



MEMORIA DE EJERCICIOS

Modelos Avanzados de Computación

Universidad de Huelva

***Grado en Ingeniería Informática
Especialidad en Computación
Curso 2022/23
Manuel Ramírez Ballesteros***

Índice:

- 1- *Introducción***
- 2- *Ejercicio 1: nsobrek***
- 3- *Ejercicio 2: raices***
- 4- *Ejercicio 3: fibonacci***
- 5- *Ejercicio 4: pertenece***
- 6- *Ejercicio 46: porcentajeRebotantes***
- 7- *Ejercicio 612: cantidadAmigos***
- 8- *Conclusión***

1- Introducción

En esta memoria se justificarán las soluciones propuestas para los cuatro ejercicios comunes y se resolverán otros dos ejercicios seleccionados de la web *Project Euler*, aplicando las nuevas funciones y estructuras vistas en las sesiones de prácticas. Además, se plantearán diversas versiones para cada función. Los ejercicios a resolver son:

- `nsobrek(n,k)`: Función que calcula el número combinatorio n sobre k , que es el número de combinaciones de n elementos tomados de k en k .
- `raices(a,b,c)`: Función que calcula las soluciones de una ecuación de segundo grado de la forma $y = a \cdot x^2 + b \cdot x + c$, con coeficientes a , b y c .
- `fibonacci(n)`: Función que devuelve el elemento n -ésimo de la sucesión de Fibonacci, donde cada elemento se obtiene a partir de la suma de los dos anteriores.
- `pertenece(a,b)`: Función que comprobará la pertenencia de un elemento a en una lista b de manera recursiva.

Los dos ejercicios de libre elección que se han planteado son:

- `porcentajeRebotantes(n)`: Función con la que se puede comprobar el porcentaje de números que son rebotantes hasta n . Se corresponde al problema 112 de la web.
- `cantidadAmigos(n)`: Función que calcula la cantidad de pares que son números amigos entre 1 y n , considerando que dos números son amigos si su representación en base 10 tiene al menos un dígito común. Se corresponde al problema 612 de la web.

2- Ejercicio 1: *nsobre n k*

La función *nsobre k n* se ha implementado en base a la definición del número combinatorio *n* sobre *k*:

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

Para ello, se ha requerido de la implementación de otra función *factorial n*, con la cual obtenemos el factorial de un número *n*. Se han propuesto tres implementaciones distintas de la función factorial *n*:

```
factorialConIf :: (Num a, Ord a) => a -> a
factorialConIf n =
    if n < 0 then error "No se calculan factoriales negativos"
    else if n == 0 then 1
    else n*factorialConIf(n-1)

factorialConGuardas :: (Num a, Ord a) => a -> a
factorialConGuardas n
    | n == 0 = 1
    | n < 0 = error "No se calculan factoriales negativos"
    | otherwise = n*factorialConGuardas(n-1)

factorialConCase :: (Num a, Ord a) => a -> a
factorialConCase n = case n of
    0 -> 1
  - | n < 0 -> error "No se calculan factoriales negativos"
    | otherwise -> n*factorialConCase (n-1)
```

Podemos ver a continuación algunas pruebas del funcionamiento de la función:

```
Main> factorialConIf 0
1
Main> factorialConIf 2
2
Main> factorialConIf 5
120
Main> factorialConIf 10
3628800
Main> factorialConIf (-1)
Program error: No se calculan factoriales negativos
```

```

Main> factorialConCase 0
1
Main> factorialConCase 3
6
Main> factorialConCase 10
3628800
Main> factorialConCase (-10)

Program error: No se calculan factoriales negativos

```

```

Main> factorialConGuardas 0
1
Main> factorialConGuardas 3
6
Main> factorialConGuardas 10
3628800
Main> factorialConGuardas (-1)

Program error: No se calculan factoriales negativos

```

Donde cabe destacar que es necesario los paréntesis cuando se desea ejecutar la función para un número negativo, sino aparece el siguiente error de inferencia debido a que Haskell entiende el signo menos como una función:

```

Main> factorialConIf -3
ERROR - Cannot infer instance
*** Instance   : Num (a -> a)
*** Expression : factorialConIf - 3

```

Empleando la función anterior, se ha implementado la función *nsobrek n k* de la siguiente forma:

```

nsobrek :: Integral a => a -> a -> a
nsobrek n k = div (factorialConIf n) (factorialConGuardas k * factorialConCase(n-k))

nsobrekConWhere :: Integral a => a -> a -> a
nsobrekConWhere n k = combinatorio
    where combinatorio = div (factorialConGuardas n) (factorialConIf k * factorialConCase(n-k))

```

Se podría haber hecho otra implementación donde en el *where* se definiesen los factoriales empleados en la definición del número combinatorio, pero se ha optado por la definición anterior utilizando las distintas funciones del factorial implementadas. Algunos ejemplos de ejecución son:

```

Main> nsobrek 4 1
4
Main> nsobrek 8 3
56
Main> nsobrek 8 (-3)

Program error: No se calculan factoriales negativos

Main> nsobrekConWhere 1 4

Program error: No se calculan factoriales negativos

Main> nsobrekConWhere 4 1
4
Main> nsobrekConWhere 8 3
56
Main> nsobrekConWhere 8 (-3)

Program error: No se calculan factoriales negativos

```

3- Ejercicio 2: raices a b c

La función *raices a b c* se ha implementado siguiendo la fórmula para obtener las soluciones reales de una ecuación de segundo grado:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Para ello se ha implementado una función que nos devuelve el valor del discriminante (aunque también podía incluirse definido en un where):

```

discriminante :: Floating a => a -> a -> a -> a
discriminante a b c = b**2 - 4*a*c

```

Cuya ejecución podemos comprobar a continuación:

```

Main> discriminante 3 2 1
-8.0
Main> discriminante 2 2 2
-12.0
Main> discriminante 3 6 2
12.0
Main> discriminante 2 2 1
-4.0
Main> discriminante 1 2 1
0.0

```

Y se han propuesto dos implementaciones de la función raices a b c, considerando en ambas los dos errores que podemos encontrar: Que el discriminante sea menor que cero, ya que nos generaría soluciones imaginarias, y que el valor de a sea cero, produciendo una división por cero:

```
raicesConGuardas :: (Ord a, Floating a) => a -> a -> a -> [a]
raicesConGuardas a b c
  | a == 0 = error "Division por cero"
  | discriminante a b c < 0 = error "Solucion no real : Discriminante negativo"
  | otherwise = [positivo, negativo]
    where positivo = (-b + sqrt(discriminante a b c)) / (2*a)
          negativo = (-b - sqrt(discriminante a b c)) / (2*a)

raicesConIf :: (Ord a, Floating a) => a -> a -> a -> [a]
raicesConIf a b c =
  if a == 0 then error "Division por cero"
  else if discriminante a b c < 0 then error "Solucion no real : Discriminante negativo"
  else [(-b + sqrt (discriminante a b c)) / (2*a), (-b - sqrt (discriminante a b c)) / (2*a)]
```

Podemos comprobar la ejecución de ambas funciones a continuación:

```
Main> raicesConGuardas 1 3 (-1)
[0.302775637731995,-3.30277563773199]
Main> raicesConGuardas 3 2 6

Program error: Solucion no real : Discriminante negativo

Main> raicesConGuardas 3 6 2
[-0.422649730810374,-1.57735026918963]
Main> raicesConGuardas 0 6 2

Program error: Division por cero

Main> raicesConGuardas 1 2 1
[-1.0,-1.0]

Main> raicesConIf 1 3 (-1)
[0.302775637731995,-3.30277563773199]
Main> raicesConIf 3 2 6

Program error: Solucion no real : Discriminante negativo

Main> raicesConIf 3 6 2
[-0.422649730810374,-1.57735026918963]
Main> raicesConIf 1 2 1
[-1.0,-1.0]
Main> raicesConIf 1 3 1
[-0.381966011250105,-2.61803398874989]
```

4- Ejercicio 3: fibonacci *n*

Esta función, que nos devuelve el *n*-ésimo valor de la sucesión de Fibonacci, resultado de sumar los dos elementos anteriores de la sucesión, se ha planteado de tres formas distintas: Una empleando if – else, otra empleando guardas y case y otra que no emplea ninguna de estas estructuras, con el inconveniente de que no se ha podido controlar los errores en caso de poner un valor negativo de *n*:

```
fibonacci :: (Num a, Num b) => b -> a
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)

fibonacciConCaseYGuardas :: (Num a, Num b, Ord b) => b -> a
fibonacciConCaseYGuardas x = case x of
    0 -> 1
    1 -> 1
    _ | x < 0 -> error "La funcion solo admite numeros positivos"
    _ | otherwise -> fibonacciConCaseYGuardas(x-1) + fibonacciConCaseYGuardas(x-2)

fibonacciConIf :: (Num a, Num b, Ord b) => b -> a
fibonacciConIf x =
    if x == 1 || x == 0 then 1
    else if x < 0 then error "La funcion solo admite numeros positivos"
    else fibonacciConIf(x-1) + fibonacciConIf(x-2)
```

Podemos comprobar la ejecución de todas ellas a continuación:

```
Main> fibonacciConIf (-10)
Program error: La funcion solo admite numeros positivos

Main> fibonacciConIf 1
1
Main> fibonacciConIf 10
89

Main> fibonacciConCaseYGuardas 0
1
Main> fibonacciConCaseYGuardas 4
5
Main> fibonacciConCaseYGuardas 10
89
Main> fibonacciConCaseYGuardas (-10)
Program error: La funcion solo admite numeros positivos
```

```
Main> fibonacci 0
1
Main> fibonacci 1
1
Main> fibonacci 5
8
Main> fibonacci 10
89
```

5- Ejercicio 4: pertenece a b

Esta función se ha planteado de tres formas distintas: Una empleando if – else, otra empleando guardas y case y otra que no emplea ninguna de estas estructuras:

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece _ [] = False
pertenece a (c:r) = a == c || pertenece a r

perteneceConIf :: Eq a => a -> [a] -> Bool
perteneceConIf _ [] = False
perteneceConIf a (c : r) =
    if a == c then True
    else perteneceConIf a r

perteneceConCaseYGuardas :: Eq a => a -> [a] -> Bool
perteneceConCaseYGuardas a b = case b of
    [] -> False
    (c:r) | c == a -> True
          | otherwise -> perteneceConCaseYGuardas a r
```

Para la implementación sin estructuras condicionales y la implementación con if – else se ha considerado que, si la lista está vacía, el resultado es falso independientemente del valor de *a*, al tener que definir la lista *b* como (c:r), cosa que no ha sido necesaria en la función con guardas y case. Para el resto de casos, se comprueba si el elemento *a* es igual a la cabeza de la lista, resultando verdadero en caso afirmativo, o la pertenencia al resto de la lista mediante una llamada recursiva a sí misma. Podemos comprobar la ejecución de dichas funciones a continuación:

```
Main> pertenece 1 []
False
Main> pertenece 1 [2,3]
False
Main> pertenece 1 [2,1,3]
True
Main> pertenece 1 [2,2,3,3,1]
True

Main> perteneceConIf 1 []
False
Main> perteneceConIf 1 [2,3]
False
Main> perteneceConIf 1 [2,1,3]
True
Main> perteneceConIf 1 [2,2,3,2,1]
True
Main> perteneceConIf 1 [1,2,3,2]
True

Main> perteneceConCaseYGuardas 1 []
False
Main> perteneceConCaseYGuardas 1 [2,3]
False
Main> perteneceConCaseYGuardas 1 [2,3,1]
True
Main> perteneceConCaseYGuardas 1 [2,1,3,2]
True
```

6- Ejercicio 112: porcentajeRebotantes *n*

Se considera un número rebotante aquel que sus cifras no son puramente crecientes ni decrecientes, es decir, 12345 o 54321 no serían números rebotantes, pero 1234321 sí. Lógicamente, no existen números rebotantes menores que 100, pero a partir de aquí el porcentaje de estos números comienza a dispararse, alcanzando el 50% en el 538 y el 90% en el 21780. En este problema se pide encontrar el número para el cual el porcentaje es del 99%.

Para resolver esto en un tiempo razonable, se han planteado diversas funciones:

- *digitos n*: Función que convierte un número *n* en una lista con todos sus dígitos para poder compararlos individualmente. Para implementarla se ha considerado que, si el número es menor que 10, se devuelva una lista con dicha cifra y, en caso contrario, se realizará una concatenación recursiva del resto de dividir *n* entre 10 a los cocientes de la división hasta que quede un número de una única cifra (la primera cifra de *n*) que se concatenará como la cabeza de la lista.

```
digitos :: Int -> [Int]
digitos n =
    if n < 10 then [n]
    else digitos (div n 10) ++ [mod n 10]
```

- *comparaCreciente a*: Función que devuelve *true* si los elementos de una lista *a* están ordenados de forma creciente. Para ello, si la lista tiene un único elemento, se considera que está ordenada crecientemente y si no, comprobaremos el primer elemento es menor o igual que el segundo mientras comprobamos recursivamente que el resto de elementos también son mayores o iguales que los primeros.

```
comparaCreciente [_] = True
comparaCreciente (c1:c2:r) = c1 <= c2 && comparaCreciente (c2:r)
```

- *comparaDecreciente a*: Función que devuelve *true* si los elementos de una lista *a* están ordenados de forma decreciente. Para ello, si la lista tiene un único elemento, se considera que está ordenada decrecientemente y si no, comprobaremos el primer elemento es mayor o igual que el segundo mientras comprobamos recursivamente que el resto de elementos también son menores o iguales que los primeros.

```
comparaDecreciente [_] = True
comparaDecreciente (c1:c2:r) = c1 >= c2 && comparaDecreciente (c2:r)
```

- *esRebotante n*: Función que devuelve *true* si *n* es un número rebotante. Para ello se ha considerado que si un número tiene menos de tres cifras, no es rebotante. En caso contrario, comprobaremos si es creciente o decreciente y devolveremos la negación. Esta función se ha implementado tanto con *if* – *else* como con guardas.

```

esRebotanteIf n =
  if length(digitos n) < 3 then False
  else not (comparaCreciente (digitos n) || comparaDecreciente (digitos n))

esRebotanteGuardas n
  | length (digitos n) < 3 = False
  | otherwise = not (comparaCreciente (digitos n) || comparaDecreciente (digitos n))

```

- porcentajeRebotantes n : Función que devuelve el porcentaje de números rebotantes que hay hasta n . Para ellos, si n es menor que cien, el porcentaje debe ser 0, si no obtendremos el porcentaje donde la cantidad de números rebotantes hasta n será la longitud de la lista resultante al filtrar los números rebotantes entre 1 y n .

```

porcentajeRebotantes n = if n < 100 then 0
  else (fromIntegral rebotantes/fromIntegral n) * 100
  where rebotantes = length(filter esRebotanteIf [1..n])

porcentajeRebotantesGuardas n
  | n < 100 = 0
  | otherwise = (fromIntegral rebotantes/fromIntegral n) * 100
  where rebotantes = length(filter esRebotanteGuardas [1..n])

```

Tras algunos intentos estimando el número que consigue el 99% de números rebotantes, obtenemos que se trata del 1587000!

A continuación, se muestran algunos ejemplos de la ejecución de las distintas funciones implementadas y de cómo se ha logrado obtener dicho valor:

```

Main> digitos 1
[1]
Main> digitos 12
[1,2]
Main> digitos 112234322
[1,1,2,2,3,4,3,2,2]
Main> digitos 1587000
[1,5,8,7,0,0,0]

Main> comparaCreciente [1,2,3,4,5]
True
Main> comparaCreciente [1,2,3,4,5,1]
False
Main> comparaCreciente [1,5,8,7,0,0,0]
False

Main> comparaDecreciente [1,2,3,4,5]
False
Main> comparaDecreciente [5,4,3,2,1]
True
Main> comparaDecreciente [1,5,8,7,0,0,0]
False

```

<pre> Main> esRebotanteIf 127346 True Main> esRebotanteIf 12234556 False Main> esRebotanteIf 987766653 False Main> esRebotanteIf 1587000 True Main> porcentajeRebotantes 1 0.0 Main> porcentajeRebotantes 100 0.0 Main> porcentajeRebotantes 101 0.99009900990099 Main> porcentajeRebotantes 538 50.0 Main> porcentajeRebotantes 53800 94.4405204460967 Main> porcentajeRebotantes 538000 98.5117100371747 Main> porcentajeRebotantes 598000 98.607525083612 Main> porcentajeRebotantes 708000 98.7505649717514 Main> porcentajeRebotantes 808000 98.804702970297 Main> porcentajeRebotantes 1000000 98.7048 Main> porcentajeRebotantes 2000000 99.20195 </pre>	<pre> Main> esRebotanteGuardas 127346 True Main> esRebotanteGuardas 12234556 False Main> esRebotanteGuardas 987766653 False Main> esRebotanteGuardas 1587000 True Main> porcentajeRebotantes 1576000 98.9939720812183 Main> porcentajeRebotantes 1578000 98.9946134347275 Main> porcentajeRebotantes 1580000 98.9955696202532 Main> porcentajeRebotantes 1590000 99.001572327044 Main> porcentajeRebotantes 1589000 99.0010069225928 Main> porcentajeRebotantes 1588000 99.0006297229219 Main> porcentajeRebotantes 1586000 98.999369482976 Main> porcentajeRebotantes 1586800 98.9998739601714 Main> porcentajeRebotantes 1587000 99.0 Main> porcentajeRebotantes 1587001 99.0000006301193 Main> porcentajeRebotantes 1586999 98.9999993698799 </pre>
---	--

7- Ejercicio 612: cantidadAmigos n

Se considera que dos números son amigos si al representarlos en base 10 tienen algún dígito en común. Por ejemplo, 1234 y 2578 son números amigos. Para este problema se pide diseñar una función que, dado un valor n , nos diga la cantidad de pares (p, q) que son amigos tal que $1 \leq p < q < n$. Se sabe que para $n = 100$, la cantidad de pares amigos es 1539. Para resolver esto se han planteado diversas funciones:

- *digitos n*: Función que convierte un número n en una lista con todos sus dígitos para poder compararlos individualmente. Es la misma implementación que en el ejercicio anterior.
- *sonAmigos a b*: Función que comprueba si dos números a y b son amigos. Para ello se verifica si algún elemento de la lista *digitos b* está presente en *digitos a*, dando como resultado *false* si todos los dígitos son diferentes y *true* si al menos uno está contenido en a .

```
sonAmigos :: Int -> Int -> Bool
sonAmigos a b = any (`elem` (digitos a)) (digitos b)
```

- pares n : Función que genera todos los pares de números (p,q) tal que: $1 \leq p < q < n$. En caso de que n sea menor que dos, se dará un mensaje de error informando de que la cantidad de pares es insuficiente. En caso contrario, se generarán todos los pares que cumplan dicha restricción. Se ha planteado esta función de dos formas diferentes: Una empleando `if – else` y otra con guardas. Puesto que no podemos emplear listas por comprensión, lo cual nos simplificaría mucho este problema, se ha planteado una alternativa recursiva donde se generan los pares a partir del 1, como primer elemento p del par, que iremos incrementando hasta $n-1$, donde devolveremos una lista vacía al igualarlo como condición de parada. Mientras no se iguale dicho valor, generaremos pares incrementando en uno el valor de q hasta que iguale el valor n , de manera que, si no lo ha igualado, se añadirá el par (p,q) a la solución y seguiremos generando los pares incrementando q . Si se llega a que $q = n$, se generarán más pares esta vez incrementando p .

```
paresIf :: (Ord a, Num a, Enum a) => a -> [(a,a)]
paresIf n =
    if n <= 2 then error "No hay pares suficientes"
    else [(p, q) | p <- [1..n-1], q <- [p+1..n-1]]

paresGuardas :: (Num a, Ord a) => a -> [(a,a)]
paresGuardas n
    | n <= 2 = error "No hay pares suficientes"
    | otherwise = generarPares 1
    where
        generarPares a
            | a >= n-1 = []
            | otherwise = generarPares2 a (a+1)
        generarPares2 a b
            | b >= n = generarPares (a+1)
            | otherwise = (a,b) : generarPares2 a (b+1)

paresIf2 n =
    if n <= 2 then error "No hay pares suficientes"
    else generarPares 1
    where
        generarPares a =
            if a >= n-1 then []
            else generarPares2 a (a+1)
        generarPares2 a b =
            if b >= n then generarPares (a+1)
            else (a,b) : generarPares2 a (b+1)
```

- cantidadAmigos n : Función que calcula la cantidad de pares de números amigos que hay hasta n . Para ello, se devolverá la longitud de filtrar los valores p, q tal que p y q son números amigos dentro de la lista de pares

generada por la función anterior.

```
cantidadAmigos :: Int -> Int
cantidadAmigos n = length (filter (\(p, q) -> sonAmigos p q) (paresIf n))
```

Pese a lograr una implementación correcta, ya que podemos verificar que `cantidadAmigos 100 = 1539`, no es posible resolverlo para 10^{18} como nos pedían en la web, ya que sale el error: Arithmetic overflow. Podemos comprobar la ejecución de dichas funciones a continuación:

```
Main> sonAmigos 123 345
True
Main> sonAmigos 123 445
False
Main> sonAmigos 113 4451
True

Main> paresIf 2

Program error: No hay pares suficientes

Main> paresIf 4
[(1,2), (1,3), (2,3)]
Main> paresIf 6
[(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)]
Main> paresIf2 2

Program error: No hay pares suficientes

Main> paresIf2 4
[(1,2), (1,3), (2,3)]
Main> paresIf2 6
[(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)]
Main> paresGuardas 2

Program error: No hay pares suficientes

Main> paresGuardas 4
[(1,2), (1,3), (2,3)]
Main> paresGuardas 6
[(1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5), (4,5)]

Main> cantidadAmigos 100
1539
Main> cantidadAmigos 10
0
Main> cantidadAmigos 11
1
Main> cantidadAmigos 120
2461
Main> cantidadAmigos 12
3
```

8- Conclusión

Estos ejercicios han servido para profundizar más en el uso de las estructuras y funciones de Haskell, resolviendo diversos problemas de distintas formas. Aunque aún no disponemos de todas las funcionalidades que nos ofrece este lenguaje, como se ha tratado de ejemplificar, ya podemos resolver problemas más complejos de manera más eficiente. Además, se incluye un problema extra que me ha resultado curioso, que es la contradicción de una de las conjeturas de Goldbach (problema 46 de la web). Al ejecutarlo, se irán mostrando los números que contradicen la conjetura y, si se tiene la suficiente paciencia, aparecerá el número 5777, que aparte de incumplir la conjetura, ¿no es un número primo!