Commencé le	mardi 20 décembre 2022, 09:37
État	Terminé
Terminé le	mercredi 28 décembre 2022, 18:49
Temps mis	8 jours 9 heures
Note	**30,00** sur 30,00 (**100**%)

Description

Pour ce rendu sur l'algorithme de Quine–McCluskey nous avons essayé de vous guider en décomposant la solution en 5 classes, présentées ci-dessous tout en vous laissant libres autant que possible de vos choix de structures. Cependant, pour que les tests passent, vous devez respecter les signatures des méthodes qui vous sont demandées. Vous pouvez/devez bien évidemment ajouter des variables d'instance et méthodes pour vous aider à les implémenter.

1. **Minterm** : un minterm encapsule le codage en base deux et l'unification de 2 termes
2. **MintertermCategory** : Une catégorie de Minterm a la responsabilité de regrouper des minterms yanat le même nombre de un et de fusionner deux catégories. Nous avons fait le choix de ne pas parler de classes mais de catégories pour vous éviter des conflits sur le terme de "class".
3. **CatagoryManager** : un gestionnaire de catégories a la responsabilité de créer les catégories en fonction d'une liste de Minterm et de boucler sur l'ensemble des catégories pour obtenir la liste des minterms résultants.
4. **PrimeImplicantChart** : il a la responsabilité de construire la grille et de déterminer les implicants essentiels et autres. On n'abordera pas l'algorithme de Petrick
5. **QMC** : Il a la responsabilité construire les minterms à partir d'une liste de nombre en notation décimal et de déclencher tous les calculs

Les classes suivantes sont importées dans l'environnement.
 //Pour laisser le choix des structures aux etudiants
import java.util.*;
import java.util.stream.*;

Question 1

Correct

Note de 10,00 sur 10,00

Dans les questions qui suivent, on désire écrire l'algorithme de Quine–McCluskey étudié en cours d'informatique théorique.

Dans cette première étape, vous devez créer la classe **Minterm.java**

- les seules valeurs codées dans un Minterm sont 0,1 et -1 (-1 pour indiquer 0 ou 1)
- les méthodes à implémenter sont celles données dans la partie pré-remplie.
- Attention, le equals doit être implémenté pour ne comparer que le contenu du minterm sous sa forme binaire, quelque soit la valeur des autres éléments. m0 et m1 sont égaux s'ils ont la même représentation binaire par exemple 1 -1 1 même s'ils ne sont pas issus des mêmes combinaisons inititiales.

Nous la recopions ici pour aider ceux qui perdraient des bouts ;-)

```
public class Minterm {
    /**
     *
     * @param decimal   the decimal number for which we want to calculate the number of bits necessary to represent it
     * @return          the minimum number of bits needed to encode this decimal in binary.
     */
    public static int numberOfBitsNeeded(int decimal) {
        return 0;
    }

    /*********************************************************
     * Management of the minterms structure
     ******************************************************** */


    /**
     * returns all the numbers that were used to build this minterm.
     * For example, [0*00] may have been created from 0 and 2 (* = -1)
     * @return all the numbers that were used to build this minterm.
     */
    public Collection getCombinations() {
        return new HashSet<>();
    }



    /**
     * marks the minterm as used to build another minTerm
     */
    public void mark(){

    }

    /**
     *
     * @return true if the minterm has been used to build another minterm, false otherwise.
     */
    public boolean isMarked(){
        return false;
    }

    /*********************************************************
     * Management of the minterms contents
     ******************************************************** */
    /**
     *
     * @return return the number of 0 in the minterm
     */
    public int numberOfZero() {
        return -1;
    }

    /**
     *
     * @return return the number of 1 in the minterm
     */
    public int numberOfOne() {
        return -1;
    }



    /*********************************************************
     * Equality
     ******************************************************** */

    /**
     * @param o
     * @return true if the representation in base 2 is the same. Ignore the other elements.
     */
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Minterm minterm = (Minterm) o;
        return this == o;
    }

    @Override
    public int hashCode() {
        return 0;
    }



/* -------------------------------------------------------------------------
        Constructors
 ------------------------------------------------------------------------- */
    /**
     * Construct a minterm corresponding to the decimal passed in parameter
     * and encode it on the given number of bits.
     * The associated combination then contains decimal.
     * @param decimal       the decimal value representing the minterm
     * @param numberOfBits  the number of bits of encoding of the decimal
     */
     public Minterm(int decimal, int numberOfBits) {
    }


    /**
     * Builds a minterm from its representation in binary which can contain -1.
     * This constructor does not update the associated combinations.
     * The size of the binary representation corresponds to the number of parameters (binary.length).
     * @param binary
     */
    protected Minterm(int... binary) {
    }



  /**
     * Compute the string showing the binary form of the minterm.
     * For example, "101" represents the minterm corresponding to 5,
     * while "1-1" represents a minterm resulting, for example from the merge of 5 and 7 (1 -1 1)
     * @return the string
     */
   @Override
    public String toString() {
        return "";
    }



/* -------------------------------------------------------------------------
        Binary <-> Decimal
 ------------------------------------------------------------------------- */

    /**
     * Calculates the integer value of the binary representation.
     * But in case one of the binary elements is -1, it returns -1.
     * This method is private because it should not be used outside this class.
     * @returns the value of the minterm calculated from its binary representation.
     */
    public int toIntValue(){
        int res = 0;
        return res;
    }


   /* -------------------------------------------------------------------------
        Merge
 ------------------------------------------------------------------------- */


     /**
     * create a Minterm from the merge of two Minterms when it is posssible otherwise return null
     * Attention two minterms can only be merged if
     *  - they differ by one value at most.
     *  - they are of the same size.
     *  If a merge is possible, the returned minterm
     *  - has the same binary representation as the original minterm, but where at most one slot has been replaced by -1,
     *  - and it has, for the combinations, the merge of the combinations of both minterms this and other)
     *  - and the both mindterms  this and other are marked
     * @param other is another Minterm which we try to unify
     * @return a new Minterm when it is possible to unify, else null * @param other is another Minterm which we try to merge
     * @return a new Minterm when it is possible to merge, else null
     */
    public Minterm merge(Minterm other) {
        return null;
    }
```

```
}
```

Par exemple:

Test	Résultat

//Test la construction d'un Minterm à partir d'un tableau via le toString
Minterm minterm = new Minterm(1,1,0,0,1);
assertEquals("11001",minterm.toString());
	11001 equals 11001?
	true

//Test la construction d'un minterm
//à partir d'un décimal et la taille de la représentation en base 2
Minterm minterm = new Minterm(25,5);
assertEquals("11001",minterm.toString());
 	
minterm = new Minterm(8,4);
assertEquals("1000",minterm.toString());
 	
minterm = new Minterm(11,4);
assertEquals("1011", minterm.toString());
	11001 equals 11001?
	true
	1000 equals 1000?
	true
	1011 equals 1011?
	true

//Test a simple merge
Minterm minterm1 = new Minterm(1,1,0,0,1);
Minterm minterm2 = new Minterm(1,1,0,0,0);
Minterm res = minterm1.merge(minterm2);
assertTrue(minterm1.isMarked());
assertTrue(minterm2.isMarked());
assertFalse(res.isMarked());
assertEquals("1100-", res.toString());
	true
	true
	false
	1100- equals 1100-?
	true

//Test numberOfBitsNeeded
assertEquals(1, Minterm.numberOfBitsNeeded(0));
assertEquals(1, Minterm.numberOfBitsNeeded(1));
assertEquals(2, Minterm.numberOfBitsNeeded(3));
assertEquals(3, Minterm.numberOfBitsNeeded(6));
assertEquals(4, Minterm.numberOfBitsNeeded(15));
assertEquals(MASK, Minterm.numberOfBitsNeeded(MASKED_NUMBER));
	1 equals 1?
	true
	1 equals 1?
	true
	2 equals 2?
	true
	3 equals 3?
	true
	4 equals 4?
	true
	5 equals 5?
	true

Minterm m = new Minterm(15, 4);
assertEquals(0, m.numberOfZero());
assertEquals(4, m.numberOfOne());
	0 equals 0?
	true
	4 equals 4?
	true

//Test Equals
 	
Minterm minterm1 = new Minterm(5, 3);
Minterm minterm2 = new Minterm(5, 3);
assertEquals(minterm1, minterm2);
minterm1.mark();
assertEquals(minterm1, minterm2);
	101 equals 101?
	true
	101 equals 101?
	true

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```java


public class BinaryTools {
    public static int[] getBinary(int x, int nbBits) {
        // Prepend with zeros to fill the array until NbBits
        int[] binary = new int[nbBits];
        int i = nbBits - 1;
        while(x > 0) {
            binary[i] = x % 2;
            x /= 2;
            i--;
        }
        return binary;
    }

    public static int getDecimal(List<Integer> binary) {
        int decimal = 0;
        for(int i = 0; i < binary.size(); i++)
            decimal += binary.get(i) * Math.pow(2, binary.size() - i - 1);
        return decimal;
    }
    public static int[] getBinary(int x) {
        return Integer.toString(x, 2).chars().map(c -> c - '0').toArray();
    }
}


public class Minterm {

    private List<Integer> binary;
    private boolean hasBeenUsed;

    /**
     *
     * @param decimal   the decimal number for which we want to calculate the number of bits necessary to represent it
     * @return          the minimum number of bits needed to encode this decimal in binary.
     */
    public static int numberOfBitsNeeded(int decimal) {
        return Integer.toBinaryString(decimal).length();
    }

    /*********************************************************
     * Management of the minterms structure
     ********************************************************* */


    /**
     * returns all the numbers that were used to build this minterm.
     * For example, [0*00] may have been created from 0 and 2 (* = -1)
     * @return all the numbers that were used to build this minterm.
     */
```

	Test	Résultat attendu	Résultat obtenu	

✓	//Test la construction d'un Minterm à partir d'un tableau via le toString
	Minterm minterm = new Minterm(1,1,0,0,1);
	assertEquals("11001",minterm.toString());
		11001 equals 11001?
		true
			11001 equals 11001?
			true
				✓

✓	//Test la construction d'un minterm
	//à partir d'un décimal et la taille de la représentation en base 2
	Minterm minterm = new Minterm(25,5);
	assertEquals("11001",minterm.toString());
	
	minterm = new Minterm(8,4);
	assertEquals("1000",minterm.toString());
	
	minterm = new Minterm(11,4);
	assertEquals("1011", minterm.toString());
		11001 equals 11001?
		true
		1000 equals 1000?
		true
		1011 equals 1011?
		true
			11001 equals 11001?
			true
			1000 equals 1000?
			true
			1011 equals 1011?
			true
				✓

✓	//Test la construction d'un minterm
	//à partir d'un décimal et vérifie la combinaison associée
	Minterm minterm = new Minterm(26,5);
	assertEquals("11010",minterm.toString());
	assertTrue(minterm.getCombinations().contains(26));
		11010 equals 11010?
		true
		true
			11010 equals 11010?
			true
			true
				✓

✓	//Test a simple merge
	Minterm minterm1 = new Minterm(1,1,0,0,1);
	Minterm minterm2 = new Minterm(1,1,0,0,0);
	Minterm res = minterm1.merge(minterm2);
	assertTrue(minterm1.isMarked());
	assertTrue(minterm2.isMarked());
	assertFalse(res.isMarked());
	assertEquals("1100-", res.toString());
		true
		true
		false
		1100- equals 1100-?
		true
			true
			true
			false
			1100- equals 1100-?
			true
				✓

✓	//Test numberOfBitsNeeded
	assertEquals(1, Minterm.numberOfBitsNeeded(0));
	assertEquals(1, Minterm.numberOfBitsNeeded(1));
	assertEquals(2, Minterm.numberOfBitsNeeded(3));
	assertEquals(3, Minterm.numberOfBitsNeeded(6));
	assertEquals(4, Minterm.numberOfBitsNeeded(15));
	assertEquals(MASK, Minterm.numberOfBitsNeeded(MASKED_NUMBER));
		1 equals 1?
		true
		1 equals 1?
		true
		2 equals 2?
		true
		3 equals 3?
		true
		4 equals 4?
		true
		5 equals 5?
		true
			1 equals 1?
			true
			1 equals 1?
			true
			2 equals 2?
			true
			3 equals 3?
			true
			4 equals 4?
			true
			5 equals 5?
			true
				✓

✓	//Test le cas où le minterm contient -1
	Minterm minterm = new Minterm(1,-1,1);
	assertEquals(-1,minterm.toIntValue());
		-1 equals -1?
		true
			-1 equals -1?
			true
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	`//Test merge of merged minterms`
	`Minterm minterm1 = new Minterm(1,-1,0,0,1);`
	`Minterm minterm2 = new Minterm(1,-1,1,0,1);`
	`Minterm res = minterm1.merge(minterm2);`
	`assertEquals("1--01", res.toString());`
		`1--01 equals 1--01?`
		`true`
			`1--01 equals 1--01?`
			`true`
				✓

✓	`//Test merge of non unifiable minterms`
	`Minterm minterm1 = new Minterm(1,-1,0,0,1);`
	`Minterm minterm2 = new Minterm(1,1,1,0,1);`
	`Minterm res = minterm1.merge(minterm2);`
	`assert(res==null);`
	`        assertFalse(minterm1.isMarked());`
	`        assertFalse(minterm2.isMarked());`
		`false`
		`false`
			`false`
			`false`
				✓

✓	`        //Test merge and combinations and marks`
	`Minterm minterm1 = new Minterm(5,3);`
	`Minterm minterm2 = new Minterm(7,3);`
	`Minterm res = minterm1.merge(minterm2);`
	`assertEquals("1-1", res.toString());`
	`        assertEquals(2, res.getCombinations().size());`
	`        assertTrue(res.getCombinations().contains(5));`
	`        assertTrue(res.getCombinations().contains(7));`
	`        assertFalse(res.isMarked());`
	`        assertTrue(minterm1.isMarked());`
	`        assertTrue(minterm2.isMarked());`
		`1-1 equals 1-1?`
		`true`
		`2 equals 2?`
		`true`
		`true`
		`true`
		`false`
		`true`
		`true`
			`1-1 equals 1-1?`
			`true`
			`2 equals 2?`
			`true`
			`true`
			`true`
			`false`
			`true`
			`true`
				✓

✓	`        //Test merge , combinations and marks`
	`        Minterm minterm1 = new Minterm(5, 3);`
	`        Minterm minterm2 = new Minterm(7, 3);`
	`        Minterm res = minterm1.merge(minterm2);`
	`        System.out.println("merge of 5 and 7 : " + res);`
	`        Minterm minterm6 = new Minterm(6, 3);`
	`        Minterm minterm4 = new Minterm(4, 3);`
	`        Minterm resBis = minterm6.merge(minterm4);`
	`        System.out.println("merge of 4 and 6 : " + resBis);`
	`        Minterm resTer = res.merge(resBis);`
	`        System.out.println("merge of 4 and 7 and 5: " + resTer);`
	`        assertEquals(4, resTer.getCombinations().size());`
	`        assertTrue(resTer.getCombinations().contains(5));`
	`        assertTrue(resTer.getCombinations().contains(6));`
	`        assertTrue(resTer.getCombinations().contains(7));`
	`        assertTrue(resTer.getCombinations().contains(4));`
		`merge of 5 and 7 : 1-1`
		`merge of 4 and 6 : 1-0`
		`merge of 4 and 7 and 5: 1-`
		`-`
		`4 equals 4?`
		`true`
		`true`
		`true`
		`true`
		`true`
			`merge of 5 and 7 : 1-1`
			`merge of 4 and 6 : 1-0`
			`merge of 4 and 7 and 5: 1-`
			`-`
			`4 equals 4?`
			`true`
			`true`
			`true`
			`true`
			`true`
				✓

✓	`//Test numberOfZero and numberOfOne`
	` Minterm m = new Minterm(0, 2);`
	` assertEquals(2, m.numberOfZero());`
	` assertEquals(0, m.numberOfOne());`
	
	`Minterm m7 = new Minterm(7, 3);`
	` assertEquals(0, m7.numberOfZero());`
	` assertEquals(3, m7.numberOfOne());`
	
	`Minterm m9 = new Minterm(9, 4);`
	` assertEquals(2, m9.numberOfZero());`
	` assertEquals(2, m9.numberOfOne());`
		`2 equals 2?`
		`true`
		`0 equals 0?`
		`true`
		`0 equals 0?`
		`true`
		`3 equals 3?`
		`true`
		`2 equals 2?`
		`true`
		`2 equals 2?`
		`true`
			`2 equals 2?`
			`true`
			`0 equals 0?`
			`true`
			`0 equals 0?`
			`true`
			`3 equals 3?`
			`true`
			`2 equals 2?`
			`true`
			`2 equals 2?`
			`true`
				✓

✓	`Minterm m = new Minterm(15, 4);`
	`assertEquals(0, m.numberOfZero());`
	` assertEquals(4, m.numberOfOne());`
		`0 equals 0?`
		`true`
		`4 equals 4?`
		`true`
			`0 equals 0?`
			`true`
			`4 equals 4?`
			`true`
				✓

✓	`//Test Equals`
	
	` Minterm minterm1 = new Minterm(5, 3);`
	`        Minterm minterm2 = new Minterm(5, 3);`
	`        assertEquals(minterm1, minterm2);`
	`        minterm1.mark();`
	`        assertEquals(minterm1, minterm2);`
		`101 equals 101?`
		`true`
		`101 equals 101?`
		`true`
			`101 equals 101?`
			`true`
			`101 equals 101?`
			`true`
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	`assertEquals(3, Minterm.numberOfBitsNeeded(4));`
	`assertEquals(3, Minterm.numberOfBitsNeeded(7));`
	`assertEquals(1, Minterm.numberOfBitsNeeded(0));`
	`assertEquals(1, Minterm.numberOfBitsNeeded(1));`
	`assertEquals(2, Minterm.numberOfBitsNeeded(2));`
	`assertEquals(4, Minterm.numberOfBitsNeeded(15));`
		`3 equals 3?`
		`true`
		`3 equals 3?`
		`true`
		`1 equals 1?`
		`true`
		`1 equals 1?`
		`true`
		`2 equals 2?`
		`true`
		`4 equals 4?`
		`true`
			`3 equals 3?`
			`true`
			`3 equals 3?`
			`true`
			`1 equals 1?`
			`true`
			`1 equals 1?`
			`true`
			`2 equals 2?`
			`true`
			`4 equals 4?`
			`true`
				✓

Tous les tests ont été réussis ! ✓

▸ **Solution de l'auteur de la question (Java)**

Correct

Note pour cet envoi : 10,00/10,00.

Question 2

Correct

Note de 4,00 sur 4,00

Une categorie de Minterms (**MintermCategory**) contient un ensemble de Minterm qui ont le même nombre de 1.

Nous la définissons comme une ArrayList de Minterm.

Implémenter les méthodes définies dans la réponse pré-remplie.

```
public class MintermCategory extends ArrayList {



    /**
     * It computes the list of minterms m, such that :
     * - either m results from  merging a minterm from the category "this" with a minterm from the other category ;
     * - either m belongs to the current category (this) and could not be unified with a minterm of the other category
     * @param otherCategory
     * @return  the list of merged minterms
     */
    public List merge(MintermCategory otherCategory){
        List result = new ArrayList<>();
        return result;
    }

}
```

Par exemple:

Test	Résultat

//Merge Categories of only one elements
MintermCategory m0Class = new MintermCategory();
Minterm m0 = new Minterm(0,4);
m0Class.add(m0);
MintermCategory m1Class = new MintermCategory();
Minterm m1 = new Minterm(1,4);
m1Class.add(m1);
List<Minterm> res = m1Class.merge(m0Class);
assertTrue(res.contains(new Minterm(0, 0, 0, -1)));
assertTrue(m0.isMarked());
assertTrue(m1.isMarked());
        Collection<Integer> combinations = res.get(0).getCombinations();
        assertTrue(combinations.contains(0));
        assertTrue(combinations.contains(1));
	true
	true
	true
	true
	true

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
import java.util.*;

public class MintermCategory extends ArrayList<Minterm> {


    private int numberOfOnes;
    private List<Minterm> minterms;

    public MintermCategory(){
        minterms = new ArrayList<>();
    }

    public MintermCategory(int numberOfOnes) {
        this.numberOfOnes = numberOfOnes;
        minterms = new ArrayList<>();
    }

    public int getNumberOfOnes() {
        return numberOfOnes;
    }

    /**
     * It computes the list of minterms m, such that :
     * - either m results from  merging a minterm from the category "this" with a minterm from the other category ;
     * - either m results from merging a minterm from the category `this` with a minterm from the other category ;
     * - either m belongs to the current category (this) and could not be unified with a minterm of the other category
     *
     * @param otherCategory
     * @return the list of merged minterms
     */
    public List<Minterm> merge(MintermCategory otherCategory) {
        List<Minterm> res = new ArrayList<>();
        for (Minterm m1 : this) {
            boolean merged = false;
            for (Minterm m2 : otherCategory) {
                Minterm mergedMinterm = m1.merge(m2);
                if (mergedMinterm != null) {
                    merged = true;
                    res.add(mergedMinterm);
                }
            }
            if (!merged) {
                res.add(m1);
            }
        }
        return res;
    }

}
```

	Test	Résultat attendu	Résultat obtenu	

✓	//Merge Categories of only one elements
	MintermCategory m0Class = new MintermCategory();
	Minterm m0 = new Minterm(0,4);
	m0Class.add(m0);
	MintermCategory m1Class = new MintermCategory();
	Minterm m1 = new Minterm(1,4);
	m1Class.add(m1);
	List<Minterm> res = m1Class.merge(m0Class);
	assertTrue(res.contains(new Minterm(0, 0, 0, -1)));
	assertTrue(m0.isMarked());
	assertTrue(m1.isMarked());
	Collection<Integer> combinations = res.get(0).getCombinations();
	assertTrue(combinations.contains(0));
	assertTrue(combinations.contains(1));
		true
		true
		true
		true
		true
			true
			true
			true
			true
			true
				✓

✓	//merge of categories
	MintermCategory m0Class = new MintermCategory();
	Minterm m0 = new Minterm(0,4);
	m0Class.add(m0);
	
	MintermCategory m1Class = new MintermCategory();
	Minterm m1 = new Minterm(1,4);
	m1Class.add(m1);
	Minterm m2 =new Minterm(2,4);
	m1Class.add(m2);
	Minterm m4 = new Minterm(4,4);
	m1Class.add(m4);
	Minterm m8 = new Minterm(8,4);
	m1Class.add(m8);
	
	List<Minterm> res = m0Class.merge(m1Class);
	
	assertTrue(m0.isMarked());
	assertEquals(4,res.size());
	assertTrue(res.contains(new Minterm(-1,0,0,0)));
	assertTrue(res.contains(new Minterm(0,0,0,-1)))
		true
		4 equals 4?
		true
		true
		true
			true
			4 equals 4?
			true
			true
			true
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	`//merge not possible`
	`MintermCategory mclass = new MintermCategory();`
	`Minterm m1 = new Minterm(-1,1,0);`
	`mclass.add(m1);`
	`Minterm m2 = new Minterm(1,-1,0);`
	`mclass.add(m2);`
	
	`MintermCategory m2class = new MintermCategory();`
	`m2class.add(new Minterm(0,0,-1));`
	
	`List<Minterm> res = mclass.merge(m2class);`
	`assertEquals(2,res.size());`
	`assertTrue( res.contains(m1) ) ;`
	`assertTrue( res.contains(m2) ) ;`
	
	`m2class = new MintermCategory();`
	`m1 = new Minterm(0,0,-1);`
	`m2class.add(m1);`
	`mclass = new MintermCategory();`
	`mclass.add(new Minterm(-1,1,0));`
	`mclass.add(new Minterm(1,-1,0));`
	`res = m2class.merge(mclass);`
	`assertEquals(1,res.size());`
	`assertTrue( res.contains(m1) ) ;`
	`assertFalse( res.contains(m2) );`
		`2 equals 2?`
		`true`
		`true`
		`true`
		`1 equals 1?`
		`true`
		`true`
		`false`
			`2 equals 2?`
			`true`
			`true`
			`true`
			`1 equals 1?`
			`true`
			`true`
			`false`
				✓

Tous les tests ont été réussis ! ✓

▸ **Solution de l'auteur de la question (Java)**

Correct

Note pour cet envoi : 4,00/4,00.

Question **3**

Correct

Note de 5,00 sur 5,00

Un gestionnaire de categories a la responsabilité
- de créer les catégories à partir d'une liste de minterms et du nombre de bits d'encodage.
- de fusionner les catégories n'ayant qu'un un d'écarts et de retourner les minterms résultants de la fusion.

Vous devez implémenter chacune de ces méthodes conformément aux spécifications qui vous sont données.

```
public class CategoryManager {

    /**
     * CategoryManager : compute the categories from a list of minterms according to the number of 11
     * @param mintermList
     * @return
     */
    public CategoryManager(List mintermList, int nbBits) {

    }


    public int numberOfCategories(){
         return 0;
    }


    /**
     *
     * @param numberOfOne
     * @return the Category Of Minterms containing  numberOfOne
     */
    public MintermCategory getCategory(int numberOfOne){
        return null;
    }


    /**
     *  isLastTurn()
     * @return true is it's the last turn.
     */
    public boolean isLastTurn() {
        return true;
    }

    /**
     * Merge the categories two by two if they have only one "one" between them.
     * The minterms are the result of the merging of the categories.
     * Be careful for a category of n "one", if the category of "n+1" has no minterms,
     *    you must recover the minterms of the category of n "one" which were not marked.
     * This is the last round if no terms could be merged.
     * @return the merged terms
     */
    public List mergeCategories() {
        List res = new ArrayList<>();
        return res;
    }

}
```

Par exemple:

Test	**Résultat**

`//Merge categories as given in blog`
`List<Minterm> list = getBlogMinterms();`
`System.out.println(list);`
`int nbBits = 4;`
`CategoryManager manager = new CategoryManager(list,nbBits );`
`List<Minterm> res = manager.mergeCategories();`
`assertEquals(16,res.size());`
`assertFalse(manager.isLastTurn());`
	`[0000, 0001, 0010, 0100, 0110, 1000, 1001, 1100, 1101, 1110, 1111]`
	`16 equals 16?`
	`true`
	`false`

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
import java.util.*;

public class CategoryManager {

    private List<MintermCategory> categories;

    private int nbBits;



    /**
     * CategoryManager : compute the categories from a list of minterms according to the number of 11
     * @param mintermList
     * @return
     */
    public CategoryManager(List<Minterm> mintermList, int nbBits) {
        this.nbBits = nbBits;
        categories = new ArrayList<>();
        for (int i = 0; i <= nbBits; i++) {
            categories.add(new MintermCategory());
        }
        for (Minterm m : mintermList) {
            categories.get(m.numberOfOne()).add(m);
        }


    }


    public int numberOfCategories(){
        return categories.size();
    }


    /**
     *
     * @param numberOfOne
     * @return the Category Of Minterms containing  numberOfOne
     */
    public MintermCategory getCategory(int numberOfOne){
        return categories.get(numberOfOne);
    }


    /**
     *  isLastTurn()
     * @return true is it's the last turn.
     */
    public boolean isLastTurn() {
        if (categories.get(nbBits).size() == 0) {
            return true;
        } else {
```

	Test	Résultat attendu	Résultat obtenu	

✓	//Merge categories as given in blog
	List<Minterm> list = getBlogMinterms();
	System.out.println(list);
	int nbBits = 4;
	CategoryManager manager = new CategoryManager(list,nbBits);
	List<Minterm> res = manager.mergeCategories();
	assertEquals(16,res.size());
	assertFalse(manager.isLastTurn());
		[0000, 0001, 0010, 0100, 0110, 1000, 1001, 1100, 1101, 1110, 1111]
		16 equals 16?
		true
		false
			[0000, 0001, 0010, 0100, 0110, 1000, 1001, 1100, 1101, 1110, 1111]
			16 equals 16?
			true
			false
				✓

✓	int nbBits = 3;
	List<Minterm> list = createMintermList(3,0,1, 2, 3,4,7);
	CategoryManager manager = new CategoryManager(list,nbBits);
	MintermCategory category = manager.getCategory(1);
	assertEquals(3,category.size());
	assertEquals(1,manager.getCategory(3).size());
	List<Minterm> res = manager.mergeCategories();
	assertEquals(6,res.size());
	assertTrue(res.contains(new Minterm(0,1,-1)));
	assertTrue(res.contains(new Minterm(0,0,-1)));
	assertTrue(res.contains(new Minterm(-1,1,1)));
	assertFalse(manager.isLastTurn());
		3 equals 3?
		true
		1 equals 1?
		true
		6 equals 6?
		true
		true
		true
		true
		false
			3 equals 3?
			true
			1 equals 1?
			true
			6 equals 6?
			true
			true
			true
			true
			false
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	`int nbBits = 3;`
	`List<Minterm> list = createMintermList(`
	`new int[]{-1, 1, 0},`
	`new int[]{1, -1, 0},`
	`new int[]{0, 0, -1},`
	`new int[]{-1, 0, 0});`
	`CategoryManager manager = new`
	`CategoryManager(list,nbBits );`
	`assertEquals(2,manager.getCategory(1).size());`
	`assertEquals(2,manager.getCategory(0).size());`
	`List<Minterm> res = manager.mergeCategories();`
	`assertEquals(3,res.size());`
	`assertTrue(res.contains(new Minterm(1, -1, 0)));`
	`assertTrue(res.contains(new Minterm(-1, -1,`
	`0)));`
	`assertTrue(res.contains(new Minterm(0, 0, -1)));`
		`2 equals 2?`
		`true`
		`2 equals 2?`
		`true`
		`3 equals 3?`
		`true`
		`true`
		`true`
		`true`
			`2 equals 2?`
			`true`
			`2 equals 2?`
			`true`
			`3 equals 3?`
			`true`
			`true`
			`true`
			`true`
				✓

✓	`int value = 10;`
	`int nbBits = 4;`
	`List<Minterm> list = Arrays.asList(new`
	`Minterm(value,nbBits));`
	
	`CategoryManager manager = new`
	`CategoryManager(list,nbBits );`
	`assertEquals(new Minterm(value,nbBits),`
	`manager.getCategory(2).get(0));`
		`1010 equals 1010?`
		`true`
			`1010 equals 1010?`
			`true`
				✓

Tous les tests ont été réussis ! ✓

▸ **Solution de l'auteur de la question (Java)**

Correct

Note pour cet envoi : 5,00/5,00.

Question 4

Correct

Note de 6,00 sur 6,00

Un **PrimeImplicantChart** a la responsabilité
- de créer la grille mettant en relation les combinaisons et les minterms à partir des implicants originaux et les minterms résultants des fusions de termes en utilisant les catégories
- d'extraire les minterms essentiels
- d'extraire les autres minterms en utilisant une stratégie simplifiée (un parmi ceux restant).

Vous devez implémenter chacune de ces méthodes conformément aux spécifications qui vous sont données.

```
public class PrimeImplicantChart {

    /**
     * Initializes the grid with the original minterms and values.
     * @param values        Initial decimal values (they are also included in the combinations of minterms).
     * @param mintermList   The list of minterms reduced by merging the categories
     */
    public PrimeImplicantChart(int [] values, List mintermList) {

    }



    /**
     * extracts only the essential minterms; they correspond to the minterms that are the only ones to represent one of the initial values.
     * @return essential minterm list
     */
    public  List extractEssentialPrimeImplicants() {

        return null;
    }


    /**
     * After removing the initial values covered by the essential minterms,
     * choose a minterm for each remaining value not covered by an essential minterm.
     */
    public  List extractRemainingImplicants() {

        return null;
    }

    }
```

Par exemple:

Test	Résultat

// extract Essential And Other Prime Implicants
 	
List<Minterm> list = new ArrayList<>();
Minterm m0 = new Minterm(-1,0,0);
list.add(m0);
m0.addCombination(0,4);
Minterm m1= new Minterm(-1,1,1);
list.add(m1);
m1.addCombination(3,7);
Minterm m2= new Minterm(1,0,-1);
list.add(m2);
m2.addCombination(4,5);
Minterm m3= new Minterm(1,-1,1);
list.add(m3);
m3.addCombination(7,5);
 	
PrimeImplicantChart pmc = new PrimeImplicantChart(new int[]{0,3,4,5,7},list);
List<Minterm> essential = pmc.extractEssentialPrimeImplicants();
assertEquals(2,essential.size());
assertTrue(essential.contains(m0));
assertTrue(essential.contains(m1));
// other
List<Minterm> implicants = pmc.extractRemainingImplicants();
 	
assertEquals(1,implicants.size());
assertTrue(list.containsAll(implicants));
assertTrue(implicants.contains(m2)||implicants.contains(m3));
	2 equals 2?
	true
	true
	true
	1 equals 1?
	true
	true
	true

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```java
import java.util.*;
import java.util.stream.Collectors;

public class PrimeImplicantChart {

    private final List<Minterm> minTerms;
    private final List<Integer> ints;

    /**
     * Initializes the grid with the original minterms and values.
     *
     * @param values      Initial decimal values (they are also included in the combinations of minterms).
     * @param mintermList The list of minterms reduced by merging the categories
     */
    public PrimeImplicantChart(int[] ints, List<Minterm> list) {
        this.minTerms = list;
        List<Integer> result = new ArrayList<>();
        for (int i : ints) {
            Integer integer = i;
            result.add(integer);
        }
        this.ints = result;
    }

    private Map<Integer, List<Minterm>> getOccurences() {
        var res = new HashMap<Integer, List<Minterm>>();
        for (Integer i : ints) {
            res.put(i, new ArrayList<>());
            for (Minterm m : minTerms) {
                if (m.getCombinations().contains(i)) {
                    res.get(i).add(m);
                }
            }
        }
        return res;
    }

    /**
     * extracts only the essential minterms; they correspond to the minterms that are the only ones to represent one o
     *
     * @return essential minterm list
     */
    public List<Minterm> extractEssentialPrimeImplicants() {
        List<Minterm> list = new ArrayList<>();
        Set<Minterm> uniqueValues = new HashSet<>();
        for (Map.Entry<Integer, List<Minterm>> e : getOccurences().entrySet()) {
            if (e.getValue().size() == 1) {
                for (Minterm minterm : e.getValue()) {
                    if (uniqueValues.add(minterm)) {
                        list.add(minterm);
                    }
```

	Test	Résultat attendu	Résultat obtenu	

✓	//extract Essential PrimeImplicants
	
	List<Minterm> list = new ArrayList<>();
	Minterm m0 = new Minterm(1,-1,1);
	list.add(m0);
	Minterm m1= new Minterm(1,1,-1);
	list.add(m1);
	m0.addCombination(5,7);
	m1.addCombination(6,7);
	PrimeImplicantChart pmc = new PrimeImplicantChart(new int[]{5,6,7},list);
	List<Minterm> essential = pmc.extractEssentialPrimeImplicants();
	assertEquals(2,essential.size());
	assertTrue(essential.contains(m0));
	assertTrue(essential.contains(m1));
	
	// Not other
	
	List<Minterm> implicants = pmc.extractRemainingImplicants();
	assertEquals(0,implicants.size());
		2 equals 2?
		true
		true
		true
		0 equals 0?
		true
			2 equals 2?
			true
			true
			true
			0 equals 0?
			true
				✓

✓	// extract Essential And Other Prime Implicants
	
	List<Minterm> list = new ArrayList<>();
	Minterm m0 = new Minterm(-1,0,0);
	list.add(m0);
	m0.addCombination(0,4);
	Minterm m1= new Minterm(-1,1,1);
	list.add(m1);
	m1.addCombination(3,7);
	Minterm m2= new Minterm(1,0,-1);
	list.add(m2);
	m2.addCombination(4,5);
	Minterm m3= new Minterm(1,-1,1);
	list.add(m3);
	m3.addCombination(7,5);
	
	PrimeImplicantChart pmc = new PrimeImplicantChart(new int[]{0,3,4,5,7},list);
	List<Minterm> essential = pmc.extractEssentialPrimeImplicants();
	assertEquals(2,essential.size());
	assertTrue(essential.contains(m0));
	assertTrue(essential.contains(m1));
	// other
	List<Minterm> implicants = pmc.extractRemainingImplicants();
	
	assertEquals(1,implicants.size());
	assertTrue(list.containsAll(implicants));
	assertTrue(implicants.contains(m2)||implicants.contains(m3));
		2 equals 2?
		true
		true
		true
		1 equals 1?
		true
		true
		true
			2 equals 2?
			true
			true
			true
			1 equals 1?
			true
			true
			true
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	// extract NoEssential And Other Prime Implicants
	
	List<Minterm> list = new ArrayList<>();
	Minterm m0 = new Minterm(0,0,-1);
	list.add(m0);
	m0.addCombination(0,1);
	Minterm m1= new Minterm(0,-1,0);
	list.add(m1);
	m1.addCombination(0,2);
	Minterm m2= new Minterm(-1,0,1);
	list.add(m2);
	m2.addCombination(1,5);
	Minterm m3= new Minterm(-1,1,0);
	list.add(m3);
	m3.addCombination(2,6);
	Minterm m4= new Minterm(1,-1,1);
	list.add(m4);
	m4.addCombination(5,7);
	Minterm m5= new Minterm(1,1,-1);
	list.add(m5);
	m5.addCombination(6,7);
	
	PrimeImplicantChart pmc = new PrimeImplicantChart(new int[]{0,1,2,5,6,7},list);
	List<Minterm> essential = pmc.extractEssentialPrimeImplicants();
	assertEquals(0,essential.size());
	
	// Not other
	
	List<Minterm> implicants = pmc.extractRemainingImplicants();
	
	// Require Petrick's method
	assertTrue(list.containsAll(implicants));
		0 equals 0?
		true
		true
			0 equals 0?
			true
			true
				✓

Tous les tests ont été réussis ! ✓

▸ **Solution de l'auteur de la question (Java)**

Correct

Note pour cet envoi : 6,00/6,00.

Question 5

Correct

Note de 5,00 sur 5,00

Dans cette dernière étape, nous créons la classe **QMC.java**.

Vous pouvez bien évidemment utiliser toutes les classes précédentes. Attention, vous ne pouvez utiliser dans ces classes que les méthodes publiques qui vous étaient demandées.

1. Écrivez le constructeur. Son rôle est de créer les termes initiaux (Minterm) en fonction du nombre de bits nécessaires à coder le plus grand chiffre. Enregistrez bien également ces valeurs initiales.
   public QMC(int... values)
2. Définir la méthode qui calcule les implicants suffisants. Pour cela réduire la liste des minterms en fusionnant tant que c'est possible puis utiliser PrimeImplicantChart pour ne garder que les termes nécessiaires et suffisants.
   public List<Minterm> computePrimeImplicants()

Surtout, pensez à utiliser les classes précédentes et à définir d'autres méthodes dans AMC pour vous aider à implémenter cette deuxième méthode.

```
public class QMC {



    /**
     * Initialize the algorithm
     * @param values    decimals
     */
    public QMC(int... values) {

    }

   /**
     * Calculates and returns the necessary and sufficient minterms.
     */
    public List computePrimeImplicants(){
        return null;
    }




}
```

Par exemple:

Test	Résultat

`QMC qmc = new QMC(1,2,3,5);`
`List<Minterm> primaryTerms =  qmc.computePrimeImplicants();`
`        assertEquals(2,primaryTerms.size());`
`        assertTrue(primaryTerms.contains(new Minterm(-1,0,1)));`
`        assertTrue(primaryTerms.contains(new Minterm(0,1,-1)));`
	2 equals 2?
	true
	true
	true

Réponse : (régime de pénalités : 0 %)

Réinitialiser la réponse

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class QMC {


    private final int[] values;
    private final int bitCount;

    /**
     * Initialize the algorithm
     *
     * @param values decimals
     */
    public QMC(int... values) {
        this.values = values;
        this.bitCount = Minterm.numberOfBitsNeeded(Arrays.stream(values).max().orElse(0));
    }

    /**
     * Calculates and returns the necessary and sufficient minterms.
     */
    public List<Minterm> computePrimeImplicants() {
        var res = new ArrayList<Minterm>();

        // Create the initial minterms
        List<Minterm> minterms = new ArrayList<Minterm>();
        for (int i : values) {
            minterms.add(new Minterm(i, bitCount));
        }

        while (true) {
            var manager = new CategoryManager(minterms, bitCount);
            // Merge the categories
            minterms = manager.mergeCategories();
            if (manager.isLastTurn()) {
                break;
            }
        }

        var chart = new PrimeImplicantChart(values, minterms);
        res.addAll(chart.extractEssentialPrimeImplicants());
        res.addAll(chart.extractRemainingImplicants());
        return res;
    }
}

```

	Test	Résultat attendu	Résultat obtenu	

✓	QMC qmc = new QMC(1,2,3,5);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	assertEquals(2,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(-1,0,1)));
	assertTrue(primaryTerms.contains(new Minterm(0,1,-1)));
		2 equals 2?
		true
		true
		true
			2 equals 2?
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0,1,3,10,11,13,15);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	assertEquals(4,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(0,0,0,-1)));
	assertTrue(primaryTerms.contains(new Minterm(1,1,-1,1)));
	assertTrue(primaryTerms.contains(new Minterm(-1,0,1,1)));
	assertTrue(primaryTerms.contains(new Minterm(1,0,1,-1)));
		4 equals 4?
		true
		true
		true
		true
		true
			4 equals 4?
			true
			true
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0,1,3, 4, 5, 7);
	
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	assertEquals(2,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(-1,0,-1)));
	assertTrue(primaryTerms.contains(new Minterm(-1,-1,1)));
		2 equals 2?
		true
		true
		true
			2 equals 2?
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0,2,3, 4, 5, 6);
	
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	assertEquals(3,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(1,0,-1)));
	assertTrue(primaryTerms.contains(new Minterm(-1,-1,0)));
	assertTrue(primaryTerms.contains(new Minterm(0,1,-1)));
		3 equals 3?
		true
		true
		true
		true
			3 equals 3?
			true
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0, 1, 2, 4, 5, 6, 7);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	
	assertEquals(3,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(-1,0,-1)));
	assertTrue(primaryTerms.contains(new Minterm(-1,-1,0)));
	assertTrue(primaryTerms.contains(new Minterm(1,-1,-1)));
		3 equals 3?
		true
		true
		true
		true
			3 equals 3?
			true
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0, 1, 2, 4, 6, 8, 9, 12, 13, 14, 15);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	
	assertEquals(3,primaryTerms.size());
	assertTrue(primaryTerms.contains(new Minterm(-1,0,0,-1)));
	assertTrue(primaryTerms.contains(new Minterm(0,-1,-1,0)));
	assertTrue(primaryTerms.contains(new Minterm(1,1,-1,-1)));
		3 equals 3?
		true
		true
		true
		true
			3 equals 3?
			true
			true
			true
			true
				✓

	Test	Résultat attendu	Résultat obtenu	

✓	QMC qmc = new QMC(5,6,7);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	       Minterm m0 = new Minterm(1,-1,1);
	       Minterm m1= new Minterm(1,1,-1);
	
	       assertEquals(2,primaryTerms.size());
	       assertTrue(primaryTerms.contains(m0));
	       assertTrue(primaryTerms.contains(m1));
		2 equals 2?
		true
		true
		true
			2 equals 2?
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0,3,4,5,7);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	assertEquals(3,primaryTerms.size());
	Minterm m0 = new Minterm(-1,0,0);
	Minterm m1= new Minterm(-1,1,1);
	assertTrue(primaryTerms.contains(m0));
	assertTrue(primaryTerms.contains(m1));
	Minterm m2= new Minterm(1,0,-1);
	Minterm m3= new Minterm(1,-1,1);
	assertTrue(primaryTerms.contains(m2)||primaryTerms.contains(m3));
		3 equals 3?
		true
		true
		true
		true
			3 equals 3?
			true
			true
			true
			true
				✓

✓	QMC qmc = new QMC(0,1,2,5,6,7);
	List<Minterm> primaryTerms = qmc.computePrimeImplicants();
	// Require Petrick's method, the minimal number is 3
	 assertTrue(primaryTerms.size()>2);
		true	true	✓

Tous les tests ont été réussis ! ✓

▸ **Solution de l'auteur de la question (Java)**

Correct

Note pour cet envoi : 5,00/5,00.