EJERCICIOS BÁSICOS

1) Definir la función **max**, que dados dos números devuelve el máximo de los dos.

```
max2 :: (Int, Int) -> Int
max2 (x, y) = if x > y then x else y
```

2) Definir la función **max3**, que dados tres números devuelve el máximo de los tres.

```
max3 :: (Int, Int, Int) -> Int
max3 (x, y, z) = max2 (max2 (x, y), z)
```

3) Definir una función que tome 3 números y los "ordene de menor a mayor".

```
ord3 :: (Int, Int, Int) -> (Int, Int, Int)
ord3 (x, y, z) = (min3 (x, y, z), med3 (x, y, z), max3 (x, y, z))

min3 :: (Int, Int, Int) -> Int
min3 (x, y, z) = min (min x y) z

med3 :: (Int, Int, Int) -> Int
med3 (x, y, z) = x + y + z - min3 (x, y, z) - max3 (x, y, z)
```

4) Programar las funciones booleanas clásicas Not, Or, And, Xor, Imp, Eqv. Comparar alguna de ellas con alguna versión que use pattern matching.

NOT

```
notF :: Bool -> Bool
notF b = if b then False else True

notPM :: Bool -> Bool
notPM True = False
notPM b = True
```

OR

```
orF :: (Bool, Bool) -> Bool
orF (x, y) = if x then True else y

orPM :: (Bool, Bool) -> Bool
orPM (True, y) = True
orPM (False, y) = y
```

AND

```
andF :: (Bool, Bool) -> Bool
andF (x, y) = if x then y else False

andPM :: (Bool, Bool) -> Bool
andPM (False, y) = False
andPM (True, y) = y
```

XOR

```
xorF :: (Bool, Bool) -> Bool
xorF (x, y) = if x /= y then True else False

xorPM :: (Bool, Bool) -> Bool
xorPM (False, y) = y
xorPM (True, y) = not y
```

IMP

```
impF :: (Bool, Bool) -> Bool
impF (x, y) = if x == y then True else y

impPM :: (Bool, Bool) -> Bool
impPM (False, y) = True
impPM (True, y) = y
```

EQV

```
eqvF :: (Bool, Bool) -> Bool
eqvF (x, y) = if x == y then True else False

eqvPM :: (Bool, Bool) -> Bool
eqvPM (True, y) = y
eqvPM (False, y) = not y

eqvPM2 :: (Bool, Bool) -> Bool
eqvPM2 (True, True) = True
eqvPM2 (False, False) = True
eqvPM2 _ = False
```

5) Programar la suma y el producto de números complejos. Usar para esto el tipo par (producto cartesiano, 2-upla).

```
sumaComplejos :: ((Int, Int), (Int, Int)) -> (Int, Int)
sumaComplejos ((a, b), (c, d)) = (a + c, b + d)

productoComplejos :: ((Int, Int), (Int, Int)) -> (Int, Int)
productoComplejos ((a, b), (c, d)) = (a * c - b * d, a * d + b * c)
```

6) Programar la suma y el producto de matrices de 2 × 2 enteros, representando cada matriz como un par de pares.

```
sumaMatriz :: (((Int, Int), (Int, Int)), ((Int, Int), (Int, Int))) -> ((Int, Int), (Int, Int))
sumaMatriz (((a1,b1),(c1,d1)), ((a2,b2),(c2,d2))) = ((a1+a2, b1+b2), (c1+c2, d1+d2))

productoMatriz :: (((Int, Int), (Int, Int)), ((Int, Int), (Int, Int))) -> ((Int, Int), (Int, Int))
productoMatriz (((a1,b1),(c1,d1)), ((a2,b2),(c2,d2))) = ((a1*a2 + b1*c2, a1*b2 + b1*d2), (c1*a2 + d1*c2, c1*b2 + d1*d2))
```

7) a) Definir la función **fib** que devuelva un elemento dado de la sucesión de Fibonacci.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

7) b) Definir una función que devuelva un elemento dado de la sucesión de Tribonacci.

```
trib :: Int -> Int
trib 0 = 0
trib 1 = 0
trib 2 = 1
trib n = trib (n - 1) + trib (n - 2) + trib (n - 3)
```

8) Definir las funciones **par** (devuelve si un número es par o no) e impar usando recursión... y de distintas maneras.

```
par :: Int -> Bool
par 0 = True
par 1 = False
par x = par (x - 2)

par2 :: Int -> Bool
par2 x = x `mod` 2 == 0
```

```haskell
impar :: Int -> Bool
impar 0 = False
impar 1 = True
impar x = impar (x - 2)

-- Recursión mutua --

parPM :: Int -> Bool
parPM 0 = True
parPM x = imparPM (x - 1)

imparPM :: Int -> Bool
imparPM 0 = False
imparPM x = parPM (x - 1)
```

9) Definir la función **esDivisiblePor**, que determina si un entero es divisible por otro (hacerlo recursivamente, sin usar la función predefinida mod). En primera instancia, puede ignorarse los negativos.

```haskell
esDivisiblePor :: (Int, Int) -> Bool
esDivisiblePor (0, y) = True
esDivisiblePor (x, 0) = False
esDivisiblePor (x, y) = if x < y then False else esDivisiblePor (x-y, y)
```

10) Definir una función que determine si un entero positivo dado es primo.

```haskell
esPrimo :: Int -> Bool
esPrimo 1 = False
esPrimo n = not (esDivisiblePorAlguno (n, 2, n-1))

esDivisiblePorAlguno :: (Int, Int, Int) -> Bool
esDivisiblePorAlguno (n, desde, hasta) = if desde > hasta then False else
                                         (if esDivisiblePor (n, desde) then True else esDivisiblePorAlguno (n, desde + 1, hasta))
```

11) Definir la función **mcd**, que calcule el máximo común divisor de dos enteros dados (distintos de 0).

```haskell
mcd :: (Int, Int) -> Int
mcd (a, b) = if a == b then a else (if a > b then mcd (a-b, b) else mcd (a, b-a))
```

12) Definir la función **mcm**, que calcule el mínimo común múltiplo de dos enteros dados (distintos de 0).

```haskell
mcm :: (Int, Int) -> Int
mcm (a, b) = (a*b) `div` mcd(a, b)
```

13) Escribir un programa que dados dos números indique cuántos números terminados en 7 hay entre ambos, inclusive.
Por ejemplo, cuantos(18, 57) devolverá 4.

```haskell
cuantos :: (Int, Int, Int) -> Int
cuantos (d, h, c) = if d > h then 0 else (if (d `mod` 10) == c then 1 + cuantos (d+1, h, c) else cuantos (d+1, h, c))
```

15) Extender la función factorial ya escrita, de modo que admita valores negativos. Para una entrada negativa el resultado deberá ser el mismo que para la entrada positiva con el mismo valor absoluto.

```haskell
fact :: Int -> Int
fact 0 = 1
fact n = if n > 0 then n * fact(n-1) else -n * fact(n+1)
```