Evaluación asincrónica 2 de Estrategias Algorítmicas, ITESO, Luis Gatica

Fecha de entrega de la evaluación completa: lunes 11 de abril de 2022

Nombre: Rodríguez Castro, Carlos Eduardo; Cordero Hernández, Marco Ricardo

Parte 1: análisis *divide y vencerás*

Ejercicios más sencillos

Potencia divide y vencerás

Use las siguientes propiedades aritméticas para programar un **método divide y vencerás** (llamado *pow*) que reciba *x*,*y* y calcule *x^y* (asuma que *x*,*y* no son muy grandes (no habrá *overflow* si usa *int* en Java)) y que *x>=1* y *y>=0*:

1. Si *y=0*, entonces *x^y = x^0 = 1*
2. Si *y* es par, entonces *x^y* se puede calcular como *x^(y/2)* multiplicado por sí mismo.
3. Si *y* es non, *x^y* se puede calcular como *x^(y/2)* multiplicado por sí mismo y luego por *x*.*

*Nota: en los puntos 2 y 3 el operador de división es el operador de división entera (sólo entrega la parte entera de la división; ej: *7/2 = 3*).

```
int pow(int x, int y) {
      return (y <= 0) ? 1 : (y % 2 == 0) ?
      pow(x * x, y / 2) : x * pow(x * x, y / 2);
}
```

Haga un análisis de la complejidad temporal del método. Una justificación completa puede ser escrita, en este caso, en dos líneas (no es necesario que elabore árboles o reglas de recurrencia, aunque puede utilizar cualquiera de esas herramientas).

R: Para la solución proporcionada, la complejidad temporal es *log(n)*, ya que las potencias se van calculando de forma que el exponente se divide entre dos y se compensa con la *x* individual de la última línea como potencia impar.

Búsqueda binaria

Escriba en Java los métodos necesarios (2) para implementar *binarySearch()* para enteros.

```
boolean search(int[] arr, int n) { // Iterativo
       int t = arr.length - 1, b = 0, m;

       while (b <= t) {
       m = b + ((t - b) / 2);

            if (arr[m] > n) t = m - 1;
            else if (arr[m] < n) b = m + 1;
            else return true;
      }

        return false;
}
```

```java
public static boolean searchR(int[] arr, int n, int b, int t) { // Recursivo
	if (t < b) return false;

	int m = b + ((t - b) / 2);

	if (n > arr[m]) return searchR(arr, n, m + 1, t);
	else if (n < arr[m]) return searchR(arr, n, b, m - 1);
	else return true;
}
```

Mergesort

Escriba en Java los métodos necesarios para implementar *mergesort*. Recuerde que sólo tiene permitido reservar memoria adicional para otro arreglo del mismo tamaño que el original, más las variables (acotadas en un orden constante) que necesite: contadores, índices, temporales...

```java
void sort(int[] array) {
	int[] helper = new int[array.length];
	mergesort(array, helper, 0, array.length - 1);
}

void mergesort(int[] array, int[] helper, int low, int high) {
	if (low < high) {
		int middle = (low + high) / 2;

		mergesort(array, helper, low, middle); // Ordenar mitad izquierda
		mergesort(array, helper, middle + 1, high); // Ordenar mitad derecha
		merge(array, helper, low, middle, high); // MEZCLARLOS
	}
}

void merge(int[] array, int[] helper, int low, int middle, int high) {
	for (int i = low; i <= high; i++) {
		helper[i] = array[i];
	}

	int helperLeft = low;
	int helperRight = middle + 1;
	int current = low;

	while (helperLeft <= middle && helperRight <= high) {
		if (helper[helperLeft] <= helper[helperRight]) {
			array[current] = helper[helperLeft];
			helperLeft++;
		} else {
			array[current] = helper[helperRight];
```

```
                        helperRight++;
                } current++;
        }

        int remaining = middle - helperLeft;
        for (int i = 0; i <= remaining; i++) {
                array[current + i] = helper[helperLeft + i];
        }
}
```

Quicksort

Escriba en Java los métodos necesarios para implementar *quicksort*, incluyendo un método recursivo principal y un método iterativo (no recursivo) *partition()* para calcular la posición final del pivote actual, colocando a ese pivote en ella, a los elementos menores o iguales a su izquierda y a los mayores a su derecha.

```
void quickSort(int[] arr, int left, int right) {
    int index = partition(arr, left, right);
    if (left < index - 1) quickSort(arr, left, index - 1);
    if (index < right) quickSort(arr, index, right);
}

void swap(int[] arr, int left, int right) {
    int holder = arr[left];
    arr[left] = arr[right];
    arr[right] = holder;
}

int partition(int[] arr, int left, int right) {
    int pivot = arr[right - 1];

    while (left < right) {
        while (arr[left] < pivot) left++;
        while (arr[right] > pivot) right--;

        if (left <= right) {
            swap(arr, left, right);
            left++;
            right--;
        }
    }

    return left;
}
```

Estadístico de orden *i*

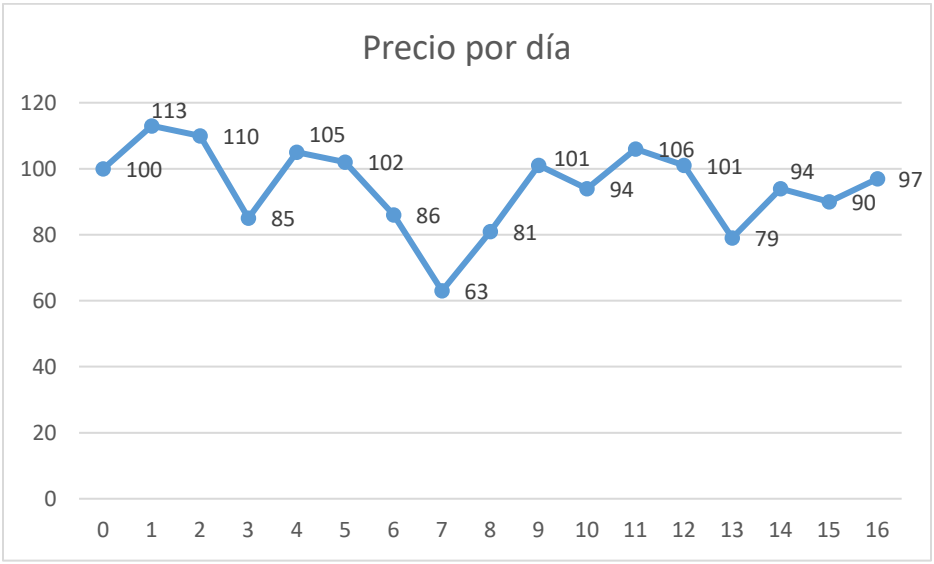
Aprovechando el método *partition()* que ya escribió, agregue los métodos necesarios para calcular el estadístico de orden *i*: es decir, para devolver el dato que estaría en el índice i si el arreglo estuviera ordenado. Siga un enfoque divide y vencerás para resolver este problema, y no intente ordenar primero todo el arreglo para simplemente devolver lo que está en la posición *i*: eso sería demasiado trabajo computacional para resolver un problema mucho más sencillo (es decir: menos complejo, en términos del orden de crecimiento de la solución buscada).

```
int findStatisticN(int[] arr, int n, int left, int right) {
    int pivot = partition(arr, left, right);

    return (pivot == n - 1) ? arr[pivot] :
           (n - 1 < pivot) ? findStatisticN(arr, n, left, pivot - 1) :
           findStatisticN(arr, n, pivot + 1, right);
}
```

El problema del subarreglo de suma máxima: introducción

El profesor de Selena está convencido de la utilidad de su clase para las dinámicas del capitalismo tardío. Por ello, deja a Selena y a sus amigos una tarea sobre optimización de ganancias.
Selena es contratada por una compañía que comercia piñas y genera toda su utilidad revendiéndolas. La información de la cual ella dispone es el precio de la piña cada día durante los próximos *N* días, y como el mercado está en una competencia perfecta, ella no puede vender la piña a un precio mayor porque no se la comprarían (ni a un precio menor, porque ganaría menos). El problema de Selena consiste en decidir cuándo comprar la piña y cuándo revenderla para maximizar su ganancia, asumiendo que sólo puede comprar en un día y sólo puede revender en otro.

Aquí podemos observar el precio de la piña (en alguna moneda über-capitalista que desconocemos) en los próximos *N*=17 días:



Una forma de determinar qué día comprar y qué día revender depende de calcular las *diferencias de precio* de un día a otro, que se pueden observar (para los datos de la gráfica anterior) en la siguiente tabla:

Día	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Precio	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Incremento		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Visto así, el problema se reduce a **encontrar los índices de un sub-arreglo contiguo de suma máxima** (tomando como arreglo original la lista de incrementos). En este caso, el sub-arreglo de suma máxima es *{18, 20, -7, 12}*, y por lo tanto Selena debe comprar el día 7 y revender el día 11. Si hay varios sub-arreglos de suma máxima, Selena dará prioridad al de índices más pequeños, porque entre más pronto ella compre y revenda, más contentos estarán sus jefes über-capitalistas.
En esta tarea, el profesor quiere que queden claros (o reforzados) los siguientes temas:

- La técnica de diseño de algoritmos divide y vencerás
- El problema del sub-arreglo de suma máxima
- El análisis a posteriori de caso *promedio*

Puede ayudar a Selena a resolver esta tarea utilizando Java, C o C++.

Descripción

1. Solución del problema

La solución al problema del sub-arreglo de suma máxima debe implementarse siguiendo la estrategia *divide y vencerás*, y la complejidad de la solución, de esta manera, será cuasilineal.

La entrada estará compuesta por un renglón que incluya un número *N* de días (identificados del *0* al *N-1*). Luego, en el siguiente renglón, *N* enteros representando cada uno el precio de la piña cada día.
La salida estará compuesta por dos líneas con el siguiente formato:

```
"Selena debe comprar las piñas el día %d", inicio
"Selena debe revender las piñas el día %d", fin
```

Ejemplo

Entrada

```
17
100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97
```

Salida

```
Selena debe comprar las piñas el día 7.
Selena debe revender las piñas el día 11.
```

```java
import java.util.ArrayList;
import java.util.List;

public class maxSum {
    private static class Pair<K, V> {
        public K a; public V b;

        public Pair(K a, V b) {
            this.a = a;
            this.b = b;
        }

        public K getA() {return a;}
        public V getB() {return b;}
    }

    private static List<Integer> parseInput(String values) {
        List<Integer> parsed = new ArrayList<>();
        String[] newValues = values.split(" ");

        for(int i = 0; i < newValues.length; i++) {
            parsed.add(Integer.parseInt(newValues[i]));
        }

        return parsed;
    }

    private static List<Integer> getDifferences(List<Integer> arr) {
        List<Integer> differences = new ArrayList<>();

        for (int i = 1; i < arr.size(); i++) {
            differences.add(arr.get(i) - arr.get(i - 1));
        }

        return differences;
    }

    private static Pair<Integer, Integer> getIndex(List<Integer> arr) {
        int currMax = 0, prevMax = 0, start = 0, i1 = 0, i2 = 0;

        for (int i = 0; i < arr.size(); i++) {
            currMax += arr.get(i);

            if (currMax < 0) {
                start = i + 1;
                currMax = 0;
            } else if (currMax > prevMax) {
                i2 = i;
                i1 = start;
                prevMax = currMax;
            }
        }

        return new Pair(i1, i2);
    }
```

```java
    private static void printList(List<Integer> arr) {
        for (int i = 0; i < arr.size(); i++) {
            System.out.printf("%d ", arr.get(i));
        }

        System.out.println();
    }

    public static void main(String[] args) {
        // Entrada de tamaño de datos (no se necesita para Java)
        int n = 17;
        // Datos de entrada separados por espacios
        String data = "100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97";

        Pair<Integer, Integer> res = getIndex(getDifferences(parseInput(data)));

        System.out.printf("Selena debe comprar las piñas el día %d.\n", res.a);
        System.out.printf("Selena debe revender las piñas el día %d.\n", res.b + 1);
    }
}
```

2. Análisis a posteriori

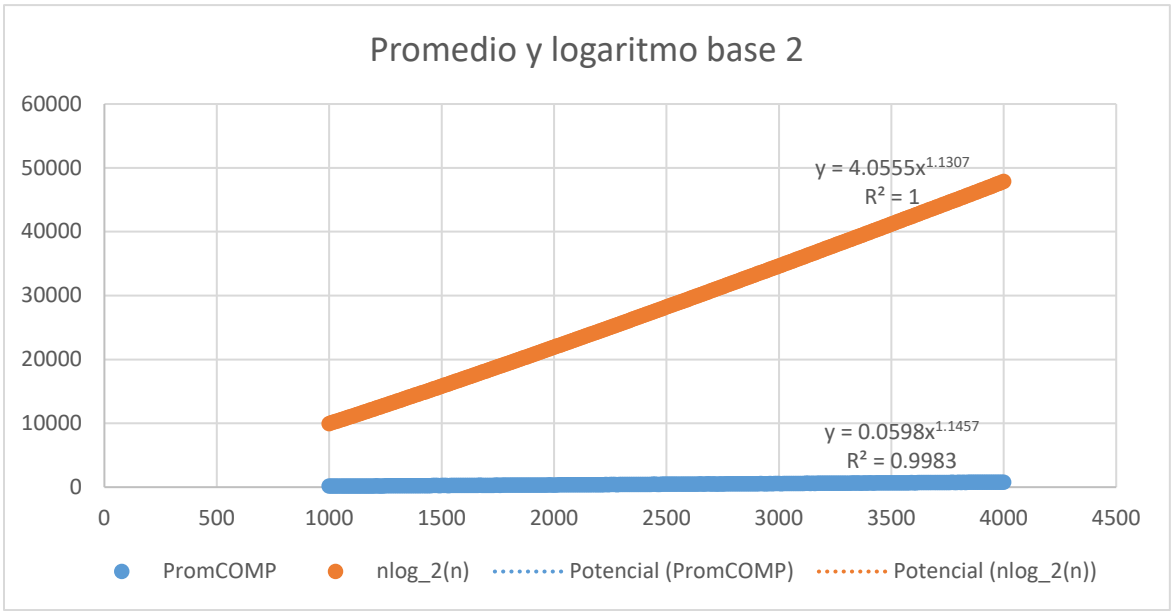
Modifique el código de su solución para contar *comparaciones* entre elementos de sus arreglos (o elementos derivados de ellos, como acumuladores).
Ahora corra el algoritmo para arreglos de tamaño *n=1000* hasta *n=4000* con incremento de *10*:

- Genere en cada caso *n/10* arreglos de enteros con datos entre ***-n/2*** y ***n/2***.
- Calcule el promedio de *comparaciones* para esos *n/10* arreglos.
- Despliegue la información con el siguiente formato:

```
"%d\t%.2f\n", n, promedioComparaciones
```

Copie esos datos, péguelos tal cual en Excel y genere una *gráfica de dispersión* donde aparezcan dos curvas:

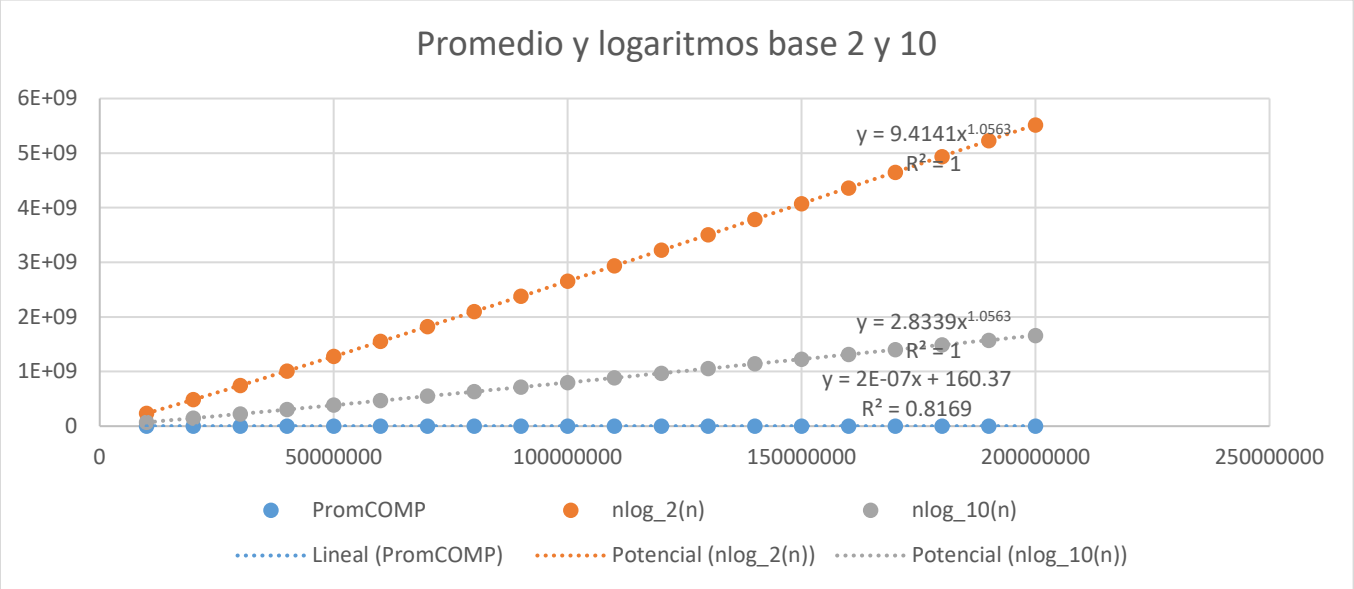
1. Promedio de *comparaciones* en función de *n*.
2. $nlog_2(n)$.



Agregue a la primera curva la ecuación de tendencia (potencial, no polinómica) y el valor de ajuste estadístico R^2.
Utilizando las siguientes ecuaciones, tabule para *N* desde 10 millones hasta 200 millones, utilizando un incremento de 10 millones:

1. La ecuación de tendencia potencial obtenida en el paso anterior.
2. $nlog_2(n)$.
3. $nlog_{10}(n)$.

No incluya aquí su tabla, pero sí una *gráfica* donde aparezcan las tres curvas correspondientes.
Escriba a qué *conclusiones* llega, a partir de estos datos y gráficas, al respecto de la complejidad algorítmica de su solución.



Dado que la curva correspondiente a la solución proporcionada siempre está por debajo de las líneas de los logaritmos calculados, se puede decir que es una solución lineal.

Parte 2: árboles

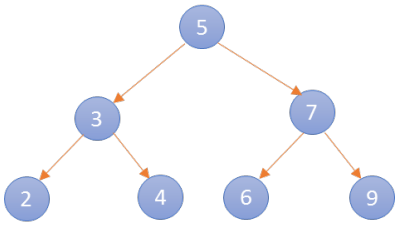
Para los siguientes ejercicios, asuma que la clase correspondiente a cada árbol contiene el tamaño (número de llaves actualmente en el árbol) y una referencia al nodo raíz. Considere que para cada tipo de árbol además de la clase árbol es necesaria una clase nodo. Los valores que almacenarán los nodos son enteros.

1) Árboles Binarios de Búsqueda (ABB)

1) Crea una función(es) que determine(n) si existe una ruta de la raíz a alguna hoja tal que el número de claves impares de los nodos visitados es igual a *x*.

Así se usaría el método:

```
boolean exists1 = existsPath(root, 1);  // exists1 = false
boolean exists2 = existsPath(root, 2);  // exists2 = true
boolean exists3 = existsPath(root, 3);  // exists3 = true
boolean exists4 = existsPath(root, 4);  // exists4 = false
```




```
private boolean countOdd(Node<Integer> node, int goal, int count) {
    if (node.key % 2 != 0) count++;

    if (node.left == null && node.right == null) {
        if (count == goal) return true;
        else return false;
    }

    return countOdd(node.left, goal, count) || countOdd(node.right, goal, count);
}

public boolean existsPath(Node<Integer> root, int goal) {
    return countOdd(root, goal, 0);
}
```

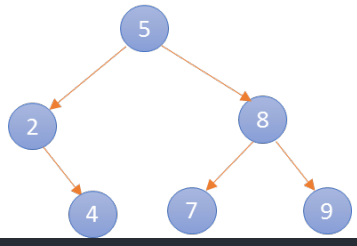
2) Crea una función(es) que determine(n) si un árbol binario es de búsqueda.

Así se usaría el método:

```
boolean isBST1 = isBST(root);  // isBST1 = true
```

Si cambiamos, por ejemplo, a 7 por 3, ó a 4 por 1:

```
boolean isBST2 = isBST(root);  // isBST2 = false
```



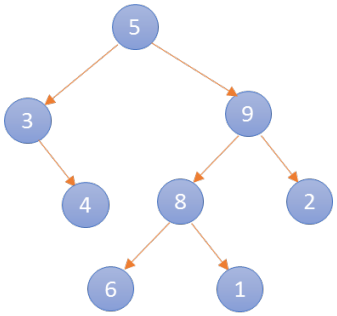

```
private Node<T> maximum(Node<T> node) {
    if (node == null) return null;

    while (node.right != null) node = node.right;

    return node;
}

public T maximum() {
    Node<T> max = maximum(root);
    return (max == null) ? null : max.key;
}

private Node<T> minimum(Node<T> node) {
    if (node == null) return null;

    while (node.left != null) node = node.left;
```

```
    return node;
}

public T minimum() {
    Node<T> min = minimum(root);
    return (min == null) ? null : min.key;
}


public boolean isBST(Node<T> root) {
    if (root == null) return true;


    if (root.left != null && maximum(root.left).key.compareTo(root.key) >= 0 ||
    root.right != null && minimum(root.right).key.compareTo(root.key) <= 0 ||
    !isBST(root.left) || !isBST(root.right)) return false;

    return true;
}
```

3) Crea una función(es) que construya(n) una lista de listas, tal que la lista *k* contiene las claves de todos los nodos que viven el nivel *k*, de izquierda a derecha.

Así se usaría el método:

```
List<List<Node>> nodeList = getNodeListPerLevel(root);
//  nodeList = [ [5], [3, 9], [4, 8, 2], [6, 1] ]
```




```
int depthBST(Node root) {
    if (root == null) return -1;

    int lDepth = depthBST(root.left);
    int rDepth = depthBST(root.right);

    if (lDepth > rDepth) return lDepth + 1;
    else return rDepth + 1;
}


void getNodesOnLevel(Node<T> node, int level, List<T> list) {
    if (node == null) return;
    if (level == 0) {
        list.add(node.key);
    } else {
        getNodesOnLevel(node.left, level - 1, list);
        getNodesOnLevel(node.right, level - 1, list);
    }
}

List<T> getNodesOnLevel(Node<T> root, int level) {
    List<T> list = new ArrayList<>();
    getNodesOnLevel(root, level, list);
    return list;
}

List<List<Node>> getNodeListPerLevel(Node root) {
    int depth = depthBST(root);
```

```
    List<List<Node>> res = new ArrayList<>(depth);

    for (int i = 0; i <= depth; i++) {
        res.add(getNodesOnLevel(root, i));
    }

    return res;
}
```

2) Árboles AVL

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho, y un entero que representa la altura del nodo.
Escriba un método (puede escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen todas las propiedades debidas:

1. A la izquierda de cada nodo sólo hay valores menores y a la derecha mayores. (O hijos nulos, en su defecto.)
2. El factor de balance de todos los nodos pertenece a {-1, 0, 1}

```java
public class AVL<T extends Comparable<? super T>> {
    private int height(Node<T> root) {
        if (root == null) return -1;

        int hl = height(root.left);
        int hr = height(root.right);

        if (hl > hr) return hl + 1;
        else return hr + 1;
    }

    private boolean isBalanced(Node<T> n) {
        if (n == null) return true;

        int hl = height(n.left), hr = height(n.right);

        if (Math.abs(hl - hr) <= 1
            && isBalanced(n.left) && isBalanced(n.right))
            return true;

        return false;

    }

    public boolean checkAVL(Node<T> root) {
        if (root == null) return true;

        // Primera condición: BST
        CheckBST<T> helper = new CheckBST<>(root);
        if (!(helper.isBST(root))) return false;

        // Segunda condición: balance
        if (!(isBalanced(root))) return false;

        return true;
    }
}
```

3) Árboles rojinegros

Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho, y un booleano que representa si el nodo es rojo.
Escriba un método (puede escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen todas las propiedades debidas:

1. A la izquierda de cada nodo sólo hay valores menores y a la derecha mayores. (O hijos nulos, en su defecto.)
2. Propiedad raíz (*root property* en las diapositivas).
3. Propiedad roja
4. Propiedad negra

```java
public class ARN<T extends Comparable<? super T>> {
    private boolean redProperty(NodeRN<T> node) {
        if (node == null) return true;

        if (node.left != null && node.isRed && node.left.isRed) return false;
        if (node.right != null && node.isRed && node.right.isRed) return false;

        return redProperty(node.left) && redProperty(node.right);
    }

    private int countBlacks(NodeRN<T> node, int count) {
        if (node == null) return count;

        if (!(node.isRed)) count++;

        int countL =  countBlacks(node.left, 0), countR = countBlacks(node.right, 0);

        return count + countL + countR;
    }

    private boolean blackProperty(NodeRN<T> node) {
        return (countBlacks(node.left, 0) - countBlacks(node.right, 0) == 0) ? true : false;
    }

    public boolean isRB(NodeRN<T> root) {
        if (root == null) return true;

        // Primera condición: BST
        CheckBST<T> helper = new CheckBST<>(root);
        if (!(helper.isBST(root))) return false;

        // Segunda condición: raíz negra
        if (root.isRed) return false;

        // Tercera condición: propiedad roja
        if (!(redProperty(root))) return false;

        // Cuarta condición: propiedad negra
        if (!(blackProperty(root))) return false;

        return true;
    }
}
```

4) Árboles B

La clase árbol además contiene un entero para el parámetro inmutable *t*. Cada nodo contiene los siguientes atributos: *parent*, *keys*, *children*, *n* (número de llaves actual), *isLeaf*.
Escriba un método (pueden escribir otros más y llamarlos) que reciba el árbol y determine si se cumplen las propiedades debidas:

1. Las llaves de todos los nodos siguen un orden no descendente (AKA: el orden "de siempre").
2. A la “izquierda” de cada llave sólo hay hijos con valores menores y a la “derecha” sólo hijos con valores mayores. (O hijos nulos, en su defecto.)
3. Todos los nodos (salvo, en su defecto, la raíz) contienen *t-1* llaves como mínimo.
4. Todos los nodos contienen *2t-1* llaves como máximo.
5. Todos los nodos internos tienen *n+1* hijos.
6. Todos los nodos internos tienen i*sLeaf* falso.
7. Todos los nodos hoja tienen *isLeaf* verdadero.

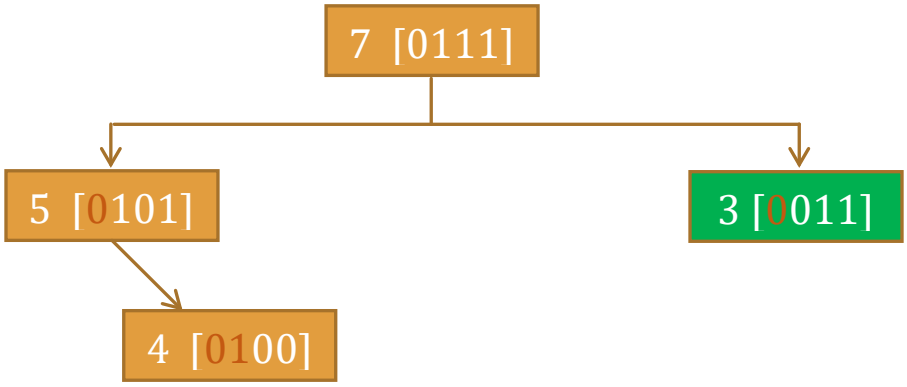
```java
import java.util.List;

public class BTree<T extends Comparable<? super T>> {

    // Primera condición
    private boolean orderedKeys(NodeBTree<T> node) {
        if (node == null) return true;

        for (int i = 0; i < node.key.size() - 1; i++) {
            if (node.key.get(i).compareTo(node.key.get(i + 1)) > 0) return false;
        }

        for (NodeBTree<T> child : node.children) {
            if (!(orderedKeys(child))) return false;
        }

        return true;
    }

    // Segunda condición
    private boolean holdsOrder(NodeBTree<T> node) {
        if (node == null) return true;

        for (int i = 0; i < node.key.size(); i++) {
            T currVal = node.key.get(i);

            for (int j = 0; j < node.children.size(); j++) {
                List<T> currChild = node.children.get(j).key;

                for (int k = 0; k < currChild.size(); k++) {
                    if (j <= i && currChild.get(i).compareTo(currVal) >= 0) return false;
                    if (j > i && currChild.get(i).compareTo(currVal) <= 0) return false;
                }
            }
        }

        for (NodeBTree<T> child : node.children) {
            if (!(holdsOrder(child))) return false;
        }

        return true;
    }

    // Tercera condición
    private boolean minimumKeys(NodeBTree<T> node, int mn) {
```

```java
        if (node == null || node.children == null) return true;

        for (NodeBTree<T> child : node.children) {
            if (child.key.size() < mn) return false;
            if (!(minimumKeys(child, mn))) return false;
        }

        return true;
    }

    // Cuarta condición
    private boolean maximumKeys(NodeBTree<T> node, int mx) {
        if (node == null || node.children == null) return true;

        for (NodeBTree<T> child : node.children) {
            if (child.key.size() < mx) return false;
            if (!(maximumKeys(child, mx))) return false;
        }

        return true;
    }

    // Quinta condición
    private boolean nChilds(NodeBTree<T> node) {
        if (node == null || node.children == null) return true;

        if (node.children.size() < (node.n + 1)) return false;

        for (NodeBTree<T> child : node.children) {
            if (!(nChilds(child))) return false;
        }

        return true;
    }

    // Sexta condición
    private boolean nodeNotLeaf(NodeBTree<T> node) {
        if (node == null) return true;

        if (node.children != null && node.isLeaf == true) return false;
        else {
            for (NodeBTree<T> child : node.children) {
                if (!(nodeNotLeaf(child))) return false;
            }

            return true;
        }
    }

    // Séptima condición
    private boolean leafNotNode(NodeBTree<T> node) {
        if (node == null) return true;

        if (node.children == null && node.isLeaf != true) return false;
        else {
            for (NodeBTree<T> child : node.children) {
                if (!(leafNotNode(child))) return false;
            }

            return true;
        }
    }

    public boolean isBTree(NodeBTree<T> root) {
        // Primera condición: nodos ascendentes
        if (!(orderedKeys(root))) return false;
```

```
        // Segunda condición: orden tipo BST
        if (!(holdsOrder(root))) return false;

        // Tercera condición: mínimo de llaves
        if (!minimumKeys(root, root.n - 1)) return false;

        // Cuarta condición: máximo de llaves
        if (!maximumKeys(root, (2 * root.n) - 1)) return false;

        // Quinta condición: número de hijos
        if (!(nChilds(root))) return false;

        // Sexta condición: nodos no son hojas
        if (!(nodeNotLeaf(root))) return false;

        // Séptima condición: hojas no son nodos
        if (!(leafNotNode(root))) return false;

        return true;
    }
}
```

5) Arboles Digitales de Búsqueda (*DST*)

La clase árbol además contiene un entero que indica la longitud de bits con la que se está trabajando.
Cada nodo contiene su dato o llave, referencia a sus hijos izquierdo y derecho.
Escriba un método (pueden escribir otros más y llamarlos) que reciba el árbol y determine si se cumple que todos los nodos en el árbol coinciden con su ruta de inserción (considerando la longitud de bits indicada en el atributo respectivo de la clase árbol). Observe que la raíz puede ser cualquier valor. Por ejemplo, en el siguiente árbol (*casi* correcto), considerando una longitud de 4 bits las claves *5* y *4* sí son aptas para su ruta de inserción, pero *3* no.




```java
public class DSTree<T extends Comparable<? super T>> {
    private boolean value(int n, int pos) { // Obtiene bits de derecha a izquierda
        return (((n >> pos) & 1) == 0) ? true : false;
    }

    private boolean check(Node<Integer> node, int bitLen, int level) {
        if (node == null) return true;

        if (node.left != null && !(value((Integer)node.left.key, bitLen - (level + 1)))) return false;
        if (node.right != null && value((Integer)node.right.key, bitLen - (level + 1))) return false;

        return check(node.left, bitLen, level + 1) && check(node.right, bitLen, level + 1);
    }

    public boolean isDST(Node<Integer> root, int bitLen) {
        return check(root, bitLen, 0);
    }
}
```