Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información

Proyecto 1 - Abril-Julio 2022

IMPLEMENTACIÓN DEL JUEGO WORDLE EN EL LENGUAJE DE PROGRAMACIÓN HASKELL

Adelina Figueira
15-10484

Luis García
15-10540

20/07/2022

Implementación del juego Wordle en el lenguaje de programación Haskell

20/07/2022

Índice

1. Introducción

El juego Wordle es una versión de Toros y Vacas que se creó en 2021. Existen una gran variedad de versiones del juego y en este proyecto se realizará un juego de Toros y Vacas o Wordle en el lenguaje de programación Haskell con dos modos de juego incluidos.
El modo mente maestra es aquel en el que la computadora elige una palabra y el usuario debe adivinarla, introduciendo las palabras que desee y el sistema le indicará los toros y las vacas.
El modo descifrador es aquel en el que la computadora debe adivinar una palabra que decida el usuario. La computadora muestra una palabra inicial y el usuario debe añadir los toros y vacas dados para que la computadora pueda producir una nueva palabra y así continuar la adivinación.
Para el modo mente maestra se busca encontrar la combinación de toros y vacas para una entrada dada por el usuario y una palabra elegida al azar de la lista de palabras.
Mientras que para el modo descifrador se utilizará un árbol minimax que consiga la palabra mínima que pueda satisfacer las condiciones deseadas.
En este informe hablaremos acerca del diseño de la solución para este juego en Haskell, cómo se realizaron las operaciones para cada uno de los modos. Además se mencionarán los módulos que existen dentro del programa, las características de Haskell que fueron aprovechadas al realizar este proyecto y las dificultades que afrontamos durante su elaboración.

2. Diseño de la solución

2.1 Diseño del modo mente maestra

Se definió un nuevo tipo de dato llamado Opciones, en el que se encuentran los tipos Toro, Vaca y Vacío, que formarán parte de la lógica de todo el juego y facilitan las validaciones de las palabras.

```
data Opciones
  = Toro -- ^ Toro es el tipo de dato que representa a los toros en el juego.
  | Vaca    -- ^ Vaca es el tipo de dato que representa a las vacas en el juego.
  | Vacio   -- ^ Vacio es el tipo de dato que representa a los valores que no son vacas ni
  ↪  toros.
  deriving (Eq) -- ^ Permite que se comparen los tipos de Opciones entre ellos con el
  ↪  operador ==
```

La lógica del modo mentemaestra se diseñó con ayuda de dos funciones. La función g se encarga de tomar dos entradas que son la palabra que introduce el usuario y la palabra a la que debe llegar, con la función g se evalúa la palabra introducida por el usuario para verificar si contiene toros.

```
g :: String -> String -> ([Either Opciones Char],Map Char Int)
g target guess = (zipWith toroVaca target guess, Prelude.foldr crearDict (fromListWith (+)
↪  [(c, 1) | c <- target]) (zip target guess))
    where
        toroVaca :: Char -> Char -> Either Opciones Char
        toroVaca x y
            | x == y = Left Toro
            | otherwise = Right y
        crearDict :: (Char,Char) -> Map Char Int -> Map Char Int
        crearDict (t,g) dict =
             if t==g
                 then Map.adjust pred t dict
                 else dict
```

Además se genera un diccionario que contiene la cantidad de letras que se encuentran disponibles para la evaluación posterior de la palabra. Si la palabra introducida por el usuario tiene el mismo carácter que la palabra objetivo en la misma posición, se disminuye en uno el valor de dicha letra en el diccionario, por lo que ya se ha encontrado una de las letras.
La función g retorna una tupla que contiene en su primer valor una lista de Either Opciones Char y en su segundo valor contiene el diccionario que aloja tuplas de letras y la cantidad que resta por evaluar.
Posteriormente, se creó una función h que se encarga de manejar la lista de Either Opciones Char y el diccionario. Esta funcion llama a una función buscar que busca en cada uno de los caracteres dn la lista Either Opciones Char si son palabras que se encuentran en el diccionario, si lo son disminuye el valor y agrega el valor Vaca al final de la lista.

```
h :: [Either Opciones Char] -> Map Char Int -> [Opciones]
h lista diccionario = fst (Prelude.foldl buscar ([],diccionario) lista)
    where
        buscar :: ([Opciones],Map Char Int) -> Either Opciones Char ->  ([Opciones],Map Char
        ↪  Int)
        buscar (ansSoFar,mem) (Left opcion)  = ( ansSoFar ++ [opcion],mem)
        buscar (ansSoFar,mem) (Right palabra) =
            if Map.lookup palabra mem > Just 0
                then (  ansSoFar ++ [Vaca],Map.adjust pred palabra mem)
                else ( ansSoFar ++ [Vacio],mem)
```

En caso de que sean Opciones, la lista ya los ha asignado como Toros y únicamente se agregar a una nueva lista de Opciones. Si no existen en el diccionario, son la opción Vacio y se agregan de esta manera en la lista de Opciones.
Luego de que la funcion h hace la llamada a la función buscar por medio de foldl que la aplica sobre toda la lista de Either Opciones Char. Retorna una tupla con la lista de Opciones y el diccionario, pero como solo se requiere la lista de Opciones, finalmente la función h retorna esta lista de Opciones.

Por lo que, las funciones h y g se utilizan para obtener la combinación de Vacas, Toros y Vacíos a partir de una palabra objetivo y la palabra que le inserta el usuario al juego.

```
conseguir :: String -> String -> [Opciones]
conseguir target = uncurry h . g target
```

2.1.1 Entrada por consola modo mente maestra

Para recibir entradas de consola en este modo, se verifica que lo introducido por el usuario sea correcto y posteriormente se llama a la función conseguir que verifica luego si el usuario introdujo la palabra correcta y ganó o realiza un ciclo para darle otra oportunidad. Solo hay un total de 6 oportunidades y si se acaban el juego termina.

2.2 Diseño del modo descifrador

Para el modo descifrador se generó un nuevo tipo de datos para los árboles denominado Arbol a b c que puede contener un nodo vacío o Empty y se puede llamar así mismo de manera recursiva con Nodo a b c [Arbol a b c].

```
data Arbol a b c
  =  Empty  -- ^ El nodo Empty es el nodo de un padre que no tiene hijos
  |  Nodo  -- ^ El Nodo a b c contiene un nodo o varios en la lista que puede ser Empty o
  ↪  una llamada recursiva.
        a  -- ^ a es la palabra que se guarda como 'String' en caso de que el nodo sea un
        ↪  nodo palabra sino es ""
        b  -- ^ b es la posibilidad que se guarda como 'Opciones' en caso de que sea un
        ↪  nodo posibilidad sino es []
        c  -- ^ c representa el valor de la palabra o la posibilidad en 'Float'
        [ Arbol a b c] -- ^ Esta es la llamada recursiva al tipo de dato, si no tiene hijos
        ↪  es Empty.
  deriving Show -- ^ derivado de 'Show' permite que el arbol pueda verse en la consola
```

Este árbol que deseamos crear es un árbol minimax, por lo que necesitamos ciertas funciones que nos ayuden en su creación. Este árbol minimax cuenta con una raíz en la que se encuentra la entrada de toros y vacas que introduce el usuario.
Luego se deben encontrar las palabras con los menores valores basados en la puntuación de Scrabble para cada letra y se deben conseguir las posibilidades partiendo de cada una de estas palabras y tomar las 10 posibilidades con mayor valor.
Para realizar todo esto, debemos conseguir las palabras, las posibilidades y mapear las letras a los valores deseados.
La función revisar recibe la lista de palabras, una palabra objetivo y una lista de opciones introducido por el usuario y guarda en una lista de strings las palabras que al operarlas por medio de la función conseguir den el mismo resultado que las opciones dadas por el usuario.

```
revisar :: [String] -> String -> [Opciones] -> [String]
revisar [] palabra guess = []
revisar (x:xs) palabra guess =
                        if conseguir palabra x == guess
                           then [x] ++ revisar xs palabra guess
                        else
                           revisar xs palabra guess
```

Se buscan las palabras cuyo resultado para conseguir evaluado con la palabra objetivo de lo mismo que el string introducido por el usuario.
Por otro lado, las posibilidades se generan por medio de una función posibilidades que retorna una lista de lista de Opciones que se consigue por medio de comprensión de listas.
Las funciones stringAOpciones, mapearAOpciones, valorPalabra, conseguirValor, valorPosibilidad y conseguirValorPosible se encargan de mapear los caracteres de un String a un valor Float que pueda ser colocado en el árbol.
La función palabrasAUsar utiliza las palabras obtenidas en revisar y les calcula su valor y retorna una lista de tuplas con el string de la palabra y su valor.

```
palabrasAUsar :: [String] -> [(String,Float)]
palabrasAUsar [] = []
palabrasAUsar (x:xs) = [(x,valorPalabra x)] ++ palabrasAUsar xs

valorPalabra :: String -> Float
valorPalabra palabra = sum (Prelude.map conseguirValor palabra)

conseguirValor :: Char -> Float
conseguirValor letra
    | letra `elem` ['A','E'] = 0.1
    | letra `elem` ['I','N','O','R','S'] = 0.2
    | letra `elem` ['D','L','C','T','U'] = 0.3
    | letra `elem` ['B','G','M','P'] = 0.5
    | letra `elem` ['F','H','Q','V','Y'] = 0.8
    | otherwise = 1
```

De manera similar, la función posibilidadAUsar recibe las posibilidades obtenidas con conseguirPosibilidades y consigue su valor con la función valorPosibilidad. Retorna una lista de tuplas con la lista de Opciones y su valor en Float.

```
posibilidadAUsar :: [[Opciones]] -> [([Opciones],Float)]
posibilidadAUsar [] = []
posibilidadAUsar (x:xs) = [(x,valorPosibilidad x)] ++ posibilidadAUsar xs

valorPosibilidad :: [Opciones] -> Float
valorPosibilidad posibilidad = 1.0 + sum (Prelude.map conseguirValorPosible posibilidad)

conseguirValorPosible :: Opciones -> Float
conseguirValorPosible opcion
    | opcion == Toro = -0.2
    | opcion == Vaca = -0.1
    | otherwise = 0
```

2.2.1 Construcción del Árbol Minimax

Una vez que tenemos todas las palabras, posibilidades y los valores que tienen, podemos construir un árbol minimax.
El árbol se construye con ayuda de la función crearArbol, el nodo padre solo cuenta con la lista de Opciones introducida por el usuario y se considera la palabra elegida al azar para la creación de los nodos hijos.

```
crearArbol :: [Opciones] -> String -> [String] -> Arbol String [Opciones] Float
crearArbol opciones palabra listaPalabras = Nodo "" opciones 0.0 ([] ++ crearHijosPalabras
↪  opciones palabra listaPalabras 1)
```

Luego con la función crearHijosPalabras se llama a una función llamada transform que toma la lista de tuplas de palabrasAUsar pero las ordena y considera solo las primeras 10. Por lo que, en transform se crean estos nodos del árbol y se llama de manera recursiva a crearHijosPosibilidades.

```
crearHijosPalabras :: [Opciones] -> String -> [String] -> Int -> [Arbol String [Opciones]
↪  Float]
crearHijosPalabras opciones palabra listaPalabrasCompleta numero = transform (Prelude.take
↪  10 (sortBy (comparing $ snd) (palabrasAUsar (revisar listaPalabrasCompleta palabra
↪  opciones)))) numero
    where
        transform :: [(String,Float)] -> Int -> [Arbol String [Opciones] Float]
        transform algo 4 = [Empty]
        transform [] num = []
        transform (x:xs) num = [Nodo (fst x) [] (snd x) (crearHijosPosibilidades (fst x)
        ↪  listaPalabrasCompleta (num+1))] ++ transform xs num
```

En crearHijosPosibilidades se llama a una función transformar que toma la lista de tuplas de posibilidades, pero las ordena de mayor a menor y toma las primeras 10. Con esto se crean nuevos nodos y se llama de manera recursiva a crearHijosPalabras.

```
crearHijosPosibilidades :: String -> [String] -> Int -> [Arbol String [Opciones] Float]
crearHijosPosibilidades palabra listaPalabras numero = transformar (Prelude.take 10 $ sortBy
↪  (comparing $ Down . snd) $ posibilidadAUsar (conseguirPosibilidades listaPalabras
↪  palabra)) numero
    where
        transformar :: [([Opciones],Float)] -> Int -> [Arbol String [Opciones] Float]
        transformar algo 4 = [Empty]
        transformar [] num = []
        transformar (x:xs) num = [Nodo "" (fst x) (snd x) (crearHijosPalabras (fst x)
        ↪  palabra listaPalabras (num+1))] ++ transformar xs num
```

El programa puede culminar la creación del árbol, ya que, se mantiene un número que indica en que nivel del árbol se encuentra y en caso de llegar al último nivel, que indica una profundidad 4, se añade un nodo Empty y finaliza la creación del árbol.
Por esto, se garantiza que el árbol será un árbol de 10 nodos hijo o menos por padre y con una profundidad de 4. Una vez tengamos el árbol es necesario recorrerlo para sumar los valores que hemos añadido en cada uno de los nodos.
La función recorrido se encarga de recorrer el árbol de manera recursiva y va sumando los valores y utiliza una función auxiliar recorre para llamar a los hijos que se encuentran entre corchetes.
Como deseamos buscar a las palabras de primer nivel que tengan el menor valor, contamos con una función conseguirPalabras que introduce estas palabras en una lista de Strings y luego de recorrer el árbol y sumar todos los valores, nos queda una lista de Float. La función recorrerArbol une estas palabras con sus valores gracias a la función zip y retorna una lista de tuplas.

```
recorrerArbol :: Arbol String [Opciones] Float -> [(String,Float)]
recorrerArbol (Nodo palabra opciones valor [Empty]) = [(palabra,valor)]
recorrerArbol (Nodo palabra opciones valor arbolHijo) = zip (conseguirPalabras arbolHijo)
↪  (Prelude.map recorrido arbolHijo)

recorrido :: Arbol String [Opciones] Float -> Float
recorrido (Nodo palabra opciones valor [Empty]) = valor
recorrido (Nodo palabra opciones valor arbolHijo) = valor + recorre arbolHijo

recorre :: [Arbol String [Opciones] Float] -> Float
recorre [] = 0
recorre (x:xs) = recorrido x + recorre xs

conseguirPalabras :: [Arbol String [Opciones] Float] -> [String]
conseguirPalabras [] = []
conseguirPalabras ((Nodo palabra opciones valor arbolHijo):xs) = [palabra] ++
↪  conseguirPalabras xs

palabraEncontrada :: Arbol String [Opciones] Float -> [String] -> [(String,Float)]
palabraEncontrada arbol palabrasGeneradas = conseguirW $ sortBy (comparing $ snd)
↪  (recorrerArbol arbol)
    where
        conseguirW :: [(String,Float)] -> [(String,Float)]
        conseguirW [] = []
        conseguirW (x:xs)
                        | (((fst x) `elem` palabrasGeneradas) == True) = conseguirW xs
                        | (((fst x) `elem` palabrasGeneradas) == False)= [x]
```

Finalmente, se puede conseguir con palabraEncontrada la palabra que adivina la computadora, al llamar a la función recorrerArbol, ordenar por la segunda posición de la tupla que contiene el valor de la palabra y tomar únicamente el elemento que es la palabra de menor valor que no se haya utilizado anteriormente, las palabras adivinadas anteriormente se encuentran en palabrasGeneradas.

2.2.2 Manejo de las trampas del jugador

En el modo descifrador el jugador puede intentar hacer trampa tratando de modificar los Toros y Vacas de manera que la computadora no sea capaz de encontrar una palabra y así ganar la partida. Esto se puede evitar gracias a la función checkGuess en el módulo Main.
Esta función se encarga de transformar lo introducido por el usuario en la partida actual y la anterior en un String de V,T y -. Luego evalúa, caracter a caracter la palabra actual y la del turno anterior que ha adivinado la computadora.
Estos caracteres se evalúan con una tupla de Char que recibe la función comparacionC y retorna True si hay trampa y False si no. Se evalúan varios casos posibles al momento de ingresar las opciones de cada palabra. Finalmente, se evalúa la lista de Bool generada al utilizar la función zipWith sobre comparacion y se consigue si hay o no trampa en toda la palabra.

```
buscarToros :: Opciones -> Char
buscarToros op
              | op == Toro = 'T'
              | op == Vaca = 'V'
              | otherwise = '-'
checkGuess :: [Opciones] -> [Opciones] -> String -> String -> Bool
checkGuess actual anterior palabra palabraAnterior =
                        if anterior == []
                            then False
                        else do
                            let act = zip (map buscarToros actual) palabra
                            let ant = zip (map buscarToros anterior) palabraAnterior
                            if (compareChars act ant) == True
                                then True
                            else
                                False
    where
        compareChars :: [(Char,Char)] -> [(Char,Char)] -> Bool
        compareChars actual anterior = and (zipWith comparacionC actual anterior)
        comparacionC :: (Char,Char) -> (Char,Char) -> Bool
        comparacionC tupla1 tupla2
                            | (fst tupla1 == fst tupla2) && (fst tupla1 == '-') && (snd
                            ↪  tupla1 /= snd tupla2) = False
                            | (fst tupla1 == fst tupla2) && (fst tupla1 == 'V' || fst tupla1
                            ↪  == 'T') && (snd tupla1 == snd tupla2) = False
                            | (fst tupla1 == fst tupla2) && (fst tupla1 == 'V' || fst tupla1
                            ↪  == 'T') && (snd tupla1 /= snd tupla2) = True
                            | (fst tupla1 /= fst tupla2) && (snd tupla1 /= snd tupla2) =
                            ↪  False
                            | (fst tupla1 /= fst tupla2) && (snd tupla1 == snd tupla2) =
                            ↪  True
                            | otherwise = False
```

2.2.3 Entrada por consola modo descifrador

Para recibir entradas de consola en este modo, se recibe la entrada del usuario y se verifica que sea correcta. Luego se llama a la función de validación validarE con la opción del usuario y el objetivo que se tiene.
En cada entrada del usuario, si es correcta, se crea un árbol y se llama a la función palabraEncontrada. Si se consigue una palabra, se pide nuevamente una entrada al usuario, con la adivinación de la computadora y se creará un nuevo árbol con la nueva entrada de usuario, realizando un ciclo hasta que se acaben las oportunidades o la computadora descubra la palabra.
La computadora conoce que el usuario ha hecho trampa, si no existen más opciones en palabraEncontrada y finaliza el juego.

2.3 Módulos del programa

El programa se ha dividido en tres módulos:

- Main

- MenteMaestra
- Descifrador

2.3.1 Módulo Main

Este módulo incluye las funciones que manejan las entradas por consola del usuario y es el módulo principal del juego. Desde este módulo se verifican las entradas del usuario y se hace uso de las funciones importadas desde los módulos MenteMaestra y Descifrador, al igual que los tipos que allí se encuentran.

2.3.2 Módulo MenteMaestra

En este módulo se encuentran las funciones g, h y conseguir que pueden encontrar una lista de tipo Opciones con tan solo introducir la palabra escrita por el usuario y la palabra objetivo elegida al azar por la computadora. En MenteMaestra se define el tipo Opciones que incluye a los tipos Toro, Vaca y Vacío que se utilizarán a lo largo de todo el programa.

2.3.3 Módulo Descifrador

En este módulo se encuentran las funciones revisar, stringAOpciones, mapearAOpciones, valorPalabra, conseguirValor, valorPosibilidad, conseguirValorPosible, palabrasAUsar, conseguirPosibilidades, posibilidadAUsar, crearArbol, crearHijosPalabras, crearHijosPosibilidades, recorrerArbol, recorrido, recorre, conseguirPalabras, palabraEncontrada y conseguirW.
El principal objetivo de este módulo es la creación del árbol minimax con ayuda de funciones auxiliares que consiguen las probabilidades y las palabras a usar, al igual que sus valores y el recorrido de dicho árbol para conseguir la palabra en el primer nivel que sea de valor mínimo.
En Descifrador se define el tipo Arbol que se utilizará para crear un árbol minimax y es la principal herramienta que usa el programa para adivinar una palabra del usuario en modo descifrador.

2.4 Características de Haskell utilizadas

Una de las características de Haskell más utilizadas en este proyecto es el excelente manejo de la recursión. Haskell es capaz de manejar recursiones mucho más extensas que otros lenguajes de programación, especialmente que los lenguajes imperativos.
Gracias a que Haskell puede manejar una gran cantidad de recursiones antes de mostrar el error "Stack Overflow", se pudieron utilizar una gran cantidad de recursiones, tanto para crear el árbol como para visitarlo y también para buscar elementos en diversas listas a lo largo de todo el programa.
Por esto, esta implementación es única de Haskell, ya que si intentasemos pasar esta implementación a cualquier lenguaje imperativo, no se podrían manejar todas las recursiones que existen en este programa.
Además como Haskell es un lenguaje funcional cuenta con una gran variedad de funciones únicas como zip, zipWith, foldl, foldr, take, sortBy y map que fueron de gran utilidad al momento de implementar este juego.
Con estas funciones se facilita el uso de listas y también se pueden aplicar a los Strings, ya que se trata de listas de caracteres, por lo que se utilizaron estas funciones para "iterar" de manera sencilla sobre listas, strings, modificar valores basándose en una función externa y validar listas, Opciones y mucho más.
También fue de gran utilidad la permisibilidad que tiene Haskell para la creación de nuevos tipos de datos de manera sencilla. Con data, se pudieron crear los tipos de datos que representan los Toros, Vacas y Vacíos, facilitando así la verificación de la existencia de palabras y se utilizó también para generar el tipo de árbol, estructura de la que depende el modo descifrador.
Además se utilizó múltiples veces la palabra reservada where, que puede ser utilizada para dividia la lógica de una función en partes más pequeñas y que se facilite su lectura y el manejo de las operaciones.
La palabra reservada where fue de gran utilidad para generar pequeñas funciones dentro de otras funciones que de otra manera necesitarían de una función definida en otro lugar del código.

3. Dificultades en la implementación

La única dificultad en la implementación fue la creación de el árbol minimax y su recorrido de manera rápida. Si bien el programa puede crear y recorrer el árbol de manera correcta, no es posible hacerlo de manera instantánea y se deben esperar unos segundos para obtener el resultado de la palabra adivinada por la computadora con el modo descifrador.

3.1 Aspectos rápidos de la implementación

El aspecto más rápido de la implementación es el modo mente maestra. En este modo se pueden obtener los resultados deseados en pocos segundos. En las funciones utilizadas para la lógica de este modo, no se incluyen una gran cantidad de recursiones y se puede ejecutar rápidamente.
La función g tiene un tiempo de ejecución de O(n), con n siendo el tamaño de las palabras, por lo que es O(5). Ya que, se genera una lista de Either Opciones Char en la que se llama a la función zipWith con la función toroVaca que tiene un tiempo de O(1).
Por otro lado, la segunda posición de la tupla se genera gracias a una función foldr que utiliza la función crearDict que tiene un tiempo de O(1) y también se llama a la función zip sobre las palabras con un tiempo de O(5).
La función h cuenta con una llamada a la funcion foldl que utiliza la función buscar que es O(1). Por lo que su llamada es O(n) o O(5). Finalmente, la búsqueda de una lista de Toros, Vacas o Vacío, tarda un tiempo de O(5). Por lo que es bastante rápido.

3.2 Aspectos lentos de la implementación

El aspecto más lento de la implementación es la creación y recorrido del árbol minimax. Aunque las asignaciones que se realizan en las funciones recursivas para la creación y recorrido del árbol tienen un tiempo de O(1). Es importante notar que debido a su recursividad el tiempo de ejecución aumenta mucho.
El árbol cuenta con 10 hijos para cada nodo y tiene 4 niveles de profundidad, por esto hay hasta 11.111 nodos dentro del árbol por lo que su creación y recorrido llevará un tiempo mucho mayor al que se puede realizar en el modo mente maestra. Aunque se obtiene un resultado correcto, no es tan rápido como se desearía.

4. Conclusiones y recomendaciones

Haskell es un lenguaje funcional con excelente manejo de las recursiones gracias a su "lazy evaluation". Además cuenta con ua gran cantidad de funciones que pueden facilitar el manejo de strings y listas y es capaz de lograr diversos cálculos en tan solo una línea de código, lo que lo hace un lenguaje muy sencillo de leer y con una gran capacidad para diversas tareas.
La implementación del juego Wordle en Haskell fue un proceso sencillo, gracias a que se pudo utilizar una de las mejores herramientas en Haskell que es la recursión. Siendo el único problema la creación de un árbol muy grande y su recorrido, los que le restan velocidad al programa.
Se recomienda a quienes deseen realizar este proyecto en un futuro, elaborar una estructura o tipo de dato distinto para representar el árbol, que no dependa tanto del uso de las recursiones o mejorar el recorrido del árbol para mejorar el tiempo de ejecución.