


	<p align="center">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
Aprobación: 2022/03/01	Código: GUIA-PRLE-001	Página: 1

INFORME DE LABORATORIO

INFORMACIÓN BÁSICA					
ASIGNATURA:	Tecnología de objetos				
TÍTULO DE LA PRÁCTICA:	Fundamentos de Java y transición desde C++/Python				
NÚMERO DE PRÁCTICA:	02	AÑO LECTIVO:	2025-B	NRO. SEMESTRE:	VI
FECHA DE PRESENTACIÓN	18/09/2025	HORA DE PRESENTACIÓN	23:59		
INTEGRANTE (s): Alexander Valdiviezo Tovar				NOTA:	
DOCENTE(s): MSc.Edith Cano					

SOLUCIÓN Y RESULTADOS
<p>I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS</p> <p>Vuelto. Contando el vuelto o cambio. Escriba una función recursiva que cuente de cuántas maneras diferentes puede dar cambio para una determinada cantidad y de acuerdo a una lista de denominaciones de monedas. Por ejemplo, hay 3 maneras de dar cambio si la cantidad=4 si tienes monedas con denominación 1 y 2: 1+1+1+1, 1+1+2, 2+2</p> <p>Realiza este ejercicio implementando la función countChange en Scala. Esta función toma una cantidad a cambiar y una lista de denominaciones únicas para las monedas. Su definición es la siguiente: def countChange(money: Int, coins: List[Int]): Int Puedes usar las funciones isEmpty, head y tail en la lista de monedas enteras.</p> <p>Solución:</p> <p>Casos base:</p> <ul style="list-style-type: none"> • money == 0: hemos encontrado una forma válida → retorna 1. • money < 0: el cambio se pasó → retorna 0. • coins.isEmpty: no hay más monedas para usar → retorna 0. <p>Recursión:</p> <ul style="list-style-type: none"> • countChange(money - coins.head, coins): incluye la moneda actual (coins.head) y sigue restando de la cantidad. • countChange(money, coins.tail): ignora la moneda actual y prueba con el resto.

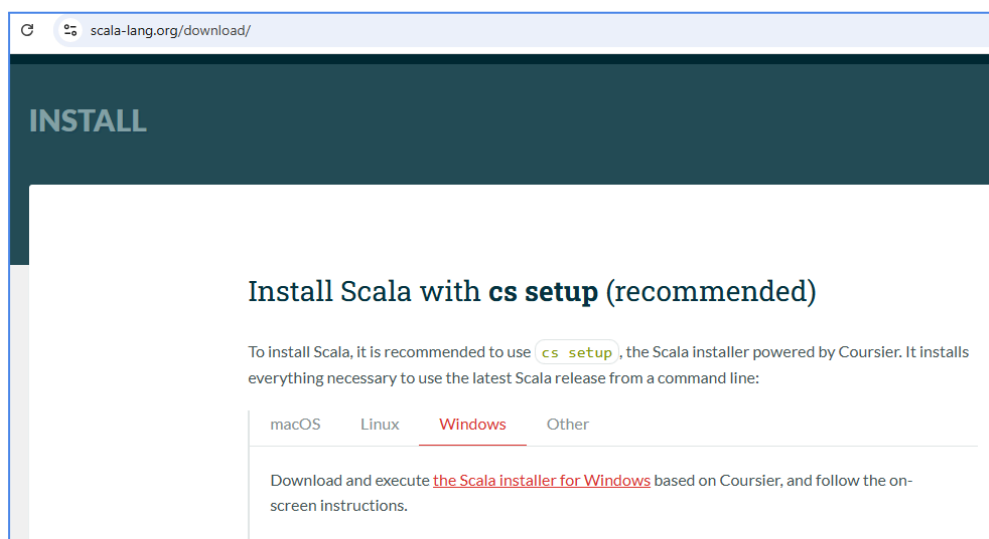
	<p>UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p>Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 2</p>

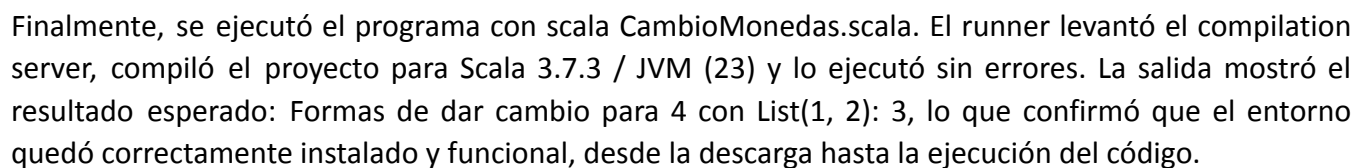
```
object CambioMonedas {
  def countChange(money: Int, coins: List[Int]): Int = {
    if (money == 0) 1
    else if (money < 0 || coins.isEmpty) 0
    else {
      // Caso 1: usar la moneda actual (head)
      val withHead = countChange(money - coins.head, coins)
      // Caso 2: no usar la moneda actual (pasar a tail)
      val withoutHead = countChange(money, coins.tail)


      withHead + withoutHead
    }
  }

  def main(args: Array[String]): Unit = {
    println(countChange(4, List(1, 2))) // Resultado esperado: 3
    println(countChange(10, List(2, 5, 3, 6))) // Otro ejemplo: 5 maneras
  }
}
```

Descargamos e instalamos el paquete de Scala (vía Coursier). El instalador verificó que hubiera una JVM disponible y detectó correctamente Java 23 en C:\Program Files\Java\jdk-23. También comprobó que la ruta ~\AppData\Local\Coursier\data\bin estuviera en el PATH y confirmó la instalación de las aplicaciones estándar: ammonite, cs, coursier, scala, scalac, scala-cli, sbt, sbt-n, scalafmt. Tras estas validaciones, el instalador pidió presionar ENTER para continuar.





	<p style="text-align: center;">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p style="text-align: center;">Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 4</p>

```
E:\to\scala>scala CambioMonedas.scala
Starting compilation server
Compiling project (Scala 3.7.3, JVM (23))
Compiled project (Scala 3.7.3, JVM (23))
Formas de dar cambio para 4 con List(1, 2): 3
E:\to\scala>
```

II. CUESTIONARIO

1. Explique el caso base implementado.

Tenemos tres casos base:

money == 0 → retornar 1

- a. Significa que ya encontramos una forma válida de dar el cambio.
- b. Ejemplo: si querías cambiar 4 y llegaste a 0 usando 2+2, eso cuenta como 1 solución.

money < 0 → retornar 0

- c. Quiere decir que nos pasamos de la cantidad (por ejemplo, intentamos formar 4 con 2+2+2 = 6).
- d. No es una solución válida, por eso se descarta.

coins.isEmpty y money > 0 → retornar 0

- e. Significa que ya no quedan monedas para usar, pero aún falta cubrir parte del monto.
- f. Ejemplo: querías formar 3 con monedas [2]. No puedes, entonces no hay solución.

2. ¿Este problema se resuelve mejor en el modelo de programación funcional?

Sí: este problema encaja muy bien en programación funcional (PF). Es naturalmente declarativo y recursivo (“número de formas = usar la moneda actual + no usarla”), con subproblemas idénticos (mismos money y subconjunto de coins). En PF, la inmutabilidad y la transparencia referencial facilitan razonar sobre la corrección, probar casos y componer soluciones. Además, la definición queda cercana a las ecuaciones matemáticas del problema.

3. ¿Qué beneficios se podrían obtener para este problema utilizando memoización?

La memoización aporta un salto grande: la versión inicial hace llamadas repetidas a los mismos subproblemas (explosión exponencial). Con memoización (top-down) se pasa de ~exponencial a $O(M \times N)$, donde $M = \text{coins.length}$ y $N = \text{money}$, porque solo se resuelve una vez cada estado (money, i) (i = índice de moneda). En práctica, para cantidades medianas/grandes, la diferencia es enorme (milisegundos vs. segundos/minutos). También se reduce el consumo de energía y

	<p style="text-align: center;">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p style="text-align: center;">Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 5</p>

hace el programa más predecible. Como extra, al ser funciones puras, memoizar es seguro (mismo input → mismo output).

Una versión funcional con memoización en Scala (top-down), clave (money, i) para evitar depender de la identidad de la lista se presenta a continuación:

```
object CambioMonedasMemo {
  def countChange(amount: Int, coins: List[Int]): Int = {
    val arr = coins.toArray
    import scala.collection.mutable

    val memo = mutable.Map.empty[(Int, Int), Int]

    def go(money: Int, i: Int): Int = {
      if (money == 0) 1
      else if (money < 0 || i == arr.length) 0
      else memo.getOrElseUpdate((money, i),
        go(money - arr(i), i) + // usar moneda i (repetible)
        go(money, i + 1) // saltar a la siguiente moneda
      )
    }

    go(amount, 0)
  }

  def main(args: Array[String]): Unit = {
    val monedas = List(1, 2)
    val cantidad = 4
    println(s"Formas de dar cambio para $cantidad con $monedas: " +
      countChange(cantidad, monedas))
  }
}
```

```
E:\to\scala>scala CambioMonedasMemo.scala
Compiling project (Scala 3.7.3, JVM (23))
Compiled project (Scala 3.7.3, JVM (23))
Formas de dar cambio para 4 con List(1, 2): 3
```

III. CONCLUSIONES

- El problema de calcular las combinaciones de cambio encaja de manera natural con el paradigma de programación funcional. La estructura recursiva, el uso de funciones puras y la ausencia de efectos secundarios permiten expresar la solución de forma más clara y cercana al modelo matemático, lo que facilita tanto la comprensión como la verificación del código.
- En segundo lugar, la incorporación de la memoización transforma la eficiencia del algoritmo. Al almacenar los resultados de subproblemas que se repiten, se evita recalcularlos y se logra

	<p align="center">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p align="center">Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p align="center">Código: GUIA-PRLE-001</p>	<p align="right">Página: 6</p>

reducir significativamente la complejidad temporal. Esto se traduce en un mejor rendimiento, especialmente cuando la cantidad de dinero o la lista de monedas es grande.

- Por último, la unión de un diseño funcional con técnicas de optimización como la memoización permite alcanzar un equilibrio entre claridad y eficiencia. El código mantiene una estructura sencilla, legible y fácil de mantener, mientras que al mismo tiempo se obtiene un comportamiento mucho más rápido y escalable, lo que lo convierte en un enfoque recomendable para este tipo de problemas combinatorios.

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA

<https://docs.scala-lang.org/es/tour/tour-of-scala.html>
<https://www.geeksforgeeks.org/scala/scala-programming-language/>
<https://onecompiler.com/scala>
<https://www.programiz.com/scala/online-compiler/>
<https://www.jdoodle.com/compile-scala-online>
<https://paiza.io/es/projects/new>