función *mcd3* calcula el máximo común divisor de a, b y c:

```
mcd3 :: (Int, Int, Int) → Int
Precondición: { a ≥ 1 ∧ b ≥ 1 ∧ c ≥ 1}

mcd3(a, b, c)
   | a == b && b == c        = a
   | a – c ≥ b               = mcd3(a – c, b, c)
   | a – c ≥ c               = mcd3(b, a – c, c)
   | a – c ≥ 1               = mcd3(b, c, a – c)
   | a – c ≤ 0               = mcd3(b, c, c – a)
```

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

mcd3_re(18, 12, 24)

18. barrer (eliminar los que son menores que el primero y poner el 1º al final) (junio 2008)

Sea la siguiente función que, dada una lista de enteros, devuelve la lista que se obtiene eliminando todos los elementos de la lista que son menores que el primero y que coloca dicho entero al final de la nueva lista:

La idea es que el primer elemento atraviesa toda la lista haciendo desaparecer los elementos que son menores que él y finalmente él mismo se queda en la última posición.

```
barrer :: ([Int]) → ([Int])
Precondición ≡ {true}

  barrer(s)
     |longitud(s) ≤ 1    = s
     |primero(s) > primero(resto(s))   = barrer(primero(s):resto(resto(s)))
     |otherwise          = [primero(resto(s))] ++ barrer(primero(s):resto(resto(s)))
```

Ejemplos:

barrer([5, 8, 8, 4, 9, 3]) = [8, 8, 9, 5]
barrer([5, 8, 5, 4, 9, 3]) = [8, 5, 9, 5]
barrer([5, 8, 8, 9, 6]) = [8, 8, 9, 6, 5]

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule *barrer(s)* para una lista de enteros s.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

barrer_re([6, 4, 8, 7])

19. dividir (dividir los que son divisibles por el primero y mantener los demás igual) (septiembre 2008)

Sea la siguiente función que, dada una lista no vacía de enteros, devuelve una lista que tiene un elemento menos que el de entrada y se obtiene considerando los elementos que van a partir del segundo:

b) Aquellos elementos divisibles por el primero (los que dan resto 0) son sustituidos por el resultado que se obtiene al dividirlos por el primero.
c) Aquellos elementos no divisibles por el primero se mantienen.

Si la lista contiene sólo un elemento, devuelve la lista vacía.

Ejemplos:

dividir([**2**, 8, 6, 5, 9, 20]) = [4, 3, 5, 9, 10]	dividir([**3**, 15]) = [5]
dividir([**5**, 8, 8, 15, 9]) = [8, 8, 3, 9]	dividir([**3**, 14]) = [14]

La idea es que el primer elemento va atravesando la lista y dividiendo aquellos que son divisibles y manteniendo igual los que no son divisibles. El primer número no aparecerá en la lista resultante.

```
dividir :: ([Int]) → [Int]
Precondición ≡ {¬es_vacia(s)}

dividir(s)
  |longitud(s) == 1                          = []
  |primero(resto(s)) mod primero(s) == 0     = [primero(resto(s)) div primero(s)] ++ dividir(primero(s):resto(resto(s)))
  |otherwise                                 = [primero(resto(s))] ++ dividir(primero(s):resto(resto(s)))
```

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule dividir(s) para una lista no vacía de enteros s.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

dividir_re([5, 8, 15, 9])

Recordemos que **mod** calcula el resto de la división entera y **div** calcula la división entera.

20. suig (longitudes de las sublistas formadas por valores iguales) (Junio 2009)

Sea la siguiente función que, dados un valor entero c y una lista no vacía de enteros s, devuelve la lista que contiene el número de componentes de cada sublista formada por valores iguales. La única excepción es el caso de la primera sublista, para la cual la nueva lista contiene la longitud de esa primera sublista más c:

suig :: (Int, [Int]) -> [Int]
Precondición: {¬es_vacia(s)}

suig(c, s**)**
| longitud(s) == 1 = [c + 1]
| primero(s) == primero(resto(s)) = **suig(**c + 1, resto(s)**)**
| primero(s) /= primero(resto(s)) = [c + 1] ++ **suig(0,** resto(s)**)**

Ejemplos:
suig(0, [8, 8, 8, 8, 5, 9, 9, 8, 8]) = [4, 1, 2, 2] suig(0, [8]) = [1]
suig(3, [8, 8, 8, 8, 5, 9, 9, 8, 8]) = [7, 1, 2, 2] suig(3, [8]) = [4]
suig(0, [8, 8, 5, 9, 9]) = [2, 1, 2] ← **Utilizar este ejemplo para calcular el invariante**

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule suig(c, s) para un entero c y una lista de enteros s.

21. ult_primo (devolver último primo de la lista) (Junio 2010)

Sea la siguiente función *ult_primo* que dada una lista de enteros que contiene al menos un número primo, devuelve el último primo de la lista, es decir, el que está más a la derecha.

ult_primo:: ([Int]) → Int
Precondición: {∃ k (1 ≤ k ≤ longitud(s) ∧ es_primo(elem(k, s)))}

ult_primo(s)

| es_vacia(resto(s))	= primero(s)	(#1)
| es_primo(primero(resto(s)))	= ult_primo(resto(s))	(#2)
| not es_primo(primero(resto(s)))	= ult_primo(primero(s):resto(resto(s)))	(#3)

Funciones auxiliares:

- Dados un elemento y una lista, la función *esta* devuelve true si ese elemento está en esa lista y devuelve false en caso contrario.
- Dada una lista, la función *primero* devuelve el primer elemento de la lista
- Dada una lista, la función *resto* devuelve la lista que se obtiene eliminando el primer elemento de la lista.
- Dada una lista, la función *longitud* devuelve el número de elementos de la lista.
- La función *es_primo*, dado un entero devuelve true si es primo y false en caso contrario, es decir, dado un entero devuelve true si el entero es positivo y tiene justo dos divisores (2, 3, 5, 7, 11, 13, 17, … son números primos).
- Dados un número entero y una lista, la función *elem* devuelve el elemento de la lista que ocupa la posición indicada por el número (siempre que el número sea adecuado).

La precondición viene a decir que la lista de entrada s contiene al menos un número primo y por tanto sabemos que no es vacía.

Ejemplos para la función ult_primo:

- ult_primo([8, 11, -7, 2, **5**, 6]) = 5
  En esta lista hay tres números primos: 11, 2 y 5. La función ha de devolver el último, es decir, 5.
- ult_primo([2, -7, **2**, 4, 10]) = 2
  En esta lista hay dos números primos: 2 y 2. La función ha de devolver el último, es decir, 2.
- ult_primo([8, 2, **11**, 6]) = 11
  En esta lista hay dos números primos: 2 y 11. La función ha de devolver el último, es decir, 11. ← **Utilizar este ejemplo para calcular el invariante**

Utilizando el **método de Burstall**, diseñar un algoritmo iterativo que calcule ult_primo(s) para una lista de enteros s que contiene al menos un número primo.

Funciones recursivas que son mezcla de los casos B) y D)

22. princ (poner todas las apariciones de x al principio)

La siguiente función, dados un elemento x y una lista, calcula la lista que se obtiene poniendo todas las apariciones de x al principio de la lista:

```
princ :: (Int, [Int]) -> [Int]
Precondición ≡ {true}

princ(x, []) = []
princ(x, z : s)
   | z == x                    = z : princ(x, s)
   | otherwise                 = princ(x, s) ++ [z]
```

Obtener una función iterativa utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

princ_re(8, [7, 8, 2, 8])

23. mayores (elimina de una lista los elementos que sean mayores que x)

Sea la siguiente función que, dados un número entero y una lista de enteros, devuelve la lista que se obtiene al eliminar de la lista inicial los elementos que son mayores que dicho número:

```
mayores :: (Int, [Int]) -> [Int]
Precondición ≡ {true}

mayores(x, []) = []
mayores(x, z : s)
   | z > x             = mayores(x, s)
   | z <= x            = z : mayores (x, s)
```

Ejemplos:

mayores(3, [5, 2, 1, 8]) = [2, 1]
mayores(6, [5, 2, 1, 2]) = [5, 2, 1, 2]

Utilizando el **método de Burstall**, diseñar una función iterativa que calcule la lista que se obtiene al eliminar de una lista h los elementos mayores que x.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

mayores_re(5, [7, 3, 2, 8])

24. f (cada elemento sustituye a los siguientes hasta encontrar uno mayor) (junio 2007)

Sea la siguiente función que, dada una lista no vacía de enteros, devuelve la lista que se obtiene empezando desde el primer elemento y haciendo que un elemento sustituya a los que vienen a continuación hasta encontrar uno mayor. Cuando se encuentre uno mayor ese número mayor será el que empiece a sustituir a los que vienen a continuación y así hasta recorrer toda la lista:

```
f :: ([Int]) -> [Int]
Precondición: {¬es_vacia(s)}

f(s)
  | es_vacia(resto(s))                 = s
  | primero(s) ≤ primero(resto(s))     = primero(s) : f(resto(s))
  | primero(s) > primero(resto(s))     = primero(s) : f(primero(s) : resto(resto(s)))
```

Ejemplos:

f([4, 2, 3, 8, 9, 4]) = ([4, 4, 4, 8, 9, 9])
f([4, 2, 8, 9, 4]) = ([4, 4, 8, 9, 9]) **(Utilizar este ejemplo al calcular el invariante)**
f([3, 9, 9, 3]) = ([3, 9, 9, 9])
f([3, 8, 8, 9]) = ([3, 8, 8, 9])

Utilizando el **método de Burstall**, diseñar una función iterativa que calcule f(s) para una lista no vacía de enteros s.

25. eco (eliminar repeticiones contiguas) (Septiembre 2007)

Sea la siguiente función que, dada una lista no vacía de enteros, devuelve la lista que se obtiene dejando solo una copia cuando un elemento aparece repetido en posiciones contiguas, es decir, se eliminan las repeticiones contiguas:

```
eco :: ([Int]) -> [Int]
Precondición: {¬es_vacia(s)}

eco(s)
  | es_vacia(resto(s))                 = s
  | primero(s) = primero(resto(s))     = eco(resto(s))
  | primero(s) /= primero(resto(s))    = primero(s) : eco(resto(s))
```

Ejemplos:

eco([0, 8, 8, 8, 4, 0, 0]) = ([0, 8, 4, 0])
eco([0, 8, 8, 0, 0]) = ([0, 8, 0]) **(Utilizar este ejemplo al calcular el invariante)**
eco([3, 9, 9, 3]) = ([3, 9, 3])
eco([5, 2, 6]) = ([5, 2, 6])

Utilizando el **método de Burstall**, diseñar una función iterativa que calcule *eco(s)* para una lista no vacía de enteros s.

Funciones recursivas que son mezcla de los casos C) y D)

26. mcd (máximo común divisor de dos números)

Dados dos números enteros positivos a y b (a ≥ 1, b ≥ 1), la función *mcd* calcula el máximo común divisor de a y b:

```
mcd :: (Int, Int) -> Int
Precondición: {a ≥ 1 ∧ b ≥ 1}

mcd(a, b)
   | a == 1 || b == 1        = 1
   | a == b                  = a
   | a > b                   = mcd(a – b, b)
   | a < b                   = mcd(a, b – a)
```

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

mcd_re(12, 18)

27. prod (producto de dos números)

Dados dos números naturales x e y (x ≥ 0, y ≥ 0), la función *prod* calcula el producto de x e y:

```
prod :: (Int, Int) -> Int
Precondición: {x ≥ 1 ∧ y ≥ 1}

prod(x, y)
   | x = 0 || y = 0          = 0
   | x = 1                   = y
   | par(x)          = prod(x / 2, 2 ∗ y)
   | impar(x)        = prod(x / 2, 2 ∗ y) + y
```

Transformar la función en iterativa usando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

prod_re(29, 3)

28. factores (factores primos de x que sean mayores o iguales que m)

Dados dos números enteros x y m ($x \geq 1$, $m \geq 2$), la función *factores* obtiene la lista de todos los factores primos (con repetición) de x que son mayores o iguales que m:

```
factores :: (Int, Int) -> [Int]
Precondición: {x ≥ 1 ∧ m ≥ 2}

factores(x, m)
  | x < m                              = []
  | primo(x)                           = x : []
  | not primo(m) || x mod m /= 0       = factores(x, m + 1)
  | primo(m) && x mod m = 0            = m : factores(x / m, m)
```

Ejemplos:

factores(15, 2) = [3, 5]
factores(15, 7) = []
factores(8, 2) = [2, 2, 2]
factores(8, 4) = []
factores(33, 2) = [3, 11]
factores(33, 3) = [3, 11]
factores(33, 4) = [11]
factores(7, 2) = [7]
factores(12, 2) = [2, 2, 3]
factores(12, 3) = [3]
factores(12, 16) = []
factores(45, 2) = [3, 3, 5] **(Utilizar este ejemplo al calcular el invariante)**

Obtener una función iterativa utilizando el método de Burstall.

29. apar (número de veces que el dígito d aparece en el número n)

Dados un dígito *d* (donde $0 \leq d \leq 9$) y un número entero *n* (tal que $n \geq 0$), la función *apar* calcula el número de veces que aparece el dígito d en el número n:

```
apar :: (Int, Int) -> Int

Precondición: {0 ≤ d ≤ 9 ∧ n ≥ 0}

apar(d, n)
  | d = n                  = 1
  | 0 <= n <= 9            = 0
  | n mod 10 == d          = apar(d, n / 10) + 1
  | n mod 10 /= d          = apar(d, n / 10)
```

Obtener una función iterativa utilizando el método de Burstall.

Para calcular el invariante hay que desarrollar el siguiente ejemplo:

apar_re(8, 8584)