Profesor: Víctor Zamora Gutiérrez
Ayudantes: Edgar Quiroz Castañeda
Sara Doris Montes Incín

Instrucciones

Completa los cuerpos de las funciones incompletas en `src/LC`. Al terminar, hay que verificar que la práctica compile y que pase todas las pruebas unitarias. Finalmente, se debe ejecutar correctamente el programa escrito en `app/Main.hs`

Hay 23 puntos posibles. Se va a calificar sobre 20, por lo que pueden no realizar algunos ejercicios.

1 Listas

Estos ejercicios tiene como objetivo familializar con la creación y manipuación de listas.

0.5 puntos `squarePrimes :: [Int] -> Int`: Obten aquellos que son primos y sacan su cuadrados. Pueden usar listas por comprensión o las funciones `map` y `filter`.

0.5 puntos `oddConcat :: [[Int]] -> [Int]`: Dada una lista de listas, eliminar las listas que contengan un número par, y concatenar las listas resultantes en una nueva lista. Pueden usar `filter` para filtrar las listas y `fold` para concatenarlas.

0.5 puntos `isPalindrome :: [a] -> Bool`: Dada una lista, verificar si es palindromo, es decir que se lea igual de derecha a izquierda o de izquierda a derecha. Podría ser útil definir una función auxilar `reversa` que obtenga la reversa de una lista.

1 punto `countEquals :: Eq a => [a] -> [(Int, a)]`: Contar los elementos consecutivos iguales. Se puede definir la función recursivamente o usar `fold` y definir una función auxiliar para procesar elemento por elemento.

1 punto `rotate :: [a] -> Int -> [a]`: Rotar una lista n lugares a la derecha. Puede ser útil definir una función auxilar para partir una lista en dos a partir de n-ésimo elemento.

2.5 puntos `permutations :: [a] -> [[a]]`: Genera todas las permutaciones de una lista. El problema se puede definir recursivamente, aunque el paso inductivo no es tan sencillo.

2 puntos `pow :: [a] -> [[a]]`: Calcula el conjunto potencia de una lista. Al igual que la función auxiliar, es útil fórmular el problema recursivamente.

2 Árboles

Ejercicios para familiazarse con funcione recursivas sobre tipos similares a las fórmulas.

0.5 puntos `size :: Tree a -> Int`: El número de elementos de un árbol. Es fácil calcularlo de forma recursiva.

0.5 puntos `depth :: Tree a -> Int`: Profunidad de un árbol. Es la distancia más grande entre un nodo y la raíz. También es fácil definirlo recursivamente.

1 punto `width :: Tree a -> Int`: Ancho de un árbol. Es el número de hojas.

1 punto `flatten :: Tree a -> TreeOrder -> [a]`: Aplana el árbol a una lista siguiendo el tipo de recorrido dado.

- `InOrder`: visitar primero el hijo izquierdo, luego la raíz y finalmente el hijo derecho.
- `PreOrder`: visitar primero a raíz, luego el hijo izquierdo y finalmente el derecho

¿Realmente necesitas imprimir esta hoja?

– `PosOrder`: primero el hijo derecho, luego el izquierdo y finalmente la raíz.

Sería conveniente crear funciones auxiliares para cada tipo de recorrido.

2 puntos `level :: Eq a => a -> Tree a -> Maybe Int`: El nivel de un nodo es la distancia desde la raíz hasta el nodo. Si el elemento está varias veces en el árbol, tomar el que esté más cerca. Si no está en el árbol, regresar `Nothing`.

2 puntos `levels :: Tree a -> [[a]]`: Guarda los nodos de cada nivel del árbol en una nueval lista.

0.5 puntos `contains :: Eq a => a -> Tree a -> Bool`: Si el árbol contiene el elemento.

1 punto `addOrd :: Ord a => a -> Tree a -> Tree a`: Añade un elemento a un árbol ordenado.

3 Cifrados

Uno de los esquemas de cifrados clásicos más antiguos es el cifrado de César. Este usa un entero k como llave para encriptar un texto desplazando cada letra k lugares a la derecha de su posición en el alfabeto. Por ejemplo, con $k = 7$

$$\text{POR MI RAZA HABLARA EL ESPIRITU} \xrightarrow{7} \text{WVY TP YHGH OHISHYH LS LZWPYPAB}$$

Para desencriptar un texto, basta con encriptar el cifrado usando $-k$. Para simplificar las operaciones, el alfabeto a usar será

$$\text{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$

Hay tantas llaves como letras en el alfabeto. Con un alfabeto pequeño, es fácil probar todas las posibilidades para encontrar la llave correcta.

1 punto `normalize :: String -> String`: Eliminar todas las letras que no sea parte del alfabeto. Acentos, espacios, puntuaciones deben retirarse. En caso de las letras del alfabeto en minúscula, se suitituirá por su forma mayúscula. Para esto pueden usar la función `toUpper`.

1 punto `(<<) :: Char -> Int -> Char`: Es la función de desplazamiento explicada anteriormente. Puede usar las funciones `ord` y `chr` para trasformar caracteres a números y viceversa. También puede ser útil la función `mod` para operaciones de aritmética modular.

1 punto `encrypt :: CaesarKey -> String -> String`: Aplica el cifrado de César. Basta con aplicar el desplazamiento a cada letra.

1 punto `decrypt :: CaesarKey -> String -> String`: Desencripta un mensaje. Basta con encriptar usando la llave inversa.

2 puntos `candidates :: String -> String -> [(CaesarKey, String)]`: Dado un texto cifrado y una palabra contenida en el texto original, aplica un ataque de fuerza bruta para encontrar las posibles llaves y textos desencriptados. Puedes definir una función auxiliar para saber si una cadena contiene como subcadena a otra.

0.5 puntos ¿De dónde viene el texto cifrado que muestra `app/Main.hs`?

¿Realmente necesitas imprimir esta hoja?