



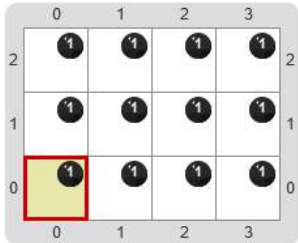
## Examen argentina programa T3

Fundamentos de Programación (Instituto Universitario Escuela Argentina de Negocios)

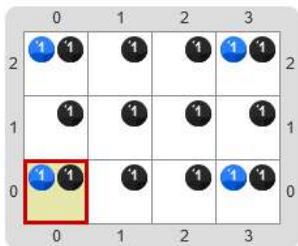
# Ejercicio 1: Ejercicio 1



Una extravagante repostería nos pidió ayuda para decorar su famosa torta rectangular de chocolate 🍰:



Para eso crearemos un programa que ponga confites de color **Azul** en sus extremos de la siguiente forma:



```
1 program {
2   Poner(Azul)
3   Mover(Norte)
4   Mover(Norte)
5   Poner(Azul)
6   repeat(3){
7     Mover(Este)
8   }
9   Poner(Azul)
10  Mover(Sur)
11  Mover(Sur)
12  Poner(Azul)
13 }
```

▶ Enviar

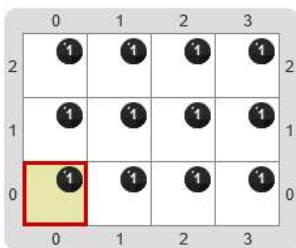
Creá el programa que decore la torta de la forma solicitada. El cabezal comienza en el extremo Sur Oeste y no importa dónde termina.

💡 ¡Dame una pista!

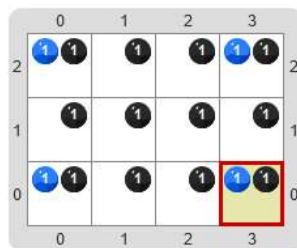
Tené en cuenta que el programa solo debe poner los confites, la torta ya está creada. 😊

✅ ¡Muy bien! Tu solución pasó todas las pruebas

Tablero inicial



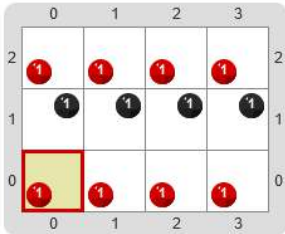
Tablero final





## Ejercicio 2: Ejercicio 2

Un grupo de estudiantes nos solicitó ayuda para armar la bandera de su curso utilizando Gobstones . La bandera tendrá 3 franjas de 4 celdas cada una y quieren probar la combinación de distintos colores. Para ello definiremos un procedimiento que reciba como argumento el color de las franjas superior e inferior y el color de la franja del medio. Por ejemplo, si lo invocamos con los colores Rojo y Negro haciendo `ProbarCombinacionBandera(Rojo, Negro)` la bandera deberá verse así:



Definí el procedimiento `ProbarCombinacionBandera` que reciba dos colores y arme una bandera a partir de ellos. El cabezal comienza en el extremo Sur Oeste y no importa dónde finaliza.

```
1 procedure ProbarCombinacionBandera(color1, color2){
2   repeat(3){
3     Poner(color1)
4     Mover(Este)
5   }
6   Poner(color1)
7   Mover(Norte)
8   repeat(3){
9     Poner(color2)
10    Mover(Oeste)
11  }
12  Poner(color2)
13  Mover(Norte)
14  repeat(3){
15    Poner(color1)
16    Mover(Este)
17  }
18  Poner(color1)
19 }
20
21
```

▶ Enviar

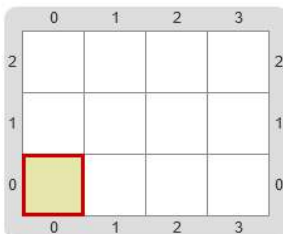
💡 ¡Dame una pista!

✅ ¡Muy bien! Tu solución pasó todas las pruebas

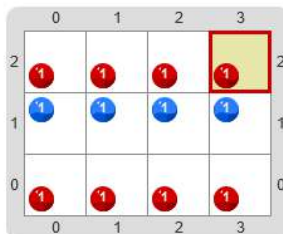
Resultados de las pruebas:



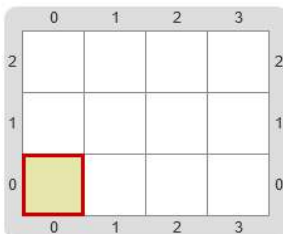
Tablero inicial



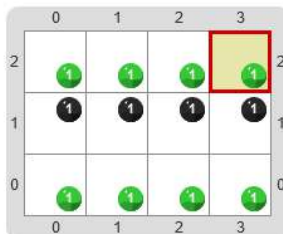
Tablero final



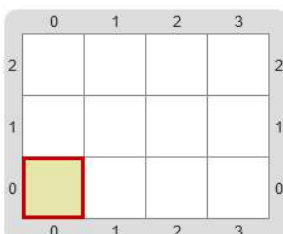
Tablero inicial



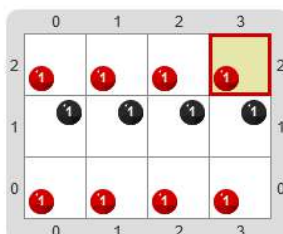
Tablero final



Tablero inicial



Tablero final



# Ejercicio 3: Ejercicio 3

JS

Los colores tienen distintas clasificaciones, dentro de ellas podemos encontrar a los colores primarios: azul, rojo y amarillo 🌈. Para distinguirlos tenemos la función `esUnColorPrimario`:

```
esUnColorPrimario("amarillo")  
true
```

```
esUnColorPrimario("verde")  
false
```

Definí la función `esUnColorPrimario` que recibe un color y nos dice si es un color primario o no.

 Solución  Consola

```
1 function esUnColorPrimario(color){  
2   return (color === "azul") || (color === "rojo")  
   || (color === "amarillo");  
3 }
```

 Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

## Ejercicio 4: Ejercicio 4


Al salir a la calle es recomendable ver la temperatura 🌡, pero no siempre sabemos si está fresco o caluroso. Para ello definiremos una función que resuelva este dilema. Si la temperatura es menor a 17 diremos que está fresco y en caso contrario que está caluroso:

```
comoEsta(27)  
'está caluroso'
```

```
comoEsta(17)  
'está caluroso'
```

```
comoEsta(16)  
'está fresco'
```

Definí la función `comoEsta`.

 Solución  Consola

```
1 function comoEsta(temperatura) {  
2   if (temperatura < 17){  
3     return "está fresco";  
4   } else{  
5     return "está caluroso";  
6   }  
7 }
```

 Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas

## Ejercicio 5: Ejercicio 5


JS

Un vivero quiere saber qué plantas puede trasplantar 🌱. Para ello nos pidió una función que a partir de una lista de alturas de plantas nos diga cuáles tienen más de 25 cms:

```
listasParaTrasplantar([24,30, 10, 25, 26])  
[30, 26]
```

Definí la función `listasParaTrasplantar`.

☒ Solución

 Biblioteca

 Consola

```
1 function listasParaTrasplantar(altura){  
2   let sumatoria = []  
3   for (let cm of altura) {  
4     if (cm > 25){  
5       agregar (sumatoria, cm);  
6     }  
7   }  
8   return sumatoria;  
9 }
```

 Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 6: Ejercicio 6

En una empresa de viajes 🏕️ guardan registro de los viajes realizados por sus clientes de la siguiente forma:

```
let luca = {  
  nombre: "Luca Maggio",  
  destinos: ["Neuquén", "Salta", "Mendoza", "Córdoba"],  
  fechaUltimoViaje: "15/11/2021"  
}  
  
let margarita = {  
  nombre: "Margarita Lopez",  
  destinos: ["Jujuy", "Tucumán"],  
  fechaUltimoViaje: "19/09/2021"  
}
```

Definí la función `resumenDeLosViajes` que permita obtener un resumen de la información registrada. Por ejemplo:

```
resumenDeLosViajes(luca)  
"Luca Maggio realizó su último viaje el 15/11/2021 y en  
total realizó 4 viajes"  
  
resumenDeLosViajes(margarita)  
"Margarita Lopez realizó su último viaje el 19/09/2021 y  
en total realizó 2 viajes"
```

[Solución](#)[Biblioteca](#)[Consola](#)

```
1 function resumenDeLosViajes(cliente){  
2   return cliente.nombre + " realizó su último viaje  
   el " + cliente.fechaUltimoViaje + " y en total  
   realizó " + (longitud(cliente.destinos)) + "  
   viajes"  
3 }
```

[▶ Enviar](#)

✅ ¡Muy bien! Tu solución pasó todas las pruebas

# Ejercicio 7: Ejercicio 7

¡Dejemos atrás a JavaScript para pasar a Ruby! 🐘

La famosa novela gráfica Sandman tendrá pronto su propia serie ⌚. También quieren desarrollar un videojuego de la misma y para ello vamos a modelar a su protagonista: `Sandman`. De él sabemos que:

- tiene un descanso inicial de 2;
- puede `descansar!` una cantidad de minutos y aumentar su descanso en esa cantidad;
- si tiene menos de 31 de descanso diremos que `necesita_reposo?`.

Definí en Ruby, el objeto `Sandman` que tenga un atributo `@descanso` con su getter. El objeto entiende los mensajes `descansar!` (que recibe minutos como argumento) y `necesita_reposo?`. No te olvides de inicializar el atributo `@descanso` con el valor correspondiente.

🔍 Solución ➤ Consola

```
1 module Sandman
2   @descanso = 2
3
4   def self.descanso
5     @descanso
6   end
7
8   def self.descansar!(minutos)
9     @descanso += minutos
10  end
11
12  def self.necesita_reposo?
13    @descanso < 31
14  end
15 end
```

▶ Enviar

✅ ¡Muy bien! Tu solución pasó todas las pruebas



## Ejercicio 8: Ejercicio 8



En un curso tenemos un conjunto de estudiantes, a la hora de cerrar las actas es necesario saber quiénes aprobaron . Teniendo en cuenta que cada estudiante sabe responder al mensaje `aprobo_materia?` ...

Solución

Consola

Definí en Ruby el método `quienes_aprobaron` que retorne qué estudiantes aprobaron del `Estudiantado`.

```
1 module Estudiantado
2   @estudiantes = [May, Gus, Ro, Agus, Lu, Ale]
3
4   def self.quienes_aprobaron
5     @estudiantes.select{ |estudiantes|
6       estudiantes.aprobo_materia?}
7   end
8 end
```

Enviar

¡Muy bien! Tu solución pasó todas las pruebas

## Ejercicio 9: Ejercicio 9



A la hora de viajar sabemos que las instancias de la clase `Persona` usan su boleto

Actualmente tenemos dos tipos de boletos:

- `BoletoDescartable`: solo se pueden usar una vez y tienen un atributo `@usado`;
- `BoletoConSaldo`: tiene un `@saldo` que disminuye 19 con cada viaje.

Definí el método `viajar!` en la clase `Persona` y el método `usar!` en los distintos tipos de boleto.

**Solución** >\_ Consola

```
1 class Persona
2   def initialize(un_boleto)
3     @boleto = un_boleto
4   end
5
6   def viajar!
7     @boleto.usar!
8   end
9 end
10
11 class BoletoDescartable
12   def initialize()
13     @usado = false
14   end
15
16   def usar!
17     @usado = true
18   end
19 end
20
21 class BoletoConSaldo
22   def initialize(un_saldo)
23     @saldo = un_saldo
24   end
25
26
27   def usar!
28     @saldo -= 19
29   end
30 end
```

▶ Enviar

✓ ¡Muy bien! Tu solución pasó todas las pruebas