PRUEBA DE PROGRAMACION EN CODILITY
TRADUCCIÓN DE LOS PROBLEMAS PLANTEADOS

TAREA 1
JSListDepth
Dado un documento DOM, calcula la profundidad máxima de listas UL/OL.

Descripción de la tarea
Dado un árbol DOM, tienes que analizar las etiquetas de lista <ul> y <ol> dentro del mismo. Tu tarea es encontrar la profundidad máxima de etiquetas de lista / anidadas. Una única lista / está anidada a 1 nivel de profundidad. Cada lista / dentro de otra lista / está anidada un nivel más profundo. Si no hay ninguna lista <ul> o <ol> en el árbol DOM, la profundidad es igual a 0.

Nota que las listas / pueden estar anidadas directa o indirectamente; por ejemplo, una lista <ul> dentro de una tabla dentro de una lista <ol> está anidada a 2 niveles de profundidad.

Por ejemplo, dado un documento HTML con el siguiente contenido dentro de la etiqueta <body>:

```
<ul>
  <li>Item:
     <ol>
       <li>Point:
         <div>
           <ul>
             <li>elem1</li>
           </ul>
         </div>
       </li>
     </ol>
  </li>
  <li>elem2</li>
</ul>
<ul>
  <li>simple list1</li>
</ul>
<ul>
</ul>
```

hay una lista <ul> anidada a 3 niveles de profundidad. Esto es, “elem1” está en una lista <ul>, la cual está dentro de una lista <ol> que contiene “Point”, mientras esta lista <ol> está dentro de otra lista <ul> que contiene “Item”.

Escribe una función:
function solution();
que, dado el árbol DOM, devuelva la profundidad máxima de listas / anidadas. Por ejemplo, dado el árbol DOM del documento presentado anteriormente, la función debería retornar 3, según la

explicación dada.

Dado el contenido siguiente:

```
<ol>
  <li>
    <ol>
      <li></li>
    </ol>
  </li>
</ol>
```

la función debería retornar 2.

Asume que:

- el árbol DOM representa un documento HTML5 válido
- la longitud del documento HTML no excede 4KB
- jQuery 2.1 está soportado

TAREA 2
TwoDigitHours
Dados dos strings que representan puntos en el tiempo, encuentra el número de puntos entre ellos, de tal manera que su representación en el formato HH:MM:SS use a lo mucho dos dígitos diferentes.

Descripción de la tarea
En esta tarea investigaremos patrones interesantes que podrían ser observados en un reloj digital. Tal reloj representa un *punto en el tiempo* usando el formato HH:MM:SS en donde:

- HH es la hora del día (00 a 23), como dos dígitos decimales
- MM es el minuto dentro de la hora (00 a 59), como dos dígitos decimales
- SS es el segundo dentro del minuto (00 a 59), como dos dígitos decimales

Decimos que un punto en el tiempo es *interesante* si *a lo mucho* usa *dos* dígitos diferentes en todo el string. El objetivo es contar el número de puntos interesantes que pueden ser observados en el reloj en un período de tiempo dado.

Escribe una función:

function solution (S, T);

que, dados dos strings S y T que especifican puntos en el tiempo en el formato “HH:MM:SS”, devuelva el número de puntos interesantes en el tiempo entre S y T (inclusivo). Puedes asumir que S indica un punto en el tiempo anterior a T en el mismo día.

Por ejemplo, dados “15:15:00” y “15:15:12”, tu función debería devolver 1 porque existe únicamente un punto interesante en el tiempo entre estos puntos (“15:15:11”). Dados “22:22:21” y “22:22:23”, tu función debería devolver 3 con puntos interesantes en el tiempo “22:22:21”, “22:22:22” y “22:22:23”.

Asume que:

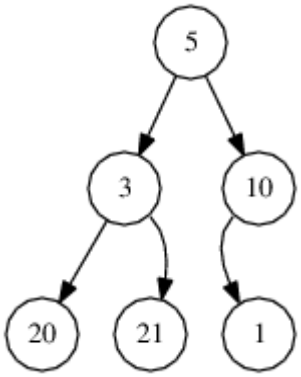
- los strings S y T siguen el formato “HH:MM:SS” de forma estricta
- el string S describe un punto en el tiempo anterior a T en el mismo día

En tu solución, enfócate en **exactitud**. El desempeño de tu solución no será el enfoque de la evaluación.

TAREA 3
TreeVisibility
Calcula el número de nodos visibles en un árbol binario.

Descripción de la tarea
En este problema consideramos árboles binarios, representados por estructuras de datos de punteros.

Un árbol binario es o un árbol vacío o un nodo (llamado *raíz*) que consiste de un valor entero y dos árboles binarios adicionales, llamados el *subárbol izquierdo* y el *subárbol derecho*. Por ejemplo, la figura a continuación muestra un árbol binario de seis nodos. Su raíz contiene el valor 5, y las raíces de sus subárboles izquiero y derecho tienen los valores 3 y 10, respectivamente. El subárbol derecho del nodo que contiene el valor 10, así como los subárboles izquiero y derecho de los nodos que contienen los valores 1, 20, y 21 son árboles vacíos.



Un árbol binario puede ser dado utilizando una estructura de datos de punteros. Asume que las siguientes declaraciones son dadas:

```
// Tree obj is an Object with attributes
// obj.x - type: int
// obj.l - type: Tree
// obj.r - type: Tree
```

Un árbol binario está representado por el puntero vacío (denotado por *null*). Un árbol no vacío está representado por un puntero a un objeto representando su raíz. El atributo x contiene el integer contenido en la raíz, en donde los atributos l y r contienen los subárboles izquierdo y derecho del árbol binario, respectivamente.

Un *camino* en un árbol binario es una secuencia no vacía de nodos que uno puede atravesar siguiendo los punteros. La *longitud* de un camino es el número de punteros atravesados. Más formalmente, un camino de longitud K es una secuencia de nodos P[0], P[1],..., P[K], de tal manera que el nodo P[I+1] es la raíz del subárbol de la izquierda o derecha de P[I], for 0<=I<=K. Por ejemplo, la secuencia de nodos con valores 5, 3, 21 es un camino de longitud 2 en el árbol de la figura anterior. La secuencia de nodos con valores 10, 1 es un camino de longitud 1. La secuencia de nodos con valores 20, 3, 21 no es un camino válido.

La *altura* de un árbol binario está definida como la longitud del camino más largo posible en el arbol. En particular, un árbol que consiste de un solo nodo tiene altura 0 y, convencionalmente, un árbol vacío tiene altura -1. Por ejemplo, el árbol de la figura anterior tiene altura 2.

Un árbol binario T es dado. Un nodo del árbol T que contiene el valor V se describe como *visible* si el camino desde la raíz del árbol al nodo no contiene un nodo con ningún valor excediendo V. En particular, la raíz es siempre visible y nodos con valores menores que el de la raíz nunca son visibles.

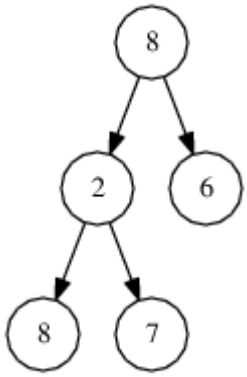
Por ejemplo, el árbol de la figura anterior tiene cuatro nodos visibles: estos son, aquellos con valores 5, 10, 20 y 21. El nodo con valor 1 no es visible porque hay un nodo con valor 10 en el camino desde la raíz al nodo. El nodo con valor 3 no es visible porque su valor es menor que el de la raíz (valor 5).

Escribe una función:

function solution(T)

que, dado un árbol binario T que consiste de N nodos, devuelva el número de nodos visibles. Por ejemplo, dado el árbol de la figura anterior, la función debería devolver 4 tal como se explicó.

Dado el árbol T con la siguiente estructura:



la función debería devolver 2, porque los únicos nodos visibles son aquellos con valor 8.

Para el propósito de que ingreses tus propios casos de prueba, puedes denotar un árbol recursivamente en la siguiente forma. Un árbol binario vacio está denotado por None. Un árbol no vacío está denotado como (X,L,R), en donde X es el valor contenido en la raíz y L y R denotan los subárboles de la izquierda y derecha, respectivamente. Los árboles de las dos figuras anteriores pueden ser denotados como:

```
  (5, (3, (20, None, None), (21, None, None)), (10,
(1, None, None), None))
```

and:

```
  (8, (2, (8, None, None), (7, None, None)), (6,
None, None))
```

Asume que:

- N es un entero dentro del rango [0..50.000]
- cada valor en el árbol es un entero dentro del rango [-100.000..100.000]
- la altura del árbol T (número de aristas en el camino más largo desde la raíz a la hoja) está dentro del rango [-1..500]

Complejidad:

- el peor caso esperado en complejidad de tiempo es O(N)
- el peor caso esperado en complejidad de espacio es O(N)