Estructuras de datos realizado por Luis Alejandro Contreras Sanchez

1. Introducción a las estructuras de datos en Kotlin
a. ¿Qué son las estructuras de datos y para qué se utilizan?
En el ámbito de la informática, las estructuras de datos son aquellas que nos permiten, como desarrolladores, organizar la información de manera eficiente, y en definitiva diseñar la solución correcta para un determinado problema.
Las estructuras de datos se utilizan en todo tipo de aplicaciones informáticas, desde bases de datos y sistemas operativos hasta juegos y aplicaciones web.

b. Ventajas de utilizar estructuras de datos en Kotlin
Acceso y manipulación de datos: Las estructuras de datos permiten el acceso y manipulación eficiente de los datos.
Almacenamiento eficiente de datos: Las estructuras de datos son diseñadas para minimizar la cantidad de espacio de almacenamiento que se necesita para guardar los datos.
Eficiencia en el procesamiento de datos: Las estructuras de datos pueden ser diseñadas para optimizar la eficiencia del procesamiento de datos.
Manejo de datos complejos: Las estructuras de datos pueden ser utilizadas para representar datos complejos.

c. Diferencias entre las estructuras de datos en Kotlin y Java
Kotlin es más conciso que Java, se calcula que con este lenguaje, se reduce un 40% de líneas de código en comparación con Java.
Mutabilidad: En Kotlin, las estructuras de datos pueden ser inmutables o mutables, lo que significa que sus valores pueden cambiar. En Java, la mayoría de las estructuras de datos son mutables por defecto.
Null safety: En Kotlin, los valores pueden ser nulos o no nulos, y el compilador se asegura de que no se produzcan errores de nulidad en tiempo de ejecución. En Java, los valores pueden ser nulos, lo que puede resultar en errores de nulidad en tiempo de ejecución.
Listas: En Kotlin, existe una clase "List" que es inmutable por defecto. En Java, la clase "ArrayList" es mutable y se utiliza comúnmente para representar una lista.
Conjuntos: En Kotlin, existe una clase "Set" que es inmutable por defecto. En Java, la clase "HashSet" es mutable y se utiliza comúnmente para representar un conjunto.
Mapas: En Kotlin, existe una clase "Map" que es inmutable por defecto. En Java, la clase "HashMap" es mutable y se utiliza comúnmente para representar un mapa.

2. Arreglos en Kotlin
a. ¿Qué es un arreglo?
Un arreglo es una estructura con valores de datos, que están almacenados de forma contigua en memoria. Todos los elementos son referenciados por un mismo nombre y tienen el mismo tipo de dato. Los elementos estarán indexados tomando como base el 0 y el tamaño declarado del arreglo será fijo.
b. Creación de arreglos en Kotlin
Para declarar un arreglo en Kotlin se usa la función arrayOf, y se indican los elementos del arreglo como argumentos a la función.

Considera que necesitas almacenar el valor de los ingresos de 12 meses del usuario. En vez de crear 12 variables para cada mes (`januaryIncome` , `febrauryIncome` , etc.) optas por crear un arreglo con estos elementos:

```
val income = arrayOf<Double>(
    0.5, 2.5, 4.0, 5.0,
    4.5, 6.0, 7.6, 8.0,
    5.0, 6.4, 4.0, 9.1
)
```

Al igual que cualquier declaración de instancias, puedes omitir el tipo parametrizado `<Double>` porque el compilador de Kotlin puede inferirlo.

Creación de arreglos con el operador de array:

```
val arr = arrayOf(1, 2, 3, 4, 5)
```

c. Accediendo a los elementos de un arreglo

En un arreglo, a cada elemento se accede mediante un índice numérico que indica la posición del elemento en el arreglo. El primer elemento del arreglo se accede con el índice 0, el segundo elemento con el índice 1, y así sucesivamente.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val primerNumero = numeros[0] // primerNumero es 1
```

d. Modificando los elementos de un arreglo

Para modificar los elementos de un arreglo en Kotlin, se utiliza la sintaxis de corchetes [ ] con el índice del elemento para acceder al elemento que se desea modificar, y luego se le asigna un nuevo valor.

```
var numeros = arrayOf(1, 2, 3, 4, 5)
numeros[0] = 10 // Modificar el primer elemento a 10
```

e. Recorriendo un arreglo

Recorrer un arreglo con un ciclo for:

```
val numeros = arrayOf(1, 2, 3, 4, 5)
for (numero in numeros) {
    println(numero)
}
```

```
1
2
3
4
5
```

```
val numeros = arrayOf(1, 2, 3, 4, 5)
numeros.forEach { numero ->
    println(numero)
}
```

```
1
2
3
4
5
```

f. Funciones útiles para trabajar con arreglos en Kotlin

Size: Devuelve el número de elementos en un arreglo.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
println(numeros.size) // Imprime 5
```

Get: Devuelve el elemento en el índice especificado.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
println(numeros.get(2)) // Imprime 3
```

Set: Establece el elemento en el índice especificado.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
numeros.set(2, 10)
println(numeros.contentToString()) // Imprime [1, 2, 10, 4, 5]
```

Contains: Devuelve true si el arreglo contiene el elemento especificado, de lo contrario, devuelve false.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
println(numeros.contains(3)) // Imprime true
```

IndexOf: Devuelve el índice de la primera aparición del elemento especificado en el arreglo, o -1 si el elemento no está presente.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
println(numeros.indexOf(4)) // Imprime 3
```

CopyOf: Devuelve una copia del arreglo original con la longitud especificada.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val copiaNumeros = numeros.copyOf(3)
println(copiaNumeros.contentToString()) // Imprime [1, 2, 3]
```

Slice: Devuelve una lista que contiene los elementos del arreglo especificados por los índices.

```
val numeros = arrayOf(1, 2, 3, 4, 5)
val subNumeros = numeros.sliceArray(1..3)
println(subNumeros.contentToString()) // Imprime [2, 3, 4]
```

3. Listas en Kotlin

a. ¿Qué es una lista?

Una lista es una estructura de datos que contiene una colección de elementos secuenciales. Cada elemento en la lista se identifica mediante un índice o posición, y las operaciones principales en la lista incluyen agregar, eliminar y obtener elementos.

b. Creación de listas en Kotlin

Crear una lista con elementos utilizando la función listOf():

```
val numeros = listOf(1, 2, 3, 4, 5)
```

Crear una lista vacía y agregar elementos uno por uno:

```
val numeros = mutableListOf<Int>()
numeros.add(1)
numeros.add(2)
numeros.add(3)
```

Crear una lista con elementos utilizando el operador de rango:

```
val numeros = (1..5).toList()
```

Crear una lista a partir de un arreglo:

```
val arreglo = arrayOf(1, 2, 3, 4, 5)
val numeros = arreglo.toList()
```

c. Accediendo a los elementos de una lista

En Kotlin, se pueden acceder a los elementos de una lista utilizando el operador de indexación “[ ]” según la posición de cada elemento.

```
val numeros = listOf(1, 2, 3, 4, 5)
val segundoNumero = numeros[1]
```

// 2

También es posible acceder a un rango de elementos en la lista utilizando la función “subList()”. Esta función toma como parámetros el índice inicial y el índice final del rango de elementos que se desea obtener, y devuelve una sublista que contiene sólo los elementos dentro del rango.

```
val numeros = listOf(1, 2, 3, 4, 5)
val primerosTresNumeros = numeros.subList(0, 3)
```

```
[1, 2, 3]
```

d. Modificando los elementos de una lista

Para modificar un elemento en particular de una lista, primero es necesario obtener una referencia al elemento utilizando el operador de indexación, y luego se puede modificar el valor del elemento utilizando la asignación normal.

```
val numeros = mutableListOf(1, 2, 3, 4, 5)
numeros[1] = 10
```

```
[1, 10, 3, 4, 5]
```

e. Recorriendo una lista

Usando el bucle for:

```
val numeros = listOf(1, 2, 3, 4, 5)
for (numero in numeros) {
    println(numero)
}
```

```
1
2
3
4
5
```

Usando la función forEach():

```
val numeros = listOf(1, 2, 3, 4, 5)
numeros.forEach { numero ->
    println(numero)
}
```

```
1
2
3
4
5
```

Usando un índice y el operador de indexación:

```
val numeros = listOf(1, 2, 3, 4, 5)
for (i in numeros.indices) {
    println(numeros[i])
}
```

```
1
2
3
4
5
```

También se pueden utilizar otras funciones como map(), filter(), y reduce() para realizar operaciones más complejas en los elementos de una lista.

f. Funciones útiles para trabajar con listas en Kotlin

Size: Devuelve el número de elementos en la lista.

```
val numeros = listOf(1, 2, 3, 4, 5)
println(numeros.size) // Imprime: 5
```

isEmpty: Devuelve true si la lista no contiene elementos.

```
val numeros = emptyList<Int>()
println(numeros.isEmpty()) // Imprime: true
```

Contains: Devuelve true si la lista contiene un elemento específico.

```
val numeros = listOf(1, 2, 3, 4, 5)
println(numeros.contains(3)) // Imprime: true
println(numeros.contains(6)) // Imprime: false
```

indexOf: Devuelve el índice de la primera ocurrencia de un elemento en la lista.

```
val numeros = listOf(1, 2, 3, 4, 5)
println(numeros.indexOf(3)) // Imprime: 2
```

sorted: Devuelve una lista ordenada en orden ascendente.

```
val numeros = listOf(5, 3, 1, 4, 2)
val numerosOrdenados = numeros.sorted()
println(numerosOrdenados) // Imprime: [1, 2, 3, 4, 5]
```

max y min: Devuelven el valor máximo y mínimo en la lista.

```
val numeros = listOf(5, 3, 1, 4, 2)
println(numeros.max()) // Imprime: 5
println(numeros.min()) // Imprime: 1
```

filter: Devuelve una lista de elementos que cumplen con una condición dada.

```
val numeros = listOf(1, 2, 3, 4, 5)
val numerosPares = numeros.filter { it % 2 == 0 }
println(numerosPares) // Imprime: [2, 4]
```

map: Devuelve una lista de elementos transformados de acuerdo con una función dada.

```
val numeros = listOf(1, 2, 3, 4, 5)
val numerosCuadrados = numeros.map { it * it }
println(numerosCuadrados) // Imprime: [1, 4, 9, 16, 25]
```

4. Conjuntos en Kotlin

a. ¿Qué es un conjunto?

Un conjunto (también conocido como set) es una colección de elementos sin orden y sin elementos repetidos. Es decir, un conjunto es una estructura de datos que almacena elementos de tal manera que cada elemento aparece una sola vez en el conjunto, y no hay ningún orden específico en el que se almacenan los elementos.

b. Creación de conjuntos en Kotlin

En Kotlin, se pueden crear conjuntos utilizando cualquiera de las implementaciones de la interfaz Set. Las implementaciones más comunes de Set son HashSet, TreeSet y LinkedHashSet.

HashSet: Es la implementación más común de Set. Los elementos se almacenan en un orden no específico y no se garantiza que el orden se mantenga.

```
val conjunto = HashSet<String>()
conjunto.add("Manzana")
conjunto.add("Banana")
conjunto.add("Naranja")
println(conjunto) // Imprime: [Banana, Manzana, Naranja]
```

TreeSet: Es una implementación de Set que almacena elementos en un orden ordenado. El orden se determina utilizando un árbol binario y se puede definir utilizando un comparador.

```
val conjunto = TreeSet<String>()
conjunto.add("Manzana")
conjunto.add("Banana")
conjunto.add("Naranja")
println(conjunto) // Imprime: [Banana, Manzana, Naranja]
```

LinkedHashSet: Es una implementación de Set que almacena elementos en el orden en que se agregaron.

```
val conjunto = LinkedHashSet<String>()
conjunto.add("Manzana")
conjunto.add("Banana")
conjunto.add("Naranja")
println(conjunto) // Imprime: [Manzana, Banana, Naranja]
```

c. Accediendo a los elementos de un conjunto

En Kotlin, no se puede acceder a los elementos de un conjunto por índice como se hace en un arreglo o una lista, ya que los conjuntos no mantienen un orden específico. En su lugar, se utiliza el método contains() para verificar si un elemento está presente en el conjunto.

```
val conjunto = setOf("Manzana", "Banana", "Naranja")
println(conjunto.contains("Manzana")) // Imprime: true
```

d. Modificando los elementos de un conjunto

En Kotlin, los conjuntos son inmutables, lo que significa que no se pueden modificar una vez que se han creado. Sin embargo, se puede crear un nuevo conjunto a partir del original con elementos agregados o eliminados.

Agregar elementos a un conjunto:

```kotlin
val conjunto1 = setOf("Manzana", "Banana", "Naranja")
val conjunto2 = conjunto1.plus("Mango")
println(conjunto1) // Imprime: [Manzana, Banana, Naranja]
println(conjunto2) // Imprime: [Manzana, Banana, Naranja, Mango]
```

e. Recorriendo un conjunto

Recorrer un conjunto con un bucle for:

```kotlin
val conjunto = setOf("Manzana", "Banana", "Naranja")

for (elemento in conjunto) {
    println(elemento)
}
```

```
Manzana
Banana
Naranja
```

Recorrer un conjunto con un iterador:

```kotlin
val conjunto = setOf("Manzana", "Banana", "Naranja")
val iterador = conjunto.iterator()

while (iterador.hasNext()) {
    val elemento = iterador.next()
    println(elemento)
}
```

```
Manzana
Banana
Naranja
```

forEach() para recorrer un conjunto en Kotlin:

```kotlin
val conjunto = setOf("Manzana", "Banana", "Naranja")

conjunto.forEach { elemento ->
    println(elemento)
}
```

```
Manzana
Banana
Naranja
```

f. Funciones útiles para trabajar con conjuntos en Kotlin

La función “union()” se utiliza para crear un nuevo conjunto que contiene todos los elementos de dos conjuntos.

```kotlin
val conjunto1 = setOf(1, 2, 3)
val conjunto2 = setOf(3, 4, 5)
val union = conjunto1.union(conjunto2)

println(union) // imprime [1, 2, 3, 4, 5]
```

La función “intersect()” se utiliza para crear un nuevo conjunto que contiene solo los elementos que están presentes en ambos conjuntos.

```kotlin
val conjunto1 = setOf(1, 2, 3)
val conjunto2 = setOf(3, 4, 5)
val interseccion = conjunto1.intersect(conjunto2)

println(interseccion) // imprime [3]
```

La función “subtract()” se utiliza para crear un nuevo conjunto que contiene sólo los elementos que están presentes en el primer conjunto pero no en el segundo conjunto.

```
val conjunto1 = setOf(1, 2, 3)
val conjunto2 = setOf(3, 4, 5)
val diferencia = conjunto1.subtract(conjunto2)

println(diferencia) // imprime [1, 2]
```

5. Mapas en Kotlin

a. ¿Qué es un mapa?

Un mapa, también conocido como diccionario, es una estructura de datos que permite asociar claves con valores. En un mapa, cada clave debe ser única y está asociada con un valor.

b. Creación de mapas en Kotlin

En Kotlin, los mapas se representan mediante la interfaz Map. Existen varias implementaciones de esta interfaz, incluyendo HashMap y LinkedHashMap.

Crear mapas utilizando la función “mapOf”:

```
val myMap = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Crear un mapa mutable:

```
val prices = mutableMapOf(
    "producto1" to 10.0,
    "producto2" to 20.0,
    "producto3" to 30.0
)
```

```
// Crear un mapa vacío
val emptyMap = emptyMap<String, Int>()
```

c. Accediendo a los elementos de un mapa

Se puede acceder a los elementos de un mapa utilizando su clave.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

// Acceder al valor de la clave "red"
val redCode = colorCodes["red"] // "#FF0000"
```

d. Modificando los elementos de un mapa

Se pueden modificar los elementos de un mapa mediante la asignación de un nuevo valor a una clave existente o mediante la inserción de una nueva clave con su respectivo valor.

```
val colorCodes = mutableMapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

// Modificar el valor de la clave "green"
colorCodes["green"] = "#008000"

// Insertar una nueva clave con su valor
colorCodes["yellow"] = "#FFFF00"
```

e. Recorriendo un mapa

En Kotlin, se puede recorrer un mapa utilizando un bucle for o mediante la función de orden superior forEach o forEachIndexed.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

// Usando un bucle for
for ((colorName, colorCode) in colorCodes) {
    println("$colorName: $colorCode")
}

// Usando la función forEach
colorCodes.forEach { (colorName, colorCode) ->
    println("$colorName: $colorCode")
}

// Usando la función forEachIndexed
colorCodes.forEachIndexed { index, (colorName, colorCode) ->
    println("$index: $colorName - $colorCode")
}
```

```
red: #FF0000
green: #00FF00
blue: #0000FF
```

f. Funciones útiles para trabajar con mapas en Kotlin

getOrDefault: Esta función devuelve el valor asociado con la clave especificada si existe en el mapa, de lo contrario devuelve un valor predeterminado.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

val colorCode = colorCodes.getOrDefault("black", "#000000")
println("Black color code: $colorCode")
```

```
Black color code: #000000
```

getOrElse: Esta función devuelve el valor asociado con la clave especificada si existe en el mapa, de lo contrario se evalúa una expresión lambda y se devuelve el resultado.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

val colorCode = colorCodes.getOrElse("black") { "#000000" }
println("Black color code: $colorCode")
```

```
Black color code: #000000
```

mapKeys y mapValues: Estas funciones transforman las claves o valores del mapa mediante una expresión lambda.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

val capitalizedColors = colorCodes.mapKeys { it.key.capitalize() }
val lowerCaseCodes = colorCodes.mapValues { it.value.toLowerCase() }

println("Capitalized colors: $capitalizedColors")
println("Lowercase color codes: $lowerCaseCodes")
```

```
Capitalized colors: {Red=#FF0000, Green=#00FF00, Blue=#0000FF}
Lowercase color codes: {red=#ff0000, green=#00ff00, blue=#0000ff}
```

filterKeys y filterValues: Estas funciones filtran las entradas del mapa según las claves o valores que satisfacen una expresión lambda.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

val filteredColors = colorCodes.filterKeys { it.length == 3 }
val brightColors = colorCodes.filterValues { it.startsWith("#FF") }

println("Filtered colors: $filteredColors")
println("Bright colors: $brightColors")
```

```
Filtered colors: {red=#FF0000}
Bright colors: {red=#FF0000}
```

toList: Esta función convierte el mapa en una lista de pares clave-valor.

```
val colorCodes = mapOf(
    "red" to "#FF0000",
    "green" to "#00FF00",
    "blue" to "#0000FF"
)

val colorList = colorCodes.toList()
println("Color list: $colorList")
```

```
Color list: [(red, #FF0000), (green, #00FF00), (blue, #0000FF)]
```

6. Pares en Kotlin

a. ¿Qué es un par?

Un par es una estructura de datos que almacena dos valores relacionados entre sí. También se conoce como una tupla de dos elementos. Los pares son muy útiles cuando se necesita agrupar dos valores para poder manipularlos juntos en una sola unidad, en lugar de tratarlos por separado.

b. Creación de pares en Kotlin

La forma más común de crear un par es utilizando la clase predefinida Pair.

```
// Creando un par de dos strings
val pair1 = Pair("Hola", "Mundo")
println(pair1.first) // Imprime "Hola"
println(pair1.second) // Imprime "Mundo"

// Creando un par de un entero y un string
val pair2 = Pair(123, "Kotlin")
println(pair2.first) // Imprime 123
println(pair2.second) // Imprime "Kotlin"
```

También se puede utilizar la función to para crear un par:

```
// Creando un par de un string y un entero utilizando la función "to"
val pair4 = "Kotlin" to 2022
println(pair4.first) // Imprime "Kotlin"
println(pair4.second) // Imprime 2022
```

c. Accediendo a los elementos de un par

En Kotlin, se puede acceder a los elementos de un par utilizando las propiedades first y second. Estas propiedades representan el primer y segundo elemento del par, respectivamente.

```kotlin
val pair5 = "Kotlin" to true
println(pair5.first) // Imprime "Kotlin"
println(pair5.second) // Imprime true
```

d. Modificando los elementos de un par

En Kotlin, los pares son objetos inmutables, lo que significa que una vez creados, sus elementos no se pueden modificar. Si se desea cambiar los valores de un par, es necesario crear un nuevo par con los valores actualizados.

```kotlin
val pair1 = Pair("Hola", "Mundo")

// Creando un nuevo par con el segundo elemento actualizado
val pair2 = pair1.copy(second = "Kotlin")

// Imprimiendo el primer y segundo elemento del nuevo par
println(pair2.first) // Imprime "Hola"
println(pair2.second) // Imprime "Kotlin"
```

e. Recorriendo un par

En Kotlin, los pares son objetos inmutables y solo tienen dos elementos, por lo que no se puede recorrer un par como se hace con una lista o un arreglo. Sin embargo, se pueden acceder a sus elementos de forma individual utilizando las propiedades first y second.

```kotlin
val pair = Pair(42, "Kotlin")

// Imprimiendo el contenido del par utilizando plantillas de cadena
println("El par contiene los elementos: ${pair.first} y ${pair.second}")
```

```
El par contiene los elementos: 42 y Kotlin
```

f. Funciones útiles para trabajar con pares en Kotlin

first: Propiedad que devuelve el primer elemento del par.
second: Propiedad que devuelve el segundo elemento del par.
component1: Función que devuelve el primer elemento del par.
component2: Función que devuelve el segundo elemento del par.
toString: Función que devuelve una cadena que representa el par en forma de (primer elemento, segundo elemento).
copy: Función que devuelve una copia del par con los mismos elementos.
equals: Función que compara si dos pares son iguales (es decir, si tienen los mismos elementos).
hashCode: Función que devuelve un valor hash del par.

7. Prácticas de estructuras de datos en Kotlin

a. Ejercicios prácticos para aplicar los conceptos aprendidos

1. Encontrar el número más grande en un arreglo
2. Crear un conjunto de números y eliminar todos los números impares.
3. Crear un mapa de nombres y edades y mostrar los nombres de las personas mayores de 18 años.
4. Crear un par de coordenadas y mostrar el valor de cada componente.

5. Crear una lista de cadenas y mostrar la longitud de la cadena más larga.

b. Solución a los ejercicios prácticos

1.

```
fun main(){
    encontrarMaximo(arrayOf(2,5,6,8,23,15))
}
fun encontrarMaximo(arr: Array<Int>){
    var maximo = arr[0]
    for (i in 1 until arr.size) {
        if (arr[i] > maximo) {
            maximo = arr[i]
        }
    }
    println(maximo)
}
```

```
23
```

2.

```
fun main() {
    val nums = setOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    val evens = nums.filter { it % 2 == 0 }
    println("Los números pares son $evens")
}
```

```
Los números pares son [2, 4, 6, 8, 10]
```

3.

```
fun main() {
    val people = mapOf("Juan" to 25, "Maria" to 17, "Pedro" to 20, "Luisa" to 18)
    val adults = people.filterValues { it >= 18 }.keys
    println("Los adultos son $adults")
}
```

```
Los adultos son [Juan, Pedro, Luisa]
```

4.

```
fun main() {
    val coords = Pair(3.5, 4.8)
    val x = coords.first
    val y = coords.second
    println("Las coordenadas son ($x, $y)")
}
```

```
Las coordenadas son (3.5, 4.8)
```

5.

```kotlin
fun main() {
    val strings = listOf("hola", "mundo", "este", "es", "un", "ejemplo")
    var maxLength = 0
    for (string in strings) {
        if (string.length > maxLength) {
            maxLength = string.length
        }
    }
    println("La longitud de la cadena más larga es $maxLength")
}
```

```
La longitud de la cadena más larga es 7
```