```python
# hammingCode.py

from matrix import Matrix
from random import randint
from operator import xor

class HammingCodeBase :

    def __init__(self,k):
        if k < 2:
            raise Exception("The number of parity bits must at least be 2. k={}".format(k))
        self.k = k
        self.data_chunk_size = 2**k-k-1

    @staticmethod
    def getBinaryRepresentation(n,min_nb_of_bit=0):
        """ Renvoie une représentation d'un entier en binaire sur
            au moins min_nb_of_bit bits.
            Les bits de poids faible sont en premier"""

        bits = []
        i = n
        while i != 0:
            bits += [i % 2]
            i = i//2
        if len(bits) < min_nb_of_bit:
            bits += [0]*(min_nb_of_bit - len(bits))
        return bits

    @staticmethod
    def parityCheck(l):
        """ Renvoie 0 si le nombre de 1 dans la liste passée en argument est pair, 1 sinon

            Argument:
            -l (list): Un liste de données binaires"""
        parity = 0
        for elt in l:
            parity ^= elt
        return parity

    def cutDataInChunks(self,data):
        """Coupe la liste en morceaux de taille adaptée pour le code de
            Hamming avec le nombre de bits de parité voulus

            Argument:
            -data (list): Une liste de données binaires"""

        chunks, i = [], 0
        #print("k={},data_chunk_size={}".format(self.k,self.data_chunk_size))
        while i*(self.data_chunk_size) < len(data):
            chunk = data[i*(self.data_chunk_size):(i+1)*(self.data_chunk_size)]
            if len(chunk) < self.data_chunk_size:
                chunk += [0]*(self.data_chunk_size-len(chunk))
            chunks.append(chunk)
            i += 1
        return chunks

    def calculateGeneratorMatrix(self):
        """Calcule la matrice génératrice associée au code de Hamming voulu"""
        res = []
        for i in range(0,self.data_chunk_size):
            vect = [0]*self.data_chunk_size
            vect[i] = 1
            res.append(self.getEncryptedDataChunk(vect))
        self.generator_matrix = Matrix.fromArray(res).getTransposed()

    def calculateParityBitsValue(self,hamming_block):
        """Calcule le produit de la matrice de parité et du
            bloc de données pour obtenir les valeurs des bits de parité

            Argument:
            -hamming_block (list): Un liste de données binaires"""

        if len(hamming_block) != self.hamming_block_size:
            raise Exception("Wrong size : {} instead of {}".format(
                len(hamming_block),self.hamming_block_size))

```

```
076|         data_column = Matrix.toColumn(hamming_block)
077|         return Matrix.multiply(self.H,data_column)
078|
079|     def getEncryptedDataChunk(self,data_chunk):
080|         """Retourne le bloc de donnée encodé selon le code de Hamming
081|             avec le nombre de bits de parité voulus
082|
083|             Argument:
084|             -data_chunk (list): Un liste de données binaires contenant
085|                                  un bloc de Hamming non valide"""
086|         if len(data_chunk) != self.data_chunk_size:
087|             raise Exception("Wrong size : {} instead of {}".format(
088|                 len(data_chunk),self.data_chunk_size))
089|
090|         corr_data_chunk = self.insertParityBits(data_chunk)
091|         self.correctParityBits(corr_data_chunk)
092|         return corr_data_chunk
093|
094|     def getEncryptedDataChunkWithGeneratorMatrix(self,data_chunk):
095|         """Retourne le bloc de donnée encodé selon le code de Hamming
096|             avec la matrice génératrice avec le nombre de bits de parité voulus
097|
098|             Argument:
099|             -data_chunk (list): Un liste de données binaires contenant
100|                                  un bloc de Hamming non valide"""
101|         if len(data_chunk) != self.data_chunk_size:
102|             raise Exception("Wrong size : {} instead of {}".format(
103|                 len(data_chunk),self.data_chunk_size))
104|
105|         column = Matrix.toColumn(data_chunk)
106|         #On obtient une matrice colonne contenant le message encodé,
107|         #on la transpose pour extraire la liste contenant le message
108|         return Matrix.multiply(self.generator_matrix,column).getTransposed()[0]
109|
110|     def getDecryptedHammingBlock(self,hamming_block):
111|         block = hamming_block[:]
112|         self.ensureValidityAndCorrect(block)
113|         return self.removeParityBits(block)
114|
115|     def encodeData(self,data):
116|         chunks = self.cutDataInChunks(data)
117|         return [self.getEncryptedDataChunk(d) for d in chunks]
118|
119|     def encodeDataWithGeneratorMatrix(self,data):
120|         chunks = self.cutDataInChunks(data)
121|         return [self.getEncryptedDataChunkWithGeneratorMatrix(d) for d in chunks]
122|
123|     def decodeBlocks(self,blocks):
124|         return [self.getDecryptedHammingBlock(block) for block in blocks ]
125|
126| class HammingCode(HammingCodeBase):
127|
128|     empty_parity_bits = None
129|
130|     def __init__(self,k):
131|         """Regroupe différentes fonctions relatives au code de Hamming
132|
133|             Arguments:
134|             -k (int) : nombre de bits de parité"""
135|         HammingCodeBase.__init__(self,k)
136|         self.parity_bit_number = k
137|         self.hamming_block_size = self.data_chunk_size + self.parity_bit_number
138|
139|         self.empty_parity_bits = Matrix(self.parity_bit_number,1)
140|         self.calculateParityMatrix()
141|         self.calculateGeneratorMatrix()
142|
143|     def calculateParityMatrix(self):
144|         """ Renvoie une matrice de parité contenant les entiers de 1 à 2**k
145|             en représentation binaire.
146|             Les bits de poids faible sont en premier """
147|
148|         self.H = Matrix(self.hamming_block_size,self.parity_bit_number)
149|         for i in range(1,2**self.k):
150|             bin_repr = HammingCodeBase.getBinaryRepresentation(i,self.k)
151|             self.H.changeLine(i-1,bin_repr)
152|         self.H.transpose()
```

```
153|
154|
155|     def insertParityBits(self,data_chunk):
156|         """Retourne le bloc de données à encoder avec
157|             les bits de parité mis à 0
158|
159|             Argument:
160|             -data_chunk (list): Une liste de données binaires non initialisée"""
161|
162|         if len(data_chunk) != self.data_chunk_size:
163|             raise Exception("Wrong size : {} instead of
{}".format(len(data_chunk),self.data_chunk_size))
164|
165|         res = [0]
166|         pos = 0
167|         length = 0
168|         for i in range(0,self.k-1):
169|             length += 2**i
170|             res += [0] + data_chunk[pos:pos+length]
171|             pos += length
172|         return res
173|
174|     def removeParityBits(self,hamming_block):
175|         """Retourne le bloc de données auquel on enlevé les bits de parité
176|
177|             Argument:
178|             -hamming_block (list): Un liste de données binaires contenant
179|                                 un bloc de Hamming"""
180|         res = []
181|         for i in range(1,self.k):
182|             res += hamming_block[2**i:2**(i+1)-1]
183|         return res
184|
185|     def correctParityBits(self,hamming_block):
186|         """Retourne le bloc de données avec les bits
187|             de parité mis à la bonne valeur.
188|
189|             Argument:
190|             -hamming_block (list): Un liste de données binaires contenant
191|                                 un bloc de Hamming"""
192|
193|         if len(hamming_block) != self.hamming_block_size:
194|             raise Exception("Wrong size : {} instead of {}".format(
195|                 len(hamming_block),self.hamming_block_size))
196|
197|         #data_column = Matrix.toColumn(hamming_block)
198|         #parity_bits = Matrix.multiply(self.H,data_column)
199|         #for i in range(0,self.parity_bit_number):
200|         #    hamming_block[2**i-1] ^= parity_bits[i][0]
201|
202|         for i in range(0,self.parity_bit_number):
203|            parity_bit = Matrix.multiplyLineWithColumn(self.H[i],hamming_block)
204|            hamming_block[2**i-1] ^=  parity_bit
205|
206|     def ensureValidityAndCorrect(self,hamming_block):
207|         """Assure la validité du bloc et le corrige si besoin est
208|
209|         Argument:
210|         -hamming_block (list): Un liste de données binaires contenant
211|                             un bloc de Hamming à vérifier"""
212|         parity_bits = self.calculateParityBitsValue(hamming_block)
213|         if parity_bits == self.empty_parity_bits:
214|             return
215|         column = Matrix.toColumn(hamming_block)
216|         check_results = Matrix.multiply(self.H,column)
217|         e = self.H.searchColumn(check_results)
218|         hamming_block[e] =  1 - hamming_block[e]
219|
220| class ExtendedHammingCode(HammingCodeBase) :
221|
222|     empty_parity_bits = None
223|
224|     def __init__(self,k):
225|         """Regroupe différentes fonctions relatives au code de Hamming
226|
227|             Arguments:
228|             -k (int) : nombre de bits de parité
```

```
229|             -extended (bool): utilisation du code de Hamming étendu, ce dernier
230|                             ajoute un bit de parité final permettant de
231|                             détecter jusqu'à 2 erreurs"""
232|         HammingCodeBase.__init__(self,k)
233|         self.parity_bit_number = k + 1
234|         self.hamming_block_size = self.data_chunk_size + self.parity_bit_number
235|
236|         self.empty_parity_bits = Matrix(self.parity_bit_number,1)
237|         self.calculateParityMatrix()
238|         self.calculateGeneratorMatrix()
239|
240|     def calculateParityMatrix(self):
241|         """ Renvoie une matrice de parité contenant les entiers de 1 à 2**k
242|             en représentation binaire.
243|             Les bits de poids faible sont en premier """
244|
245|         self.H = Matrix(self.hamming_block_size,self.parity_bit_number)
246|         for i in range(1,2**self.k):
247|             bin_repr = [1] + HammingCodeBase.getBinaryRepresentation(i,self.k)
248|             self.H.changeLine(i,bin_repr)
249|             first_line =  [1] + [0]*self.k
250|             self.H.changeLine(0,first_line)
251|         self.H.transpose()
252|
253|     def insertParityBits(self,data_chunk):
254|         """Retourne le bloc de données à encoder avec
255|             les bits de parité mis à 0
256|
257|             Argument:
258|             -data_chunk (list): Une liste de données binaires non initialisée"""
259|
260|         if len(data_chunk) != self.data_chunk_size:
261|             raise Exception("Wrong size : {} instead of
{}".format(len(data_chunk),self.data_chunk_size))
262|
263|         res = [0]
264|         pos = 0
265|         length = 0
266|         for i in range(0,self.k-1):
267|             length += 2**i
268|             res += [0] + data_chunk[pos:pos+length]
269|             pos += length
270|         res = [0] + res
271|         return res
272|
273|     def removeParityBits(self,hamming_block):
274|         res = []
275|         for i in range(1,self.k):
276|             res += hamming_block[2**i+1:2**(i+1)]
277|         return res
278|
279|     def correctParityBits(self,hamming_block):
280|         """Retourne le bloc de données avec les bits
281|             de parité mis à la bonne valeur.
282|
283|             Argument:
284|             -hamming_block (list): Un liste de données binaires contenant
285|                                 un bloc de Hamming"""
286|
287|         if len(hamming_block) != self.hamming_block_size:
288|             raise Exception("Wrong size : {} instead of {}".format(
289|                 len(hamming_block),self.hamming_block_size))
290|
291|         for i in range(1,self.parity_bit_number):
292|             parity_bit = Matrix.multiplyLineWithColumn(self.H[i],hamming_block)
293|             hamming_block[2**(i-1)] ^=  parity_bit
294|
295|         #Le cas i = 0 est différent car il repose sur les précédentes modifications
296|         #Il doit donc être traité à part
297|         parity_bit = Matrix.multiplyLineWithColumn(self.H[0],hamming_block)
298|         hamming_block[0] ^= parity_bit
299|
300|     def ensureValidityAndCorrect(self,hamming_block):
301|         """Assure la validité du bloc et le corrige si besoin est
302|
303|         Argument:
304|         -hamming_block (list): Un liste de données binaires contenant
```

```
                            un bloc de Hamming à vérifier"""
        parity_bits = self.calculateParityBitsValue(hamming_block)
        if parity_bits == self.empty_parity_bits:
            return
        column = Matrix.toColumn(hamming_block)
        check_results = Matrix.multiply(self.H,column)
        if check_results[0][0] == 0:
            raise Exception("Il y a au moins deux erreurs. Bloc impossible à corriger")
        e = self.H.searchColumn(check_results)
        hamming_block[e] =  1 - hamming_block[e]


class HammingCodeXOR(HammingCodeBase):

    def __init__(self,k):
        """Regroupe différentes fonctions relatives au code de Hamming

            Arguments:
            -k (int) : nombre de bits de parité
            -extended (bool): utilisation du code de Hamming étendu, ce dernier
                            ajoute un bit de parité final permettant de
                            détecter jusqu'à 2 erreurs"""
        HammingCodeBase.__init__(self,k)
        self.parity_bit_number = k + 1
        self.hamming_block_size = self.data_chunk_size + self.parity_bit_number

    def insertParityBits(self,data_chunk):
        """Retourne le bloc de données à encoder avec
            les bits de parité mis à 0

            Argument:
            -data_chunk (list): Une liste de données binaires non initialisée"""

        if len(data_chunk) != self.data_chunk_size:
            raise Exception("Wrong size : {} instead of
{}".format(len(data_chunk),self.data_chunk_size))

        res = [0]
        pos = 0
        length = 0
        for i in range(0,self.k-1):
            length += 2**i
            res += [0] + data_chunk[pos:pos+length]
            pos += length
        res = [0] + res
        return res

    def removeParityBits(self,hamming_block):
        res = []
        for i in range(1,self.k):
            res += hamming_block[2**i+1:2**(i+1)]
        return res


    def correctParityBits(self,hamming_block):
        pos = self.binaryListXOR(hamming_block)
        binary = HammingCodeBase.getBinaryRepresentation(pos)
        for i, bit in enumerate(binary):
            hamming_block[2**i] = (hamming_block[2**i] +bit) % 2

        hamming_block[0] = HammingCodeBase.parityCheck(hamming_block)

    def ensureValidityAndCorrect(self,hamming_block):
        """Assure la validité du bloc et le corrige si besoin est

        Argument:
        -hamming_block (list): Un liste de données binaires contenant
                            un bloc de Hamming à vérifier"""
        pos = self.binaryListXOR(hamming_block)

        if pos != 0 and HammingCodeBase.parityCheck(hamming_block) == 0:
            raise Exception("Il y a au moins deux erreurs. Bloc impossible à corriger")

        hamming_block[pos] = 1 - hamming_block[pos]

    def binaryListXOR(self,l):
        sum = 0
```

```
381|         for i, bit in enumerate(l):
382|             if bit == 1:
383|                 sum ^= i
384|         return sum
385|
386|     def getEncryptedDataChunk(self,data_chunk):
387|         """Retourne le bloc de donnée encodé selon le code de Hamming
388|             avec le nombre de bits de parité voulus
389|
390|             Argument:
391|             -data_chunk (list): Un liste de données binaires contenant
392|                                  un bloc de Hamming non valide"""
393|         if len(data_chunk) != self.data_chunk_size:
394|             raise Exception("Wrong size : {} instead of {}".format(
395|                 len(data_chunk),self.data_chunk_size))
396|
397|         corr_data_chunk = self.insertParityBits(data_chunk)
398|         self.correctParityBits(corr_data_chunk)
399|         return corr_data_chunk
400|
401|     def encodeData(self,data):
402|         chunks = self.cutDataInChunks(data)
403|         return [self.getEncryptedDataChunk(d) for d in chunks]
404|
405|     def getDecryptedHammingBlock(self,hamming_block):
406|         block = hamming_block[:]
407|         self.ensureValidityAndCorrect(block)
408|         return self.removeParityBits(block)
409|
410|     def decodeBlocks(self,blocks):
411|         return [self.getDecryptedHammingBlock(block) for block in blocks ]
412|
413| class HammingCodeTests :
414|
415|     h = None
416|     extended_h = None
417|     h_xor = None
418|
419|     def __init__(self):
420|         self.h = HammingCode(4)
421|         self.extended_h = ExtendedHammingCode(4)
422|         self.h_xor = HammingCodeXOR(4)
423|
424|     def generateRandomBits(self,n):
425|         return [ randint(0,1) for i in range(0,n)]
426|
427|     def expectEqual(self,a,b):
428|         if a != b :
429|             raise Exception("Test Failed: a={}, b={}".format(a,b))
430|
431|     def expectTrue(self,b):
432|         if not b :
433|             raise Exception("Test Failed")
434|
435|     def expectFalse(self,b):
436|         if b :
437|             raise Exception("Test Failed")
438|
439|     def testAll(self):
440|         self.testCutDataInChunks()
441|         self.testParityMatrix()
442|         self.testGeneratorMatrix()
443|         self.testInsertParityBits()
444|         self.testRemoveParityBits()
445|         self.testCorrectParityBits()
446|         self.testGetEncryptedDataChunk()
447|         self.testGetEncryptedDataChunkWithGeneratorMatrix()
448|         self.testGetDecryptedHammingBlock()
449|         self.testReversabilityEncryption()
450|         self.testSimilarityEncryption()
451|         self.testSimilarityEncryptionData()
452|
453|     def testParityMatrix(self):
454|         comp = Matrix.fromArray(
455|                 [[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
456|                 [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
457|                 [0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1],
```

```
458|                 [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]])
459|         self.expectEqual(self.h.H, comp)
460|
461|         ext_comp = Matrix.fromArray(
462|                 [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
463|                 [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
464|                 [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
465|                 [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1],
466|                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1], ])
467|
468|         self.expectEqual(self.extended_h.H, ext_comp)
469|
470|     def testGeneratorMatrix(self):
471|         m = Matrix.multiply(self.h.H,self.h.generator_matrix)
472|         comp = Matrix(self.h.H.getHeight(),self.h.generator_matrix.getWidth())
473|         self.expectEqual(comp, m)
474|
475|         ext_m = Matrix.multiply(self.extended_h.H,self.extended_h.generator_matrix)
476|         ext_comp =
Matrix(self.extended_h.H.getHeight(),self.extended_h.generator_matrix.getWidth())
477|         self.expectEqual(ext_comp, ext_m)
478|
479|     def testCutDataInChunks(self):
480|         comp = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
481|                 [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21],
482|                 [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32],
483|                 [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43],
484|                 [44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54],
485|                 [55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65],
486|                 [66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76],
487|                 [77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87],
488|                 [88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98],
489|                 [99,0,0,0,0,0,0,0,0,0,0]]
490|         data = [i for i in range(100)]
491|         data_chunks = self.h.cutDataInChunks(data)
492|         self.expectEqual(comp, data_chunks)
493|
494|     def testInsertParityBits(self):
495|         comp = [0, 0, 1, 0, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11]
496|         d = [1,2,3,4,5,6,7,8,9,10,11]
497|         self.expectEqual(self.h.insertParityBits(d),comp)
498|
499|         ext_comp = comp = [0, 0, 0, 1, 0, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11]
500|         self.expectEqual(self.extended_h.insertParityBits(d),ext_comp)
501|
502|     def testRemoveParityBits(self):
503|         comp = [1,2,3,4,5,6,7,8,9,10,11]
504|         d = [0, 0, 1, 0, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11]
505|         self.expectEqual(self.h.removeParityBits(d), comp)
506|
507|         ext_d = [0, 0, 0, 1, 0, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11]
508|         self.expectEqual(self.extended_h.removeParityBits(ext_d), comp)
509|
510|     def testCorrectParityBits(self):
511|         comp = [1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1]
512|         b =    [0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1]
513|         b2 =    [0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1]
514|         self.h.correctParityBits(b)
515|         self.expectEqual(b, comp)
516|
517|         ext_comp = [1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1]
518|         ext_b = [0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1]
519|         self.extended_h.correctParityBits(ext_b)
520|         self.expectEqual(ext_b, ext_comp)
521|
522|     def testGetEncryptedDataChunk(self):
523|         d = [1,1,1,1,1,1,1,1,1,1,1]
524|         comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
525|         self.expectEqual(self.h.getEncryptedDataChunk([1,1,1,1,1,1,1,1,1,1,1]), comp)
526|
527|         ext_comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
528|         self.expectEqual(self.extended_h.getEncryptedDataChunk([1,1,1,1,1,1,1,1,1,1,1]),
ext_comp)
529|
530|     def testGetEncryptedDataChunk2(self):
531|         d = [1,1,1,1,1,1,1,1,1,1,1]
532|         comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

```
533|         self.expectEqual(self.h.getEncryptedDataChunk(d), comp)
534|
535|         ext_comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
536|         self.expectEqual(self.extended_h.getEncryptedDataChunk2(d), ext_comp)
537|
538|     def testGetEncryptedDataChunkWithGeneratorMatrix(self):
539|         d = [1,1,1,1,1,1,1,1,1,1]
540|         comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
541|         self.expectEqual(self.h.getEncryptedDataChunk(d), comp)
542|
543|         ext_comp = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
544|         self.expectEqual(self.extended_h.getEncryptedDataChunkWithGeneratorMatrix(d),
ext_comp)
545|
546|     def testGetDecryptedHammingBlock(self):
547|         comp = [1,1,1,1,1,1,1,1,1,1,1]
548|         d = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
549|         self.expectEqual(self.h.getDecryptedHammingBlock(d),comp)
550|
551|     def testReversabilityEncryption(self):
552|         data = [randint(0,1) for i in range (0,90)]
553|         chunks = self.h.cutDataInChunks(data)
554|         blocks = self.h.encodeData(data)
555|         self.expectEqual(chunks, self.h.decodeBlocks(blocks))
556|
557|         ext_blocks = self.extended_h.encodeData(data)
558|         self.expectEqual(chunks,self.extended_h.decodeBlocks(ext_blocks))
559|
560|         ext_blocks_xor = self.h_xor.encodeData(data)
561|         self.expectEqual(chunks,self.h_xor.decodeBlocks(ext_blocks_xor))
562|
563|     def testSimilarityEncryption(self):
564|         for i in range(0,500):
565|             data = self.generateRandomBits(self.h.data_chunk_size)
566|             block = self.h.getEncryptedDataChunk(data)
567|             block_gen = self.h.getEncryptedDataChunkWithGeneratorMatrix(data)
568|             self.expectEqual(block,block_gen)
569|
570|             ext_block = self.extended_h.getEncryptedDataChunk(data)
571|             ext_block_gen = self.extended_h.getEncryptedDataChunkWithGeneratorMatrix(data)
572|             ext_block_xor = self.h_xor.getEncryptedDataChunk(data)
573|             self.expectEqual(ext_block,ext_block_gen)
574|             self.expectEqual(ext_block,ext_block_xor)
575|
576|     def testSimilarityEncryptionData(self):
577|         for i in range(0,1500):
578|             data = self.generateRandomBits(self.h.data_chunk_size*10)
579|             blocks = self.h.encodeData(data)
580|             blocks_gen = self.h.encodeDataWithGeneratorMatrix(data)
581|             self.expectEqual(blocks,blocks_gen)
582|
583|             ext_blocks = self.extended_h.encodeData(data)
584|             ext_blocks_gen = self.extended_h.encodeDataWithGeneratorMatrix(data)
585|             self.expectEqual(ext_blocks,ext_blocks_gen)
586|
587| HammingCodeTests().testAll()
```