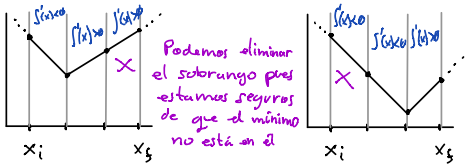
## Ejercicio 3

Para resolver el ejercicio se hace uso de una interfaz que define un protocolo de comunicación de una función a la que se le proporciona un valor x y nos retorna un valor $f(x)$.
Se implementa una función cuadrática que extiende de la interfaz para el ejemplo.
El algoritmo utilizado consiste en tomar de la función un rango $[x_i, x_f]$. Dicho rango será dividido en 3 partes iguales sobre las que se evaluará el crecimiento de la función.
Buscamos uno de los siguientes casos:




En estos dos casos concretos habremos reducido el rango de búsqueda donde sabemos que hay un mínimo relativo de la función.
En cualquier otra situación sabremos que dentro del rango no habrá ningún mínimo relativo.

Repitiendo recursivamente el procedimiento acotaremos en 2/3 el rango inicial en cada iteración.
La recursión tiene como caso base que $x_i - x_f$ sea menor que un valor que indica la precisión del resultado obtenido.

```
/**
 Función que encuentra el mínimo relativo de la una función en un rango con la precisión indicada
 */
func findMIn (over function : Function, x1 : Double = 300, x4 : Double = -300, precision min_range : Double = 0.0001) -> (Double, Double)? {
    let range_width = x1 - x4
    if (range_width < min_range){
        return (x1, x4)
    }
    //Dividimos la función en 3 rangos de igual tamaño (Cada vez el tamaño será menor)
    let x2 = x4 + range_width/2
    let x3 = x4 + range_width/4
    let y1 = function.getY(x: x1)
    let y2 = function.getY(x: x2)
    let y3 = function.getY(x: x3)
    let y4 = function.getY(x: x4)
    //Evaluamos si la función crece o decrece en cada rango
    let range1 = y1 > y2
    let range2 = y2 > y3
    let range3 = y3 > y4
    if (range1 && !range2 && !range3) { //El mínimo está entre los rangos 1 y 2
        return findMIn(over: function, x1: x1, x4 : x3, precision: min_range)
    } else if (range1 && range2 && !range3) { //El mínimo está entre los rangos 2 y 3
        return findMIn(over: function, x1: x2, x4 : x4, precision: min_range)
    } else { //No sabemos donde está el mínimo
        return nil
    }
}
```

## Ejercicio 6

Para resolver el ejercicio se puede expresar el callejero como una matriz N×N de modo que la traspuesta de dicha matriz sea el resultado.
Se ha creado un algoritmo divide y vencerás capaz de transponer matrices de cualquier tamaño.

1. Partiendo de una matriz N×M se obtiene max(N,M)
2. Se obtiene la potencia de dos igual o superior a max(N,M)
3. Se extiende la matriz original rellenando con 0s hasta que sus dimensiones sean las de la potencia de dos mencionada.
4. Recursivamente la matriz se divide en cuartos hasta llegar al caso base de una matriz 2×2 donde se realiza la transposición $\begin{bmatrix} A & B \\ C & D \end{bmatrix} \rightarrow \begin{bmatrix} A & C \\ B & D \end{bmatrix}$
5. Los cuartos de matriz generados se unen de forma que también son traspuestos.
6. Finalmente de la matriz resultado solo se proporciona la parte correspondiente al tamaño de matriz introducido. $\begin{bmatrix} & \\ & \end{bmatrix}^T = \begin{bmatrix} & \\ & \end{bmatrix}$

Adaptación de la entrada (1)

```
//Tamaño de la matriz (múltiplo de 2) sobre el que el algoritmo puede aplicarse
let toGO = max(Int(pow(2,log2(Double(cols)).rounded(FloatingPointRoundingRule.up))),
    Int(pow(2,log2(Double(rows)).rounded(FloatingPointRoundingRule.up))))
let inc_cols = toGO - cols
let inc_rows = toGO - rows
//Se amplia la matriz original hasta el tamaño anterior
if (inc_rows > 0){
    for index in 0..<domain.count{
        domain[index].append(contentsOf: [Int].init(repeating: 0, count: inc_rows))
    }
}
if (inc_cols > 0){
    for _ in 0..<inc_cols{
        domain.append([Int].init(repeating: 0, count: (rows+inc_rows)))
    }
}
```

(2)
(3)

Función principal:

```swift
/**
 Función recursiva que implmenta un algoritmo divide y vencerás para transponer matrices
 */
func transpose (_ data :[[Int]]) ->[[Int]]{
    //Caso base
    if data.count == 2 {
        var data = data
        let temp = data[0][1]
        data[0][1] = data [1][0]
        data[1][0] = temp
        return data
    }else{
        //La matriz se divide en cuatro cuartos
        let reduced =  quarter(data)
        var r1 = transpose(reduced.0)
        var r2 = transpose(reduced.1)
        var r3 = transpose(reduced.2)
        var r4 = transpose(reduced.3)
        //Se montan los resultados parciales
        let num = r1[1].count
        for i in 0..<num{
            r1[i].append(contentsOf: r2[i])
        }
        for i in 0..<num{
            r3[i].append(contentsOf: r4[i])
        }
        for i in 0..<num{
            r1.append(r3[i])
        }
        return r1
    }
}
```

(4)

(5)

Función auxiliar para dividir en cuartos:

```swift
/**
 Toma una matriz y la divide en cuatro partes iguales correspondiento estas con los 4 cuadrantes
 */
func quarter (_ data :[[Int]]) -> ([[Int]],[[Int]],[[Int]],[[Int]]) {
    let cropSize = data.count/2;
    var q1 = [[Int]]()
    var q2 = [[Int]]()
    var q3 = [[Int]]()
    var q4 = [[Int]]()
    for i in 0..<cropSize{
        q1.append(Array<Int>(data[i][0..<cropSize]))
    }
    for i in cropSize..<data.count{
        q2.append(Array<Int>(data[i][0..<cropSize]))
    }
    for i in 0..<cropSize{
        q3.append(Array<Int>(data[i][cropSize..<data.count]))
    }
    for i in cropSize..<data.count{
        q4.append(Array<Int>(data[i][cropSize..<data.count]))
    }
    return (q1,q2,q3,q4)
}
```

Se proporciona como el resultado solo el fragmento adecuado de la matriz generada

```swift
print ("Transposed:")
//De la matriz resultado se muestra solo la parte correspindiente al tamaño de matriz introducido
var str = "["
for i in 0..<rows{
    for j in 0..<cols {
        if j == cols-1{
            str += "\(result[i][j])]"
        }
        else{
            str += "\(result[i][j]), "
        }
    }
    print(str)
    str = "["
}
```

(6)