

```
-- ENUNCIADO
-- Se trata de, dada una lista de enteros "nums" y un entero "n" (todos positivos),
-- construir una expresión aritmética (con suma, resta, producto y división entera) que
-- utilice cada número de "nums" a lo sumo una vez y cuyo valor sea "n".
-- Ej.: nums = [1,2,3,4,7] y n = 19 tiene como soluciones
--         (3*4)+7, (7*3)-2, (7*3)-(4/2), (7*2)+(4+1), ...
-- Además, todos los resultados intermedios deben ser enteros y positivos
-- En los ejemplos de arriba se cumple ya que 3*4, 7*3, 4/2, 7*2, 4+1, son enteros positivos.
-- Sin embargo, no sería aceptable como solución al ejemplo de arriba
-- (7*3)+(1-3) ya que (1-3) no es positivo.
```

```
-- ESPECIFICACIÓN DEL PROBLEMA
```

```
data Op = Sum | Res | Mul | Div
      deriving Show
```

```
aplAcept :: Op -> Int -> Int -> Bool
```

```
-- (aplAcept o i1 i2) decide si la aplicación de o a dos enteros positivos i1 e i2
-- produce un entero positivo (es aceptable)
```

```
aplAcept Sum _ _ = True
```

```
aplAcept Res x y = x > y
```

```
aplAcept Mul _ _ = True
```

```
aplAcept Div x y = x `mod` y == 0
```

```
{-
Main> aplAcept Res 5 7
False
Main> aplAcept Res 5 4
True
-}
```

```
aplicar :: Op -> Int -> Int -> Int
```

```
-- (aplicar o i1 i2) da el entero que resulta de aplicar o a i1 e i2
```

```
aplicar Sum x y = x + y
```

```
aplicar Res x y = x - y
```

```
aplicar Mul x y = x * y
```

```
aplicar Div x y = x `div` y
```

```
data Expr = Val Int | App Op Expr Expr
      deriving Show
```

```
-- Ejemplos
```

```
-- exp1 representa (1+50)*(25-10)
```

```
exp1 = App Mul (App Sum (Val 1) (Val 50)) (App Res (Val 25) (Val 10))
```

```
exp2 = App Mul (App Sum (Val 1) (Val 50)) (App Res (Val 10) (Val 25))
```

```
valores :: Expr -> [Int]
```

```
-- (valores e) da la lista de todos los enteros que aparecen en e
```

```
valores (Val n) = [n]
```

```
valores (App _ l r) = valores l ++ valores r
```

```
{-
Main> valores exp1
[1,50,25,10]
-}
```

```

evaluar :: Expr -> [Int]
-- (evaluar e) obtiene la lista vacía si alguna sub-expresión de e es no aceptable
-- (se evalúa a un número no-positivo),
-- en caso contrario da la lista que contiene el entero que resulta de evaluar e.
evaluar (Val n) = [n | n > 0]
evaluar (App o l r) = [aplicar o x y | x <- evaluar l, y <- evaluar r, aplAcept o x y ]
{-
Main> evaluar exp1
[765]
Main> evaluar exp2
[]
-}

subcjets :: [a] -> [[a]]
subcjets xs = [zs | ys <- sublis xs, zs <- perms ys]
  where
    sublis :: [a] -> [[a]]
    sublis [] = [[]]
    sublis (x:xs) = let yss = sublis xs in yss ++ map (x:) yss
    perms :: [a] -> [[a]]
    perms [] = [[]]
    perms (x:xs) = let
        intercalar :: a -> [a] -> [[a]]
        intercalar x [] = [[x]]
        intercalar x (y:ys) = (x:y:ys) : map (y:) (intercalar x ys)
      in concat (map (intercalar x) (perms xs))
    {-
Main> subcjets [1,2,3]
[[],[3],[2],[2,3],[3,2],[1],[1,3],[3,1],[1,2],[2,1],[1,2,3],[2,1,3],[2,3,1],[1,3,2],
[3,1,2],[3,2,1]]
-- Pruebas de funciones locales (script separado)
Main> sublis [1,2,3]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
Main> intercalar 1 [2,3]
[[1,2,3],[2,1,3],[2,3,1]]
Main> perms [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
-}

esSol :: Expr -> [Int] -> Int -> Bool
-- esSol e nums n decide si e es una solución usando la lista nums para el numero objetivo n
esSol e nums n = elem (valores e) (subcjets nums) && evaluar e == [n]
{-
Main> esSol exp1 [1,3,7,10,25,50] 765
True
Main> esSol exp1 [1,3,7,10,25,50] 831
False
-}

-- IMPLEMENTACIÓN "DE FUERZA BRUTA"

escisiones :: [a] -> [[a],[a]]

```

```

-- Calcula todos los pares de listas cuya concatenación es la lista parámetro
escisiones [] = [([],[])]
escisiones (x:xs) = ([],x:xs) : [(x:ls,rs) | (ls,rs) <- escisiones xs]
{-
Main> escisiones [1,2,3,4,5]
[( [], [1,2,3,4,5]), ([1], [2,3,4,5]), ([1,2], [3,4,5]), ([1,2,3], [4,5]), ([1,2,3,4], [5]), ([1,2,3,4,5], [])]
-}

escisionesNoVac :: [a ] -> [[a],[a]]
escisionesNoVac = filter noVac . escisiones
    where
        noVac :: ([a],[b]) -> Bool
        noVac (xs, ys) = not (null xs || null ys)
{-
Main> escisionesNoVac [1,2,3,4,5]
[[ [1], [2,3,4,5]), ([1,2], [3,4,5]), ([1,2,3], [4,5]), ([1,2,3,4], [5])]
-}

listaExprs :: [Int] -> [Expr]
-- Calcula la lista de todas las expresiones que se pueden construir con la lista de valores
-- que se le pasa como parámetros.
listaExprs [] = []
listaExprs [n] = [Val n]
listaExprs ns = [e | (ls, rs) <- escisionesNoVac ns, l <- listaExprs ls, r <- listaExprs rs, e
  <- combinar l r]
    where
        combinar :: Expr -> Expr -> [Expr]
        combinar l r = [App o l r | o <- [Sum, Res, Mul, Div]]
{-
Main> length (listaExprs [1,3,7,10,25,50])
43008
Main> listaExprs [1,3]
[App Sum (Val 1) (Val 3),
App Res (Val 1) (Val 3),
App Mul (Val 1) (Val 3),
App Div (Val 1) (Val 3)]
Main> length (listaExprs [1,3,7])
32
-}

listaExprsSol :: [Int] -> Int -> [Expr]
listaExprsSol nums n = [e | nums' <- subcjts nums, e <- listaExprs nums', evaluar e == [n]]
{-
Main> head (listaExprsSol [1,3,7,10,25,50] 765)
App Mul (Val 3) (App Res (App Mul (Val 7) (App Res (Val 50) (Val 10)))) (Val 25))
Main> length (listaExprsSol [1,3,7,10,25,50] 765) -- tarda unos minutos
780
-}

-- Ejercicio 1:
-- MEJORAR LA FORMA EN QUE LAS EXPRESIONES SE MUESTRAN
-- Definir instancias

```

```
-- instance Show Op where show .....
-- instance Show Exp where show .....
```

```
-- tales que:
```

```
{-
```

```
Main> exp1
```

```
((1+50)*(25-10))
```

```
Main> exp2
```

```
((1+50)*(10-25))
```

```
Main> head (listaExprsSol [1,3,7,10,25,50] 765)
```

```
(3*((7*(50-10))-25))
```

```
(23500004 reductions, 41992688 cells, 42 garbage collections)
```

```
Main> (3*((7*(50-10))-25))
```

```
765
```

```
-}
```

```
-- 8 líneas de código
```

```
-- Ejercicio 2:
```

```
-- MEZCLAR LA GENERACIÓN Y LA EVALUACIÓN DE LAS EXPRESIONES PARA GANAR EFICIENCIA
```

```
-- Utilizar el siguiente tipo de pares que asocian a cada expresión su valor
```

```
type Resultado = (Expr, Int)
```

```
-- a) Definir una función "resultados" que dada una lista de números nums calcule
```

```
-- todas los pares (e,v) tales que e es una expresión que se puede construir con nums
```

```
-- y v es su valor. La definición más obvia es
```

```
-- resultados:: [Int] -> [Resultado]
```

```
-- resultados nums = [(e, v) | e <- listaExprs nums, v <- evaluar e]
```

```
-- pero se trata de mezclar la generación con la evaluación de modo que sea más eficiente
```

```
-- La nueva función resultados debe sustituir a la función "listaExprs" generando
```

```
-- los pares directamente, es decir sin hacer uso ni de "listaExprs" ni de "evaluar".
```

```
-- Debe mezclar efectivamente la ejecución de dichas dos funciones.
```

```
-- 10 líneas de código
```

```
-- b) Definir una función
```

```
-- listaExprsSol' :: [Int] -> Int -> [Expr]
```

```
-- que utilice la función resultados y haga lo mismo (pero más eficientemente)
```

```
-- que la función "listaExprSol" de arriba.
```

```
-- 2 líneas de código
```

```
-- Activar la opción "Print statistics (heap cells, reductions) after each computation"
```

```
-- y comprobar la diferencia entre listaExprsSol y listaExprSol'.
```

```
-- Para ello crear un conjunto de 30 posibles inputs que incluyan los dos siguientes
```

```
-- nums=[1,3,7,10,25] , n=135
```

```
-- nums=[1,3,7,10,25,50], n=765
```

```
-- Como indicación:
```

```
{-
```

```
Main> length (listaExprsSol [1,3,7,10,25] 135)
```

```
208
```

```
(34837914 reductions, 62156676 cells, 63 garbage collections)
```

```
Main> length (listaExprsSol' [1,3,7,10,25] 135)
```

```
208
```

```
(2880034 reductions, 6395256 cells, 6 garbage collections)
```

```
Main> head (listaExprsSol [1,3,7,10,25] 135)
```

```
(3*((7*10)-25))
```

```
(52386 reductions, 96831 cells)
```

```
Main> head (listaExprsSol' [1,3,7,10,25] 135)
```

```
(3*((7*10)-25))
```

```
(13801 reductions, 29310 cells)
```

```
Main> head (listaExprsSol [1,3,7,10,25,50] 765)
```

```
(3*((7*(50-10))-25))
```

```
(23500104 reductions, 41992901 cells, 42 garbage collections)
```

```
Main> head (listaExprsSol' [1,3,7,10,25,50] 765)
```

```
(3*((7*(50-10))-25))
```

```
(1837214 reductions, 4086446 cells, 4 garbage collections)
```

```
Main> length (listaExprsSol [1,3,7,10,25,50] 765)
```

```
780
```

```
(2764174929 reductions, 591273642 cells, 4955 garbage collections)
```

```
Main> length (listaExprsSol' [1,3,7,10,25,50] 765)
```

```
780
```

```
(134700242 reductions, 302637367 cells, 306 garbage collections)
```

```
-}
```

```
-- Ejercicio 3:
```

```
-- USO DE PROPIEDADES ALGEBRAICAS PARA GANAR EFICIENCIA
```

```
-- La función "aplAcept" debería utilizar las siguientes propiedades
```

```
-- para evitar la generación de soluciones equivalentes:
```

```
-- 1) la suma y el producto son operaciones conmutativas de manera que se tome como aceptable
```

```
-- solo una entre (Sum il 12) y (Sum i2 il), e igual con Mul
```

```
-- 2) el número 1 es el neutro del producto, por tanto la multiplicación por 1
```

```
-- da una solución para la que existe otro menor
```

```
-- 3) lo mismo con la división por 1.
```

```
-- En otras palabras, las siguientes listas de soluciones deberían ser de longitud 1:
```

```
{-
```

```
Main> listaExprsSol' [1,2,3] 7
```

```
[(1+(2*3)),((2*3)+1),(1+(3*2)),((3*2)+1)]
```

```
Main> listaExprsSol' [1,2,3,5] 35
```

```
[((1+(2*3))*5),(((2*3)+1)*5),((1+(3*2))*5),  
(((3*2)+1)*5),(5*(1+(2*3))), (5*((2*3)+1)), (5*(1+(3*2))), (5*((3*2)+1))]
```

```
Main> listaExprsSol' [1,4,24] 6
```

```
[(24\4),(1*(24\4)),((1*24)\4),(24\((1*4)),((24*1)\4),  
((24\1)\4),(24\((4*1)),(24\((4\1)),((24\4)*1),((24\4)\1)]
```

```
-}
```

```
-- Se pide: Definir una nueva versión aplAcept' de dicha función que no genere
```

```
-- soluciones equivalentes (en cuanto a las tres propiedades de arriba).
```

```
-- Con aplAcept' en lugar de aplAcept, se obtiene una nueva versión
```

```
-- listaExprsSol'' tal que:
```

```
{-
```

```
Main> listaExprsSol'' [1,2,3] 7
```

```
[(1+(2*3))]
```

```
Main> listaExprsSol'' [1,2,3,5] 35
[(5*(1+(2*3)))]

Main> listaExprsSol'' [1,4,24] 6
[(24\4)]

Main> length (listaExprsSol'' [1,3,7,10,25] 135)
16 -- antes 208
(1266328 reductions, 2200969 cells, 2 garbage collections)

Main> head (listaExprsSol'' [1,3,7,10,25] 135)
(3*((7*10)-25))
(16352 reductions, 28585 cells)

Main> length (listaExprsSol'' [1,3,7,10,25,50] 765)
49 -- antes 780
(36713935 reductions, 63208096 cells, 64 garbage collections)

Main> head (listaExprsSol'' [1,3,7,10,25,50] 765)
(3*((7*(50-10))-25))
(1298908 reductions, 2225609 cells, 2 garbage collections)
-}
```

```
-- 16 líneas de código
```

```
-- Ejercicio 4:
```

```
-- APLICACIÓN QUE INTERACCIONE CON EL USUARIO
```

```
-- Diseñar una función de entrada/salida que
```

- 1.- pida al usuario los datos de entrada
- 2.- compruebe que dichos datos son admisibles y en caso negativo los pida de nuevo
- 3.- una vez que los datos son admisibles, pregunte al usuario si quiere conocer el número de soluciones, una solución, todas las soluciones o salir.
- 4.- A continuación, le muestre los resultados solicitados y vuelva a proponerle las 4 opciones o termine (en el caso de haber elegido salir).
- 5.- En el caso de una solución, se debe mostrar al usuario la primera solución, e informar de si hay más soluciones, dando la opción de salir o mostrarle la siguiente solución (si la hay)

```
-- EJERCICIOS OPTATIVOS:
```

```
-- 5.- Modificar el programa para que las soluciones se generen por orden creciente del numero de valores en nums que utilizan para alcanzar el resultado.
```

```
-- 6.- Modificar la forma de mostrar la solución al usuario como se muestra en el siguiente ejemplo: Si la solución a mostrar es (3*((7*10)-25)), el output sería:
```

```
-- 7*10 = 70
```

```
-- 70-25 = 45
```

```
-- 3*45 = 135
```

```
-- c) Incorporar cualquier otra mejora de eficiencia, mostrando con las estadísticas la ganancia sobre el juego de 30 pruebas.
```