```python
from algo1 import *
from linkedlist import *
from binarytree import traverseInOrder
import array

# 1
#Se recorre desde la esquina superior derecha.
# Si el valor es mayor al buscado,se avanza hacia la izquierda; si es
menor, hacia abajo.
# Complejidad O(M + N),
def search_sorted_matrix(matrix, target):
    row = 0
    rows = length(matrix) - 1
    col = length(matrix[0]) - 1
    while row < rows and col >= 0:
        current = matrix[row][col]
        if current == target:
            return (row, col)
        elif current > target:
            col -= 1
        else:
            row += 1
    return None

# 2
# Trabajamos por posiciones haciendo llamadas recursivas.
# En una de las llamadas incluimos la posicion actual y en la otra no.

# Agrega un array al resultado de subconjuntos
def append_array_result(result, count, arr):
    result[count] = arr
    return count + 1

#Cuenta la cantidad de espacios no vacios de un array
def count_not_none(array):
    count = 0
    n = array.length(array)
    for i in range(n):
        if array[i] is not None:
            count += 1

# Función recursiva para generar subconjuntos
def subset_array_rec(L, index, current, result, count):
```

```
    if index == array.length(L):
        return append_array_result(result, count, current)

    # No incluir el valor actual
    count = subset_array_rec(L, index + 1, current, result, count)

    # Incluir el valor actual
    current[count_not_none(current)] = L[index]
    count = subset_array_rec(L, index + 1, current, result, count)
    array.delete(current, L[index])  # Volver al estado anterior

    return count


#Función principal
def subset_array(L):
    #Longitud del array a trabajar
    n = array.length(L)

    #Longitud del array resultado
    result_size = 2**n

    result = Array(result_size, Array)  #Array resultado
    current = Array(n, 0)
    #Llamado a funcion recursiva
    subset_array_rec(L, 0, current, result, 0)
    return result

#Funcion que imprime un array de arrays
def print_subsets_array(result):
    for i in range(array.length(result)):
        if result[i] is not None:
            print([result[i][j] for j in range(len(result[i])) if
result[i][j] is not None])

# 3
#Sabemos que el recorrido in Order de un arbol nos devuelve los valores
en orden ascendente (si el arbol es un BST).
#Por lo tanto, si la lista resultante es ascendente, el arbol es un
BST.
def checkBST(B):
    values = traverseInOrder(B)
    current = values.head
```

```
    while current and current.nextNode:
        if current.value >= current.nextNode.value:
            return False
        current = current.nextNode
    return True

# 4
#Separamos el algoritmo en dos partes:
# 1) Verificar si los árboles son iguales (llamada recursiva que
compara keys, y repite el proceso para sus hijos).
# 2) Comparamos desde la raiz del arbol 2 con algun nodo del arbol 1
(el nodo del arbol 1 cambia con cada llamada recursiva).
def is_subtree(BT1, BT2):
    #Verificamos si son el mismo arbol, devolviendo verdadero o falso
    def same_Tree(n1, n2):
        if n2 is None:
            return True
        if n1 is None:
            return False
        return (n1.key == n2.key and
    same_Tree(n1.leftNode, n2.leftNode) and
    same_Tree(n1.rightNode, n2.rightNode))

    def search_Subtree(n1):
        if n1 is None:
            return False
        if same_Tree(n1, BT2.root):
            return True
        return search_Subtree(n1.leftNode) or
search_Subtree(n1.rightNode)

    if BT2.root is None:
        return True         # Un árbol vacío es subárbol de cualquier
árbol
    return search_Subtree(BT1.root)


# 5
# Utilizamos una lista enlazada como cache, donde el head es el más
reciente.
# Si se accede a un elemento, se mueve al head. Si se inserta y excede
el tamaño, se elimina el último nodo.
#Busca un elemento dentro de la lista enlazada
```

```
#Funcion que nos devuelve la posicion del elemento buscado por key, si
es que este se encuentra en la cache
def search_by_key(LinkedList, key):
    current = LinkedList.head
    currentpos = 0
    while (current != None):
        if (key == current.key):
            return currentpos
        else:
            currentpos += 1
            current = current.nextNode
    return None

#Borra un elemento de la cache en una posicion determinada
def delete_position(cache, position):
    currentpos = 0
    current = cache.head
    while current.nextNode and currentpos < position-1:
        current = current.nextNode

    current.nextNode = current.nextNode.nextNode
    return 0

#Nos permite acceder a un elemento de la cache.
#Si pudimos acceder, manda el elemento a la primera posicion de la
cache
def lru_get(cache,key):
    pos = search_by_key(cache, key)
    if pos is not None:
        delete_position(cache, pos)
        add(cache,key)
        return key
    return None

#Si el elemento a insertar se encuentra en la cache, llamamos a get.
#Si no se encuentra, lo agregamos en la primera posicion, y eliminamos
el LRU element.
def lru_put(cache,key):
    pos = search_by_key(cache, key)
    if pos is not None:
        lru_get(cache,key)
    else:
        add(cache, key)
```

```
    size = length(cache)
    delete_position(cache,size-1)
```