1.swift

```
import Foundation
enum NivelCargaBateria {
    case vacio, medio, lleno
    mutating func aumentar() {
        switch self {
            case .vacio:
                self = .medio
            case .medio:
                self = .lleno
            case .lleno:
                break
        }
    }
}
var cargaTelefono = NivelCargaBateria.vacio
cargaTelefono.aumentar()
print(cargaTelefono)
cargaTelefono.aumentar()
print(cargaTelefono)
cargaTelefono.aumentar()
print(cargaTelefono)
```

10.swift

```
import Foundation
enum OperacionesCadenas {
    case concatenaInicio
    case concatenaFin
    case intercala
}
func construye (operador op: OperacionesCadenas) -> (String, String)->String {
    switch op {
        case .concatenaInicio:
            return {$0 + $1}
        case .concatenaFin:
            return {$1 + $0}
        case .intercala:
            return {$0 + $1 + $0}
    }
}
var f = construye(operador:OperacionesCadenas.concatenaInicio)
print(f("Hola", "Adios")) // Imprime HolaAdios
f = construye(operador: OperacionesCadenas.concatenaFin)
print(f("Hola", "Adios")) // Imprime AdiosHola
f = construye(operador: OperacionesCadenas.intercala)
print(f("Hola", "Adios")) // Imprime AdiosHolaAdios
```

13.swift

```
import Foundation
func añade(_ array: inout [Int], _ a: Int, veces n: Int) {
    array.append(a)  // Añade el primer elemento
    for _ in 0..<n - 1 {
        array.append(a)  // Añade los n-1 elementos restantes
    }
}
var numeros = [1, 2]
añade(&numeros, 10, veces:3) // Añadimos el & (inout)
print(numeros) // Imprime [1, 2, 10, 10, 10]
/*
    A: Porque el error Bitch?
    El código inicial tiene un error porque el array numeros no se está modificando correctamente dentro de
    Esto se debe a que los arrays en Swift se pasan por valor, no por referencia, lo que significa que cualqu
*/
```

14.swift

```
import Foundation
struct Estricto {
    static func normaliza(valor entero:Int) -> Int {
        let valor = entero % 100
        return (valor < 0) ? 100 + valor:valor
    }
    var valor: Int {
        didSet {
            valor = Estricto.normaliza(valor:valor)
        }
    }
    init(valor valorInicial:Int = 0) {
        valor = Estricto.normaliza(valor: valorInicial)
    }
}
var e = Estricto(valor:25)
print(e.valor) // Imprime 25
e = Estricto(valor: 150)
print(e.valor) // Imprime 50
e.valor = -25
print(e.valor) // Imprime 75
```

15.swift

```
import Foundation
class Vehiculo {
    var velocidad = 0.0
    func hazSonido() {
        print("Sonido genérico")
    }
    init(velocidad:Double) {
        self.velocidad = velocidad
    }
}
class CocheElectrico:Vehiculo {
    var nivelBateria = 0.0
    init(velocidad:Double, nivelBateria:Double) {
        self.nivelBateria = nivelBateria
        super.init(velocidad: velocidad)
    }
    override func hazSonido() {
        print("Zoooom")
    }
}
let vehiculos = [Vehiculo(velocidad: 50.0), CocheElectrico(velocidad: 50.0, nivelBateria: 100.0)]
for vehiculo in vehiculos { vehiculo.hazSonido() }
```

16.swift

```
import Foundation
struct Intervalo<T:Comparable> {
    var minimo: T
    var maximo: T
    func contiene(_ valor:T)-> Bool {
        return valor >= minimo && valor <= maximo
    }
}
let intervaloEnteros = Intervalo(minimo: 0, maximo: 10)
let intervaloCadenas = Intervalo(minimo: "a", maximo: "z")
let contieneEntero = intervaloEnteros.contiene(4)
print(contieneEntero)
let contieneCadena = intervaloCadenas.contiene("A")
print(contieneCadena)
```

17.swift

```
import Foundation
struct Color {
    let rojo, verde, azul: Double
    init(blanco:Double) {
        rojo = blanco
        verde = blanco
        azul = blanco
    }
}
let magenta = Color(rojo:1.0, verde:0.0, azul:1.0)
/*
    La respuesta es la C, puesto que ya no podremos usar memberwise al declarar nuestro inicializador
*/
```

18.swift

```
import Foundation
protocol A {
    var descripcion: String {get set}
    func foo()-> Int?
}
struct MiStruct:A {
    var descripcion: String
    func foo() -> Int? {
        return 1
    }
}
let xd = MiStruct(descripcion: "Chupamela")
let res = xd.foo()
print(res!)
// La respuesta correcta es la b
```

2.swift

```
import Foundation
struct Vector2D {
    var x = 0.0, y = 0.0
    static func * (izquierdo:Double, derecho:Vector2D) -> Vector2D {
        return Vector2D(x:izquierdo * derecho.x, y:izquierdo * derecho.y)
    }
}
let vector = Vector2D(x:3.0, y:1.0)
let resultado = 2.0 * vector
print(resultado)
```

3.swift

```
import Foundation
struct Arbol<T> {
    var dato:T
    var hijos: [Arbol] = []
    // No hace falta hacer esto
    func imprimirArbol(_ prefijo: String = "", _ esUltimo: Bool = true) {
        // Imprime el nodo actual
        print("\(prefijo)\(esUltimo ? "■■■ " : "■■■ ")\(dato)")
        // Calcula el nuevo prefijo
        let nuevoPrefijo = prefijo + (esUltimo ?  "    " : "■   ")
        // Imprime cada hijo
        for (index, hijo) in hijos.enumerated() {
            let esUltimoHijo = index == hijos.count - 1
            hijo.imprimirArbol(nuevoPrefijo, esUltimoHijo)
        }
    }
}
let ejemploFoto = Arbol(dato:"Padre", hijos: [Arbol(dato:"Hija", hijos: [
    Arbol(dato: "Nieta"), Arbol(dato: "Nieto")
])])
ejemploFoto.imprimirArbol()
```

6.swift

```
import Foundation
typealias Litro = Double
typealias KwHora = Double
typealias Gramo = Double
indirect enum Consumo {
    case diesel(Litro)
    case gasolina(Litro)
    case eléctrico(KwHora)
    case hibrido(_ gasolina:Consumo, _ electrico:Consumo)
}
func co2(gasoil l : Litro) -> Gramo {2700*l}
func co2(gasolina l : Litro) -> Gramo {2350 * l}
func co2(kwh:KwHora) -> Gramo {890 * kwh}
func huellaCarbono(_ consumo:Consumo) -> Gramo {
    switch consumo {
        case .diesel(let litros): return(co2(gasoil:litros))
        case .gasolina(let litros): return(co2(gasolina:litros))
        case .eléctrico(let kwh): return(co2(kwh: kwh))
        case .hibrido(let unConsumo,let otroConsumo):
            return (huellaCarbono(unConsumo) +
                huellaCarbono(otroConsumo)
            )
    }
}
let consumoDiesel = Consumo.diesel(10.0)
let consumoGasolina = Consumo.gasolina(8.0)
let consumoElectricidad = Consumo.eléctrico(100.0)
let consumoHíbrido = Consumo.hibrido(consumoGasolina, consumoElectricidad)
print("Huella de carbono para consumo de diesel: \(huellaCarbono(consumoDiesel)) gramos")
print("Huella de carbono para consumo de gasolina: \(huellaCarbono(consumoGasolina)) gramos")
print("Huella de carbono para consumo eléctrico: \(huellaCarbono(consumoElectricidad)) gramos")
print("Huella de carbono para consumo híbrido: \(huellaCarbono(consumoHíbrido)) gramos")
```

7.swift

```
import Foundation
func divisionEntera (_ x:Int,dididoEntre y:Int) -> Int? {
    if y == 0 {
        return nil
    }
    return x / y
}
func imprimeResultado(_ num:Int?) {
    if num == nil {
        print("División por 0")
    }
    else {
        print("res = \(num!)")
    }
    // Otra alternativa para imprimir el resultado
    if let res = num {
        print("res = \(res)")
    }
    else {
        print("División por 0")
    }
}
var res = divisionEntera(20, dididoEntre: 10)
imprimeResultado(res)
var XdDani = divisionEntera(20, dididoEntre: 0)
imprimeResultado(XdDani)
```

8.swift

```
import Foundation
enum Rol {
    case rey, reina, torre, caballo
    case alfil, peon
}
enum Color {case blanca, negra}
struct Pieza {
    let rol: Rol
    let color: Color
}
// Esta nos la dan implementada
func pieza(en:Posicion) -> Pieza? {
    switch en {
        case .a1, .h1: return Pieza(rol: .torre, color: .blanca)
        case .b1, .g1: return Pieza(rol: .caballo, color: .blanca)
        case .c1, .f1: return Pieza(rol: .alfil, color: .blanca)
        case .d1: return Pieza(rol: .rey, color: .blanca)
        case .e1: return Pieza(rol: .reina, color: .blanca)
        case .a2, .b2, .c2, .d2, .e2, .f2, .g2, .h2: return Pieza(rol: .peon, color: .blanca)
        case .a8, .h8: return Pieza(rol: .torre, color: .negra)
        case .b8, .g8: return Pieza(rol: .caballo, color: .negra)
        case .c8, .f8: return Pieza(rol: .alfil, color: .negra)
        case .d8: return Pieza(rol: .rey, color: .negra)
        case .e8: return Pieza(rol: .reina, color: .negra)
        case .a7, .b7, .c7, .d7, .e7, .f7, .g7, .h7: return Pieza(rol: .peon, color: .negra)
    }
}
enum Posicion {
    case a8, b8, c8, d8, e8, f8, g8, h8
    case a7, b7, c7, d7, e7, f7, g7, h7
    case a6, b6, c6, d6, e6, f6, g6, h6
    case a5, b5, c5, d5, e5, f5, g5, h5
    case a4, b4, c4, d4, e4, f4, g4, h4
    case a3, b3, c3, d3, e3, f3, g3, h3
    case a2, b2, c2, d2, e2, f2, g2, h2
    case a1, b1, c1, d1, e1, f1, g1, h1
}
func esPosibleEnroqueCortoBlancas() -> Bool {
    if pieza(en: .f1) == nil  &&  pieza(en: .g1) == nil {
        if let pieza1 = pieza(en: .e1), let pieza2 = pieza(en: .h1) {
            if pieza1.rol == .rey && pieza1.color == .blanca && pieza2.rol == .torre && pieza2.color == .blanca
                return true
            }
        }
    }
    return false
}
```

9.swift

```
import Foundation
// Qué imprime el siguiente código?
func foo (nums: [Int]) -> ([Int], [Int]) {
    return nums.reduce(([], []))
    {(res: ([Int], [Int]), n:Int ) -> ([Int], [Int]) in
                        if !res.0.contains(n) {
                            return (res.0 + [n], res.1)
                        } else {
                            return ((res.0, res.1 + [n]))
                        }
    }
}
print(foo(nums: [1,2,6,3,2,4,3,2]))
// Lo que hace esta FOS es que, recorriendo el array nums, si el elemento no está en el primer
// array dividio entonces se mete en el primer array, y si ese elemento ya está previamente se
// mete en el otro array de la derecha. Básicamente permite evitar tener elementos repetidos en el mismo
// El resultado sería: ([1, 2, 6, 3, 4], [2, 3, 2])
// Lo sabía está BIEN!!!!!!!!!!!!!!
```