Exámenes de "Programación funcional con Haskell"

Vol. 11 (Curso 2019–20)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Sevilla, 31 de octubre de 2019

Índice general

Introducción

Este libro es una recopilación de las soluciones de ejercicios de los exámenes de programación funcional con Haskell de la asignatura de Informática (curso 2019–20) del Grado en Matemática de la Universidad de Sevilla.

Los exámenes se realizaron en el aula de informática y su duración fue de 2 horas. La materia de cada examen es la impartida desde el comienzo del curso (generalmente, el 1 de octubre) hasta la fecha del examen. Dicha materia se encuentra en los libros de temas y ejercicios del curso:

- Temas de programación funcional (curso 2019–20) [1]
- Ejercicios de "Informática de 1º de Matemáticas" (2019–20) [2]
- Piensa en Haskell (Ejercicios de programación funcional con Haskell) [3]

El libro consta de 5 capítulos correspondientes a 5 grupos de la asignatura. En cada capítulo hay una sección por cada uno de los exámenes del grupo. Los ejercicios de cada examen han sido propuestos por los profesores de su grupo (cuyos nombres aparecen en el título del capítulo). Sin embargo, los he modificado para unificar el estilo de su presentación.

Finalmente, el libro contiene dos apéndices. Uno con el método de Polya de resolución de problemas (sobre el que se hace énfasis durante todo el curso) y el otro con un resumen de las funciones de Haskell de uso más frecuente.

Los códigos del libro están disponibles en GitHub [4]

Este libro es el 11º volumen de la serie de recopilaciones de exámenes de programación funcional con Haskell. Los volúmenes anteriores son

- Exámenes de "Programación funcional con Haskell". Vol. 1 (Curso 2009–10) [5]

[1] https://www.cs.us.es/~jalonso/cursos/i1m-19/temas/2019-20-IM-temas-PF.pdf
[2] https://www.cs.us.es/~jalonso/cursos/i1m-19/ejercicios/ejercicios-I1M-2019.pdf
[3] http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf
[4] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol11
[5] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol1

- Exámenes de “Programación funcional con Haskell”. Vol. 2 (Curso 2010–11) [6]
- Exámenes de “Programación funcional con Haskell”. Vol. 3 (Curso 2011–12) [7]
- Exámenes de “Programación funcional con Haskell”. Vol. 4 (Curso 2012–13) [8]
- Exámenes de “Programación funcional con Haskell”. Vol. 5 (Curso 2013–14) [9]
- Exámenes de “Programación funcional con Haskell”. Vol. 6 (Curso 2014–15) [10]
- Exámenes de “Programación funcional con Haskell”. Vol. 6 (Curso 2015–16) [11]
- Exámenes de “Programación funcional con Haskell”. Vol. 8 (Curso 2016–17) [12]
- Exámenes de “Programación funcional con Haskell”. Vol. 9 (Curso 2017–18) [13]
- Exámenes de “Programación funcional con Haskell”. Vol. 10 (Curso 2018–19) [14]

José A. Alonso
Sevilla, 31 de octubre de 2019

[6] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol2
[7] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol3
[8] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol4
[9] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol5
[10] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol6
[11] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol7
[12] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol8
[13] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol9
[14] https://github.com/jaalonso/Examenes_de_PF_con_Haskell_Vol10

1

Exámenes del grupo 3

Francisco J. Martín

1.1. Examen 1 (28 de octubre de 2019

```
-- Informática (1º del Grado en Matemáticas, Grupo 3)
-- 1º examen de evaluación continua (28 de octubre de 2019)
-- ===================================================================

-- -------------------------------------------------------------------
-- Ejercicio 1. Supongamos un tablero de ajedrez infinito de casillas de
-- tamaño 1x1, con una casilla negra situada entre las posiciones (0,0),
-- (1,0), (0,1) y (1,1). El tablero se extiende en todas direcciones
-- abarcando todo el plano. De esta forma, cada punto del plano está
-- pintado de color negro (el interior de las casillas negras y los
-- bordes) o de color blanco (el interior de las casillas blancas). Por
-- ejemplo, el punto (0.5,0.5) se encuentra pintado de negro (en el
-- interior de una casilla negra), el punto (0.5,1) también se encuentra
-- pintado de negro (en el borde de las casillas) y el punto (0.5,1.5)
-- se encuentra pintado de blanco (en el interior de una casilla blanca).
--
-- Definir la función
--   posicionNegra :: (Double,Double) -> Bool
-- tal que (posicionNegra (x,y)) se verifica si el punto (x,y) se encuentra
-- pintado de negro en el tablero de ajedrez infinito. Por ejemplo,
--   posicionNegra (0.5,0.5)    ==  True
--   posicionNegra (0.5,1)      ==  True
--   posicionNegra (0.5,1.5)    ==  False
--   posicionNegra (-0.5,0.5)   ==  False
```

```haskell
--   posicionNegra (0.5,-0.5)   ==  False
--   posicionNegra (-0.5,-0.5)  ==  True
-- ---------------------------------------------------------------------

-- 1ª solución
posicionNegra :: (Double,Double) -> Bool
posicionNegra (x,y)
  | floor x == ceiling x = True
  | floor y == ceiling y = True
  | even (floor x + floor y) = True
  | otherwise = False

-- 2ª solución
posicionNegra2 :: (Double,Double) -> Bool
posicionNegra2 (x,y) =
     floor x == ceiling x
  || floor y == ceiling y
  || even (floor x + floor y)

-- ---------------------------------------------------------------------
-- Ejercicio 2. Una lista de números [a(1),a(2),...] se denomina
-- posicionalmente coprima si para cada i ∈ {1, 2,...} se cumple que i y
-- a(i) son coprimos, es decir, el máximo común divisor de a(i) e i es
-- 1. Por ejemplo, la lista [3,5,7] es posicionalmente coprima pues 3 es
-- coprimo con 1, 5 es coprimo con 2 y 7 es coprimo con 3; por otro
-- lado, la lista [2,4,6] no es posicionalmente coprima pues 4 no es
-- coprimo con 2, ni 6 es coprimo con 3.
--
-- Definir la función
--    posicionalmenteCoprima :: [Integer] -> Bool
-- tal que (posicionalmenteCoprima xs) se cumple si la lista xs es
-- posicionalmente coprima. Por ejemplo,
--    posicionalmenteCoprima [3,5,7]  ==  True
--    posicionalmenteCoprima [2,4,6]  ==  False
--    posicionalmenteCoprima []       ==  True
--    posicionalmenteCoprima [6]      ==  True
-- ---------------------------------------------------------------------

-- 1ª solución
posicionalmenteCoprima :: [Integer] -> Bool
```

```haskell
posicionalmenteCoprima xs =
  and [gcd x i == 1 | (x,i) <- zip xs [1..]]

-- 2ª solución
posicionalmenteCoprima2 :: [Integer] -> Bool
posicionalmenteCoprima2 xs = aux xs [1..]
  where aux [] _          = True
        aux (x:xs) (n:ns) = gcd x n == 1 && aux xs ns

-- ---------------------------------------------------------------------
-- Ejercicio 3. Una forma de aproximar el valor del número π es usando
-- la siguiente igualdad:
--
--           π     4     16     36     64
--          --- = --- * ---- * ---- * ---- * ...
--           2     3     15     35     63
--
-- Es decir, el producto de los términos de la secuencia cuyo término general
-- n-ésimo es:
--                     4*k^2
--           s(k) = ---------, para k > 0
--                  4*k^2-1
--
-- Definir la función
--    aproximaPi :: Double -> Double
-- tal que (aproximaPi n) es la aproximación del número π calculada con la
-- serie anterior hasta el término n-ésimo (contando desde 0). Por ejemplo,
--    aproximaPi 10    ==  3.0677038066434985
--    aproximaPi 100   ==  3.133787490628162
--    aproximaPi 1000  ==  3.140807746030402
-- ---------------------------------------------------------------------

-- 1ª definición
aproximaPi :: Double -> Double
aproximaPi n =
  2 * product [(4*k^2) / (4*k^2-1) | k <- [1..n]]

-- 2ª definición
aproximaPiR :: Double -> Double
aproximaPiR 0 = 2
```

```haskell
aproximaPiR n = aproximaPiR (n-1) * 4 * n^2 / (4*n^2-1)

-- ---------------------------------------------------------------------
-- Ejercicio 4. La mecánica habitual para sumar números naturales
-- consiste en sumar todas las cifras que ocupan la misma posición,
-- tomadas en  orden desde las unidades, acumulando una 'llevada' desde
-- una posición a la siguiente. Consideremos un proceso de 'suma
-- sesgada' que no acumula la llevada, por ejemplo la suma ordinaria de
-- 1357 y 579 es 1936, sin embargo el valor de la suma sesgada es 1826:
--
--        Suma con llevada:               Suma sesgada:
--              1  1
--           1 3 5 7                         1 3 5 7
--         +   5 7 9                       +   5 7 9
--        ------------                    ------------
--           1 9 3 6                         1 8 2 6
--
-- Definir la función:
--    sumaSesgada :: Integer -> Integer -> Integer
-- tal que (sumaSesgada n m) es la 'suma sesgada' de los números naturales
-- n y m. Por ejemplo,
--    sumaSesgada 1357 579  ==  1826
--    sumaSesgada 9999 999  ==  9888
--    sumaSesgada 0 1234    ==  1234
-- ---------------------------------------------------------------------

-- 1ª solución
-- ===========

sumaSesgada :: Integer -> Integer -> Integer
sumaSesgada 0 m = m
sumaSesgada n 0 = n
sumaSesgada n m =
  sumaSesgada (div n 10) (div m 10) * 10 + mod (mod n 10 + mod m 10) 10

-- 2ª solución
-- ===========

sumaSesgada2 :: Integer -> Integer -> Integer
sumaSesgada2 n m =
```

```haskell
  sum [10^k * ((x + y) `mod` 10)
      | (k,x,y) <- zip3 [c-1,c-2..]
                   (replicate (c-a) 0 ++ xs)
                   (replicate (c-b) 0 ++ ys)]
  where xs = digitos n
        ys = digitos m
        a  = length xs
        b  = length ys
        c  = max a b

digitos :: Integer -> [Integer]
digitos n = [read [c] | c <- show n]
```

2

Exámenes del grupo 4

José A. Alonso

2.1. Examen 1 (30 de octubre de 2019)

```
-- Informática (1º del Grado en Matemáticas, Grupo 4)
-- 1º examen de evaluación continua (30 de octubre de 2019)
-- ---------------------------------------------------------------------

-- Nota: La puntuación de cada ejercicio es 2.5 puntos.


import Test.QuickCheck

-- ---------------------------------------------------------------------
-- Ejercicio 1. La distancia de Hamming entre dos listas es el número de
-- posiciones en que los correspondientes elementos son distintos. Por
-- ejemplo, la distancia de Hamming entre "roma" y "loba" es 2 (porque
-- hay 2 posiciones en las que los elementos correspondientes son
-- distintos: la 1ª y la 3ª).
--
-- Definir la función
--    distancia :: Eq a => [a] -> [a] -> Int
-- tal que (distancia xs ys) es la distancia de Hamming entre xs e
-- ys. Por ejemplo,
--    distancia "romano" "comino"  ==  2
--    distancia "romano" "camino"  ==  3
--    distancia "roma"   "comino"  ==  2
--    distancia "roma"   "camino"  ==  3
```

```haskell
--    distancia "romano" "ron"     ==  1
--    distancia "romano" "cama"    ==  2
--    distancia "romano" "rama"    ==  1
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
distancia :: Eq a => [a] -> [a] -> Int
distancia xs ys = sum [1 | (x,y) <- zip xs ys, x /= y]

-- 2ª definición (por recursión):
distancia2 :: Eq a => [a] -> [a] -> Int
distancia2 [] _ = 0
distancia2 _ [] = 0
distancia2 (x:xs) (y:ys) | x /= y    = 1 + distancia2 xs ys
                         | otherwise = distancia2 xs ys

-- ---------------------------------------------------------------------
-- Ejercicio 2. Definir la función
--    esPotencia :: Integer -> Integer -> Bool
-- tal que (esPotencia x a) se verifica si x es una potencia de a. Por
-- ejemplo,
--    esPotencia 32 2  ==  True
--    esPotencia 42 2  ==  False
-- ---------------------------------------------------------------------

-- 1ª definición (por comprensión):
esPotencia :: Integer -> Integer -> Bool
esPotencia x a = x `elem` [a^n | n <- [0..x]]

-- 2ª definición (por recursión):
esPotencia2 :: Integer -> Integer -> Bool
esPotencia2 x a = aux x a 0
  where aux x a b | b > x     = False
                  | otherwise = x == a ^ b || aux x a (b+1)

-- 3ª definición (por recursión):
esPotencia3 :: Integer -> Integer -> Bool
esPotencia3 0 _ = False
esPotencia3 1 a = True
esPotencia3 _ 1 = False
```

```haskell
esPotencia3 x a = rem x a == 0 && esPotencia3 (div x a) a

-- La propiedad de equivalencia es
prop_equiv_esPotencia :: Integer -> Integer -> Property
prop_equiv_esPotencia x a =
    x > 0 && a > 0 ==>
    esPotencia2 x a == b &&
    esPotencia3 x a == b
    where b = esPotencia x a

-- La comprobación es
--    ghci> quickCheck prop_equiv_esPotencia
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 3.1. Definir la función
--    sumaListas :: [Int] -> [Int] -> [Int]
-- tal que (sumaListas xs ys) es la suma de las elementos
-- correspondientes de las lista xs e ys. Por ejemplo,
--    sumaListas [2,3,4] [1,2,5]  ==  [3,5,9]
--    sumaListas [2,3,4] [1,2]    ==  [3,5,4]
--    sumaListas [2,3]   [1,2,5]  ==  [3,5,5]
-- ---------------------------------------------------------------------

-- 1ª definición
sumaListas :: [Int] -> [Int] -> [Int]
sumaListas [] ys         = ys
sumaListas xs []         = xs
sumaListas (x:xs) (y:ys) = x+y : sumaListas xs ys

-- 2ª definición
sumaListas2 :: [Int] -> [Int] -> [Int]
sumaListas2 xs ys = [x+y | (x,y) <- zip (xs ++ replicate (k-m) 0)
                                        (ys ++ replicate (k-n) 0)]
  where m = length xs
        n = length ys
        k = max m n
```

```haskell
-- ---------------------------------------------------------------------
-- Ejercicio 3.2. Comprobar con QuickCheck que el número de elementos de
-- (sumaListas xs ys) es el máximo de los números de elementos de xs e ys.
-- ---------------------------------------------------------------------

-- La propiedad es
prop_sumaListas :: [Int] -> [Int] -> Bool
prop_sumaListas xs ys =
  length (sumaListas xs ys) == max (length xs) (length ys)

-- La comprobación es
--    λ> quickCheck prop_sumaListas
--    +++ OK, passed 100 tests.

-- ---------------------------------------------------------------------
-- Ejercicio 4. Un número primo equilibrado es un número primo que es la
-- media aritmética de su primo anterior y siguiente. Por ejemplo, 5 es
-- un primo equilibrado porque es la media de 3 y 7; pero 7 no lo es
-- porque no es la media de 5 y 11.
--
-- Definir la función
--    primosEquilibrados :: Int -> [Int]
-- tal que (primosEquilibrados n) es la lista de los números primos
-- equilibrados menores o iguales que n. Por ejemplo,
--    ghci> primosEquilibrados 1000
--    [5,53,157,173,211,257,263,373,563,593,607,653,733,947,977]
-- ---------------------------------------------------------------------

primosEquilibrados :: Int -> [Int]
primosEquilibrados n = aux (primos n)
  where aux (x1:x2:x3:xs)
          | 2*x2 == x1+x3 = x2 : aux (x2:x3:xs)
          | otherwise     = aux (x2:x3:xs)
        aux _             = []

primos :: Int -> [Int]
primos n = [x | x <- [1..n], esPrimo x]

esPrimo :: Int -> Bool
esPrimo n = divisores n == [1,n]
```

```haskell
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `mod` x == 0]
```

Apéndice A

Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de x e y.
2. `x - y` es la resta de x e y.
3. `x / y` es el cociente de x entre y.
4. `x ^ y` es x elevado a y.
5. `x == y` se verifica si x es igual a y.
6. `x /= y` se verifica si x es distinto de y.
7. `x < y` se verifica si x es menor que y.
8. `x <= y` se verifica si x es menor o igual que y.
9. `x > y` se verifica si x es mayor que y.
10. `x >= y` se verifica si x es mayor o igual que y.
11. `x && y` es la conjunción de x e y.
12. `x || y` es la disyunción de x e y.
13. `x:ys` es la lista obtenida añadiendo x al principio de ys.
14. `xs ++ ys` es la concatenación de xs e ys.
15. `xs !! n` es el elemento n–ésimo de xs.
16. `f . g` es la composición de f y g.
17. `abs x` es el valor absoluto de x.
18. `and xs` es la conjunción de la lista de booleanos xs.
19. `ceiling x` es el menor entero no menor que x.
20. `chr n` es el carácter cuyo código ASCII es n.
21. `concat xss` es la concatenación de la lista de listas xss.
22. `const x y` es x.

23. `curry f` es la versión curryficada de la función `f`.
24. `div x y` es la división entera de `x` entre `y`.
25. `drop n xs` borra los `n` primeros elementos de `xs`.
26. `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
27. `elem x ys` se verifica si `x` pertenece a `ys`.
28. `even x` se verifica si `x` es par.
29. `filter p xs` es la lista de elementos de la lista `xs` que verifican el predicado `p`.
30. `flip f x y` es `f y x`.
31. `floor x` es el mayor entero no mayor que `x`.
32. `foldl f e xs` pliega `xs` de izquierda a derecha usando el operador f y el valor inicial `e`.
33. `foldr f e xs` pliega `xs` de derecha a izquierda usando el operador f y el valor inicial `e`.
34. `fromIntegral x` transforma el número entero `x` al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par `p`.
36. `gcd x y` es el máximo común divisor de de `x` e `y`.
37. `head xs` es el primer elemento de la lista `xs`.
38. `init xs` es la lista obtenida eliminando el último elemento de `xs`.
39. `iterate f x` es la lista `[x, f(x), f(f(x)), ...]`.
40. `last xs` es el último elemento de la lista `xs`.
41. `length xs` es el número de elementos de la lista `xs`.
42. `map f xs` es la lista obtenida aplicado `f` a cada elemento de `xs`.
43. `max x y` es el máximo de `x` e `y`.
44. `maximum xs` es el máximo elemento de la lista `xs`.
45. `min x y` es el mínimo de `x` e `y`.
46. `minimum xs` es el mínimo elemento de la lista `xs`.
47. `mod x y` es el resto de `x` entre `y`.
48. `not x` es la negación lógica del booleano `x`.
49. `noElem x ys` se verifica si `x` no pertenece a `ys`.
50. `null xs` se verifica si `xs` es la lista vacía.
51. `odd x` se verifica si `x` es impar.
52. `or xs` es la disyunción de la lista de booleanos `xs`.
53. `ord c` es el código ASCII del carácter `c`.

54. `product xs` es el producto de la lista de números `xs`.
55. `read c` es la expresión representada por la cadena `c`.
56. `rem x y` es el resto de `x` entre `y`.
57. `repeat x` es la lista infinita `[x, x, x, ...]`.
58. `replicate n x` es la lista formada por `n` veces el elemento `x`.
59. `reverse xs` es la inversa de la lista `xs`.
60. `round x` es el redondeo de `x` al entero más cercano.
61. `scanr f e xs` es la lista de los resultados de plegar `xs` por la derecha con `f` y `e`.
62. `show x` es la representación de `x` como cadena.
63. `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo.
64. `snd p` es el segundo elemento del par `p`.
65. `splitAt n xs` es `(take n xs, drop n xs)`.
66. `sqrt x` es la raíz cuadrada de `x`.
67. `sum xs` es la suma de la lista numérica `xs`.
68. `tail xs` es la lista obtenida eliminando el primer elemento de `xs`.
69. `take n xs` es la lista de los `n` primeros elementos de `xs`.
70. `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
71. `uncurry f` es la versión cartesiana de la función `f`.
72. `until p f x` aplica `f` a `x` hasta que se verifique `p`.
73. `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.
74. `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

A.1. Resumen de funciones sobre TAD en Haskell

A.1.1. Polinomios

1. `polCero` es el polinomio cero.
2. `(esPolCero p)` se verifica si `p` es el polinomio cero.
3. `(consPol n b p)` es el polinomio $bx^n + p$.
4. `(grado p)` es el grado del polinomio `p`.

5. `(coefLider p)` es el coeficiente líder del polinomio `p`.
6. `(restoPol p)` es el resto del polinomio `p`.

A.1.2. Vectores y matrices (`Data.Array`)

1. `(range m n)` es la lista de los índices del `m` al `n`.
2. `(index (m,n) i)` es el ordinal del índice `i` en `(m,n)`.
3. `(inRange (m,n) i)` se verifica si el índice `i` está dentro del rango limitado por `m` y `n`.
4. `(rangeSize (m,n))` es el número de elementos en el rango limitado por `m` y `n`.
5. `(array (1,n) [(i, f i) | i <- [1..n])` es el vector de dimensión `n` cuyo elemento `i`–ésimo es `f i`.
6. `(array ((1,1),(m,n)) [((i,j), f i j) | i <- [1..m], j <- [1..n]])` es la matriz de dimensión `m.n` cuyo elemento `(i,j)`–ésimo es `f i j`.
7. `(array (m,n) ivs)` es la tabla de índices en el rango limitado por `m` y `n` definida por la lista de asociación `ivs` (cuyos elementos son pares de la forma (índice, valor)).
8. `(t ! i)` es el valor del índice `i` en la tabla `t`.
9. `(bounds t)` es el rango de la tabla `t`.
10. `(indices t)` es la lista de los índices de la tabla `t`.
11. `(elems t)` es la lista de los elementos de la tabla `t`.
12. `(assocs t)` es la lista de asociaciones de la tabla `t`.
13. `(t // ivs)` es la tabla `t` asignándole a los índices de la lista de asociación `ivs` sus correspondientes valores.
14. `(listArray (m,n) vs)` es la tabla cuyo rango es `(m,n)` y cuya lista de valores es `vs`.
15. `(accumArray f v (m,n) ivs)` es la tabla de rango `(m,n)` tal que el valor del índice `i` se obtiene acumulando la aplicación de la función `f` al valor inicial `v` y a los valores de la lista de asociación `ivs` cuyo índice es `i`.

A.1.3. Tablas

1. `(tabla ivs)` es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores).
2. `(valor t i)` es el valor del índice `i` en la tabla `t`.
3. `(modifica (i,v) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `v`.

A.1.4. Grafos

1. `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
2. `(dirigido g)` se verifica si `g` es dirigido.
3. `(nodos g)` es la lista de todos los nodos del grafo `g`.
4. `(aristas g)` es la lista de las aristas del grafo `g`.
5. `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
6. `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
7. `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

Apéndice B

Método de Pólya para la resolución de problemas

B.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?

- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos
- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?
- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya “Cómo plantear y resolver problemas” (Ed. Trillas, 1978) p. 19

B.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?
- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?
- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.
- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.