Ing2 GSI. Groupe 2

Rapport - Projet Méthodes de compression sans perte

réalisé le 16 mars 2024

Sujet: Compléter différents algorithmes de compressions en Haskell

Réalisé par:

MACHNIK Adrien

CASTELAO Romain

DJEARAM Logesh

PAQUES Anatole

HUNAULT Clément

Année d'étude:

2023-2024

 Repository-github

SOMMAIRE

1. Méthode RLE

```haskell
-- | RLE compress method
compress :: Eq a => [a] -> [(a, Int)]
compress [] = []
compress (x:xs) = let (packed, rest) = span (== x) xs
                      count = 1 + length packed
                  in (x, count) : compress rest

-- | RLE uncompress method
-- If input cannot be uncompressed, returns `Nothing`
uncompress :: [(a, Int)] -> Maybe [a]
uncompress [] = Just []
uncompress ((x, count):xs)
 | count <= 0 = Nothing
 | otherwise = case uncompress xs of
                 Nothing -> Nothing
                 Just rest -> Just (replicate count x ++ rest)
```

Description :

La fonction compress parcourt la liste en regroupant les éléments identiques consécutifs, elle compte le nombre d'occurrences de chaque groupe et elle retourne une liste de paires contenant chaque élément et son nombre d'occurrences.

La fonction uncompress parcourt la liste de paires et reconstitue la liste originale en répétant chaque élément le nombre de fois spécifié dans la paire. Si la reconstruction échoue, renvoie `Nothing`, sinon renvoie la liste reconstruite.

Configuration optimale :

Plus il y a de caractères qui se répètent à la suite dans le texte de base à compresser, plus le taux de compression sera meilleur.

2. Méthode LZ

a. LZ78

Description:

Consiste à “parcourir la source en apprenant les suites de symboles les plus fréquentes. Ces suites de symboles, appelées préfixes sont stockées dans un dictionnaire, construit au fur et à mesure que l'on parcourt la source”.

Fonction de compression et de décompression :

- `compress`: prend une chaîne de caractères en entrée et la transforme en une série de paires (Int, Char). Chaque paire représente un indice dans un dictionnaire dynamique et un caractère. La fonction construit ce dictionnaire en parcourant la chaîne d'entrée, cherchant à chaque fois la plus longue sous-chaîne déjà présente dans le dictionnaire. Lorsqu'une telle sous-chaîne est trouvée, elle est étendue avec le caractère suivant, et cette nouvelle chaîne est ajoutée au dictionnaire. La paire constituée de l'indice de la sous-chaîne trouvée et du caractère suivant est alors ajoutée à la sortie. Si aucune sous-chaîne correspondante n'est trouvée, le caractère est considéré comme une nouvelle entrée avec un indice de 0.

- `uncompress` : prend la série de paires (Int, Char) générée par la compression et reconstruit la chaîne d'origine. Elle utilise un dictionnaire similaire pour retrouver les sous-chaînes référencées par les indices dans les paires. Pour chaque paire, si l'indice est 0, cela signifie que le caractère est une entrée unique et est directement ajouté à la chaîne de sortie. Si l'indice est différent de 0, la sous-chaîne correspondante est retrouvée dans le dictionnaire, étendue avec le caractère de la paire, ajoutée au dictionnaire pour les références futures, et également ajoutée à la chaîne de sortie. Si un indice invalide est rencontré, indiquant une paire faisant référence à une sous-chaîne non encore ajoutée au dictionnaire, la fonction retourne Nothing, signalant une erreur dans les données compressées.

- `compressAux` : est une fonction auxiliaire qui est présente dans le processus de compression LZ78. Elle traite la chaîne d'entrée caractère par caractère, en cherchant la plus longue sous-chaîne qui matche dans le dictionnaire courant.

Pour chaque caractère, elle trouve cette sous-chaîne la plus longue grâce à longestMatch et détermine s'il faut ajouter une nouvelle entrée au dictionnaire ou utiliser une entrée existante. Lorsqu'une sous-chaîne est trouvée dans le dictionnaire, compressAux enregistre l'indice de cette sous-chaîne dans le dictionnaire ainsi que le caractère suivant dans la chaîne de sortie. Si la sous-chaîne n'est pas trouvée, le caractère actuel est ajouté comme une nouvelle entrée dans le dictionnaire avec un indice de 0 dans la sortie. Cette approche répétitive permet de construire progressivement la liste de sortie compressée en ajoutant soit des références aux sous-chaînes déjà vues, soit de nouveaux caractères uniques.

- `longestMatch` : permet d'identifier la plus longue sous-chaîne existante dans le dictionnaire qui correspond au début de la chaîne restante à compresser. En commençant par le premier caractère, elle étend progressivement cette sous-chaîne candidate en ajoutant des caractères jusqu'à ce qu'aucune correspondance plus longue ne soit trouvée dans le dictionnaire. Cette fonction permet d'optimiser la compression en identifiant et en utilisant les plus longues séquences répétitives, réduisant ainsi la taille globale de la sortie compressée.

- `uncompressAux` : prend la liste compressée de paires (Int, Char) et reconstruit la chaîne originale. En examinant chaque paire, elle utilise l'indice pour retrouver la sous-chaîne correspondante dans le dictionnaire de décompression. Si l'indice est 0, cela indique un caractère unique qui est ajouté directement à la chaîne de sortie. Pour les autres indices, elle récupère la sous-chaîne, y ajoute le caractère de la paire, et ajoute cette nouvelle chaîne à la fois au dictionnaire et à la chaîne de sortie. Cette méthode assure que même les sous-chaînes complexes peuvent être reconstruites à partir de références simples, permettant ainsi de restaurer la chaîne originale à partir de sa version compressée.

b. LZW

Module contenant les méthodes liées à la compression LZW.

Description :

Méthode de compression qui utilise un dictionnaire pour remplacer les séquences récurrentes de données par des codes. Au fur et à mesure de la lecture des données, de nouvelles séquences sont ajoutées au dictionnaire, permettant une compression plus efficace à mesure que le processus avance.

Fonction de compression et de décompression :

- `compress` : utilise une fonction locale `compress'` pour effectuer la compression en parcourant la chaîne d'entrée, recherchant les préfixes les plus longs dans le dictionnaire et substituant ces préfixes par leurs indices dans la séquence compressée. Cela correspond à la partie du code où `compress'` est défini et utilisé pour itérer à travers la chaîne d'entrée, en trouvant les préfixes existants dans le dictionnaire et en ajoutant de nouveaux préfixes si nécessaire.
- `uncompress` : Utilise une fonction locale `uncompress'` pour décompresser la séquence compressée en utilisant le dictionnaire des motifs rencontrés précédemment. Cette fonction reconstruit la chaîne d'origine à partir des indices. Les indices sont décodés et utilisés pour retrouver les entrées correspondantes dans le dictionnaire, ce qui permet de reconstituer la chaîne d'origine.

3. Méthode Statistiques

a. EncodingTree

Module contenant les méthodes permettant d'encoder, de décoder, de compresser et de gérer tout ce qui a rapport avec des arbres d'encodage binaire.

Fonction utilitaire :

- `isLeaf` : est une feuille
- `count` : compte la fréquence d'un noeud/feuille
- `has` : si le noeud/feuille a ce symbole
- `meanLength` : moyenne de la longueur d'un arbre d'encodage

Fonctions d'encodage et de décodage :

- `encode` : encode une liste de caractères en liste de Bits
- `decodeOnce` : décode le premier caractère
- `decode` : décode une liste de Bits d'un arbre d'encodage

Fonction de compression et de décompression :

- `compress` : compression d'une liste de symboles en fonction de l'algorithme
- `uncompress` : décompression d'une liste de Bits en fonction d'un arbre d'encodage

b. Source

Module qui fournit des fonctions utilitaires pour les sources d'entrée.

Fonctions utilitaires :

- `occurences` : prend une liste de symboles et renvoies leurs occurrences
- `entropy` : calcul l'entropie d'une liste de symboles
- `ordererCounts` : renvoie une liste d'occurrences triée

c. Huffman

Module contenant les méthodes liées à la compression selon la méthode d'Huffman.

Méthode Huffman :
Consiste à créer un arbre binaire en fonction du nombre d'occurrences d'un symbole.
Plus l'occurrence est petite, plus le nœud associé au symbole sera profond.

Le but étant de réaliser un arbre d'encodage ou chaque feuilles sera associée à un symbole et un poid, et chaque nœud sera le résultat de la fusion des deux précédents nœuds/feuilles.

Pour se déplacer dans l'arbre, on associe `0` pour aller à gauche et `1` pour aller à droite.

Configuration optimale :
La configuration optimale est obtenue en parcourant l'ensemble des symboles et en construisant l'arbre de Huffman de manière à minimiser la longueur moyenne des codes.
Donc chaque source de données aura sa propre configuration optimale.

Fonctions utilitaires :

- `makeLeaves` : prend une liste de symboles et retourne une liste de feuilles encoder
- `huffmanTree` : construit l'arbre de Huffman a partir d'une liste d'arbre d'encodage
- `mergeTrees` : définit et gère comment deux arbres sont fusionnés
- `insertOrdered` : insère un arbre dans une liste d'arbre en les triant par fréquences
- `treeWeight` : extrait la fréquence/poids d'un arbre (utilisé pour le tri)

Fonctions d'encodage :

- `tree` : prend une liste de symboles et renvoie l'arbre d'encodage construit à l'aide de la méthode de Huffman

d. Shannon-fano

Méthode Shannon-fano :

Consiste à construire l'arbre binaire depuis la racine en partant de la liste de toutes les occurrences triées par ordre décroissant d'occurrence.

Puis on sépare la liste en deux de façon à ce que le nombre d'occurrences des deux parties soient le plus égales possible qui correspondent aux fils de la racine.
On recommence l'opération jusqu'à avoir des listes à un élément qui correspondent aux feuilles.

Pour se déplacer dans l'arbre, on associe `0` pour aller à gauche et `1` pour aller à droite.

Configuration optimale :

Les symboles les plus fréquents devraient recevoir des codes plus courts que les symboles moins fréquents.
Mais ce n'est pas toujours réalisé en raison de la nature de l'algorithme qui divise l'ensemble des symboles en groupes presque égaux en probabilités.
Une configuration optimale serait donc dans le cas où les symboles sont en nombres égaux.

Fonctions utilitaires :

- `closestTo` : prends 3 nombres et renvoie lequel des deux premiers est le plus proche du troisième
- `split` : prend une liste d'occurrence et permet de la séparer en deux listes d'occurrences égales ou presque.

Fonctions d'encodage :

- `tree` : prend une liste de symboles et renvoie l'arbre d'encodage construit à l'aide de la méthode de Shannon (elle fonction auxiliaire qui prend une liste d'occurrences en entrée et qui renvoie un arbre en faisant appelle à elle-même sur les deux sous-listes obtenues jusqu'à atteindre une sous-liste à un élément qui sera une feuille.)

4. Comparaisons

a. Comparaisons Statiques

chaine de char* donnée à titre exemplaire

```
exampleSource :: String
exampleSource = "bonjour je suis un exemple de source pour tester
l'algorithme de compression"
```

On va se servir d'une fonction créer arbitrairement pour calculer la taille en base 10 :

```
-- list of bits to base 10
bitsToBase10 :: [Bit] -> Int
bitsToBase10 [] = 0
bitsToBase10 (bit:bits) = fromEnum bit * 2^(length bits) + bitsToBase10 bits
```

On va aussi se servir de meanLength :

```
-- | Mean length of the binary encoding
meanLength :: EncodingTree a -> Double
meanLength tree = meanLength' tree 0 / fromIntegral (count tree)
  where
    -- Auxilary function to calculate mean length
    meanLength' :: EncodingTree a -> Int -> Double
    meanLength' (EncodingLeaf size _) depth = fromIntegral size *
fromIntegral depth
    meanLength' (EncodingNode _ left right) depth =
        meanLength' left (depth + 1) + meanLength' right (depth + 1)
```

Voici le test que l'on va réaliser sur tous les algorithmes statiques :

```
-- Générez l'arbre de compression à partir de la source
    let algoTree = tree exampleSource

    let compressTree = compress tree exampleSource

    let decompressTree = uncompress compressTree

    let sourceSize = length exampleSource  -- Taille de la chaîne d'origine

    case compressTree of
        (Just encodingTree, encodedSymbols) -> do
            let encodedBase10 = bitsToBase10 encodedSymbols  -- Conversion
des bits en base 10
            let compressedSizeBits = length encodedSymbols  -- Taille de la
chaîne compressée en bits
            let compressedSizeOctets = ceiling ( fromIntegral
compressedSizeBits  /8)
            let totlength = totalLength encodingTree
            let test = meanLength encodingTree
            putStrLn $ "Arbre de huffman : " ++ show encodingTree ++ "\n" ++
                        "Symboles d'origine : " ++ show encodedSymbols ++
"\n" ++
                       "Taille encodés base 10 : " ++ show encodedBase10 ++
"\n" ++
                       "Taille de la chaîne d'origine : " ++ show sourceSize
++ " caractères" ++ "\n" ++
                       "Taille de la chaîne compressée : " ++ show
compressedSizeBits ++ " bits" ++ "\n" ++
                       "Taille de la chaîne compressée : " ++ show
compressedSizeOctets ++ " octets" ++ "\n" ++
                       "meanLength : " ++ show test ++ "\n" ++
                       "totalLength : " ++ show totlength
        (Nothing, _) -> putStrLn "Impossible de créer l'arbre de
compression"

    case decompressTree of
        Just decodedSymbols -> putStrLn $ "Symboles décodés : " ++ show
decodedSymbols
        Nothing -> putStrLn "Impossible de décoder les symboles"
```

Huffman résultat :

```
Arbre de huffman : EncodingNode 76 (EncodingNode 30 (EncodingNode 13
(EncodingLeaf 6 's') (EncodingLeaf 7 'o')) (EncodingNode 17 (EncodingNode 8
(EncodingNode 4 (EncodingLeaf 2 'd') (EncodingLeaf 2 'j')) (EncodingNode 4
(EncodingNode 2 (EncodingLeaf 1 '\'') (EncodingLeaf 1 'a')) (EncodingLeaf 2
'c'))) (EncodingNode 9 (EncodingNode 4 (EncodingNode 2 (EncodingLeaf 1 'h')
(EncodingLeaf 1 'x')) (EncodingNode 2 (EncodingLeaf 1 'b') (EncodingLeaf 1
'g'))) (EncodingLeaf 5 'u')))) (EncodingNode 46 (EncodingNode 22
(EncodingLeaf 11 ' ') (EncodingLeaf 11 'e')) (EncodingNode 24 (EncodingNode
12 (EncodingNode 6 (EncodingLeaf 3 'i') (EncodingLeaf 3 'l')) (EncodingLeaf
6 'r')) (EncodingNode 12 (EncodingNode 6 (EncodingLeaf 3 'p') (EncodingLeaf
3 't')) (EncodingNode 6 (EncodingLeaf 3 'm') (EncodingLeaf 3 'n')))))
Symboles d'origine :
[0,1,1,0,1,0,0,0,1,1,1,1,1,1,0,1,0,0,1,0,0,1,0,1,1,1,1,1,0,1,1,0,0,0,1,0,0,1
,1,0,1,1,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1,1,1,1,0,0,1,0
,1,0,1,1,0,0,1,1,0,1,1,1,1,1,0,1,1,1,0,0,1,1,0,0,1,1,0,1,1,0,0,0,1,0,0,0,1,0
,1,1,0,0,0,0,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,1,1,0,1,1,0,0,1,1,1,0,0,0,0,1,0
,1,1,1,1,1,0,1,1,0,0,1,1,1,0,1,1,0,1,0,0,0,1,1,1,0,1,1,0,1,1,1,0,1,1,0,0,1,1
,0,0,1,0,1,0,1,0,0,0,1,0,1,0,1,1,1,0,0,1,0,1,1,0,1,1,0,0,1,1,1,0,1,1,1,0,0,0
,1,1,1,0,1,0,1,1,0,0,0,1,1,1,1,0,1,0,1,1,0,0,0,1,0,0,0,1,0,1,1,0,0,0,1,0,1,1
,0,0,1,1,1,1,1,0,1,1,1,0,0,1,1,0,1,1,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,1,1,1
,1]
Taille encodés base 10 : -3016631511758137281
Taille de la chaîne d'origine : 76 caractères
Taille de la chaîne compressée : 305 bits
Taille de la chaîne compressée : 39 octets
meanLength : 4.0131578947368425
totalLength : 76
Symboles décodés : "bonjour je suis un exemple de source pour tester
l'algorithme de compression"
```

Shannon-fano résultat :

```
Arbre de shannon-fano : EncodingNode 76 (EncodingNode 35 (EncodingNode 22
(EncodingLeaf 11 'e') (EncodingLeaf 11 ' ')) (EncodingNode 13 (EncodingLeaf
7 'o') (EncodingLeaf 6 's'))) (EncodingNode 41 (EncodingNode 20
(EncodingNode 11 (EncodingLeaf 6 'r') (EncodingLeaf 5 'u')) (EncodingNode 9
(EncodingLeaf 3 't') (EncodingNode 6 (EncodingLeaf 3 'p') (EncodingLeaf 3
'n')))) (EncodingNode 21 (EncodingNode 11 (EncodingNode 6 (EncodingLeaf 3
'm') (EncodingLeaf 3 'l')) (EncodingNode 5 (EncodingLeaf 3 'i')
(EncodingLeaf 2 'j'))) (EncodingNode 10 (EncodingNode 5 (EncodingLeaf 2 'd')
(EncodingNode 3 (EncodingLeaf 2 'c') (EncodingLeaf 1 'x'))) (EncodingNode 5
(EncodingNode 2 (EncodingLeaf 1 'h') (EncodingLeaf 1 'g')) (EncodingNode 3
(EncodingLeaf 1 'b') (EncodingNode 2 (EncodingLeaf 1 'a') (EncodingLeaf 1
'\'')))))))
Symboles d'origine :
[1,1,1,1,1,0,0,1,0,1,0,1,1,1,1,1,0,1,1,0,1,0,1,0,0,1,1,0,0,0,0,0,1,1,1,0,1,1
,0,0,0,0,0,1,0,1,1,1,0,0,1,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,1,1,1,0,0,1,0,0
,0,1,1,1,0,1,1,0,0,0,1,1,0,0,0,1,0,1,1,0,1,1,0,0,1,0,0,0,0,0,1,1,1,1,0,0,0,0
,0,0,0,1,0,1,1,0,1,0,1,0,0,1,1,0,0,0,1,1,1,0,1,0,0,0,0,0,0,1,1,0,1,1,0,0,1,0
,1,0,0,1,1,0,0,0,0,0,1,1,0,1,0,0,0,0,0,1,1,1,0,1,0,0,0,0,1,0,0,0,0,0,1,1,1,0
,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,0,1,1,1,1,1,0,1,0,1,0,1,0,0,0,1,1,0,1
,0,1,0,1,0,1,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,0,1
,0,0,1,0,1,1,0,0,0,1,0,1,1,0,1,0,0,0,0,0,0,0,1,1,0,1,1,1,1,0,1,0,0,1,0,1,0,1
,1,1]
Taille encodés base 10 : 135175249519147607
Taille de la chaîne d'origine : 76 caractères
Taille de la chaîne compressée : 306 bits
Taille de la chaîne compressée : 39 octets
meanLength : 4.026315789473684
totalLength : 76
Symboles décodés : "bonjour je suis un exemple de source pour tester
l'algorithme de compression"
```

Entropie :

```
Entropie de la source : 3.9726973087002264
```

On observe une différence de l'ordre de `10^ -2`.

L'algorithme de Huffman obtient une moyenne de `4.01` et Shannon-fano de `4.02`.

On peut en conclure que sur une chaîne de taille moyenne, l'algorithme de Huffman est légèrement meilleur sur de petites chaînes.

Essayons sur une chaîne plus grande.

```
exampleSource :: String
exampleSource = "sagittis eu volutpat odio facilisis mauris sit amet massa
vitae tortor condimentum lacinia quis vel eros donec ac odio tempor orci
dapibus ultrices in iaculis nunc sed augue lacus viverra vitae congue eu
consequat ac felis donec et odio pellentesque diam volutpat commodo sed
egestas egestas fringilla phasellus faucibus scelerisque eleifend donec
pretium vulputate sapien nec sagittis aliquam malesuada bibendum arcu vitae
elementum curabitur vitae nunc sed velit dignissim sodales ut eu sem integer
vitae justo eget magna fermentum iaculis eu non diam phasellus vestibulum
lorem sed risus ultricies tristique nulla aliquet enim tortor at auctor urna
nunc id cursus"
```

Huffman résultat :

```
Taille de la chaîne d'origine : 666 caractères
Taille de la chaîne compressée : 2691 bits
Taille de la chaîne compressée : 337 octets
meanLength : 4.04054054054054
totalLength : 666
```

Shannon-fano résultat :

```
Taille de la chaîne d'origine : 666 caractères
Taille de la chaîne compressée : 2694 bits
Taille de la chaîne compressée : 337 octets
meanLength : 4.045045045045045
totalLength : 666
```

Entropie :

```
Entropie de la source : 3.99961307380946
```

On observe une différence cette fois-ci de l'ordre de `10^-3` avec une meilleur compression encore une fois pour l'algorithme de Huffman

Essayons une dernière fois avec une chaîne très longue de plusieurs milliers de caractères.

Huffman résultat :

```
Taille de la chaîne d'origine : 6539 caractères
Taille de la chaîne compressée : 26296 bits
Taille de la chaîne compressée : 3287 octets
meanLength : 4.0214100015292855
totalLength : 6539
```

Shannon-fano résultat :

```
Taille de la chaîne d'origine : 6539 caractères
Taille de la chaîne compressée : 26303 bits
Taille de la chaîne compressée : 3288 octets
meanLength : 4.0224805016057505
totalLength : 6539
```

Entropie :

```
Entropie de la source : 3.9786415046578085
```

Encore une fois, l'algorithme de Huffman reste légèrement meilleur.

On peut en conclure que l'algorithme de Huffman est une meilleure méthode de compression que celle de Shannon-Fano.

En général, l'algorithme de Huffman est considéré comme supérieur à Shannon-Fano en termes d'efficacité de compression, car il garantit l'optimalité des codes générés pour un ensemble donné de symboles et de leurs probabilités. Cependant, Shannon-Fano peut être préféré pour sa simplicité dans certains cas d'utilisation où la différence de performance est négligeable ou lorsque les conditions d'implémentation privilégient la simplicité sur l'efficacité maximale.

b. Comparaison LZ78 / LZW

Pour mesurer l'efficacité de ces méthodes nous allons utiliser le taux de compression qui est égale à la taille de du texte initial divisé par la taille du texte compressé. Un taux de compression de `1.0` signifie qu'il n'y a pas eu de compression du tout, c'est-à-dire que la taille des données initiales est identique à celle des données compressées. Un taux de compression supérieur à `1.0` indique une compression, où les données compressées sont plus petites que les données initiales

```
-- Fonction pour calculer le taux de compression
compressionRatio :: Int -> Int -> Double
compressionRatio initialLength compressedLength = fromIntegral initialLength
/ fromIntegral compressedLength
```

Et voici la fonction de test des méthodes :

```
-- Fonction pour tester la compression et afficher les résultats
testCompression :: String -> IO ()
testCompression input = do
   let compressed = compress input
       decompressed = fromMaybe "" (uncompress compressed)  --
       initialLength = length input
       compressedLength = length compressed
       ratio = compressionRatio initialLength compressedLength
   putStrLn $ "Input initial: " ++ input
   putStrLn $ "Input compressé: " ++ show compressed
   putStrLn $ "Taux de compression: " ++ show ratio
   putStrLn $ "Décompression réussie: " ++ decompressed
   putStrLn ""
```

Ensemble de test :

```
main :: IO ()
main = do
   putStrLn "Test de compression et décompression avec LZ78 :"
   testCompression "abcdefgh"
   testCompression "abababab"
   testCompression "abcabcabcabc"
   testCompression "abababcdeabababcde"
   testCompression "aaaaabbbbbccccc"
   testCompression
"aaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaa
bbbbbaaaaabbbbb"
```

```
   testCompression
"abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvw
xyz"
   testCompression "abcdefghijklmnopqrstuvwxyz1234567890"
```

Résultats LZ78 :

```
Test de compression et décompression avec LZ78 :
Input initial: abcdefgh
Input compressé:
[(0,'a'),(0,'b'),(0,'c'),(0,'d'),(0,'e'),(0,'f'),(0,'g'),(0,'h')]
Taux de compression: 1.0
Décompression réussie: abcdefgh

Input initial: abababab
Input compressé: [(0,'a'),(0,'b'),(1,'b'),(3,'a'),(2,'\NUL')]
Taux de compression: 1.6
Décompression réussie: abababab

Input initial: abcabcabcabc
Input compressé: [(0,'a'),(0,'b'),(0,'c'),(1,'b'),(3,'a'),(2,'c'),(4,'c')]
Taux de compression: 1.7142857142857142
Décompression réussie: abcabcabcabc

Input initial: abababcdeabababcde
Input compressé:
[(0,'a'),(0,'b'),(1,'b'),(3,'c'),(0,'d'),(0,'e'),(3,'a'),(2,'a'),(2,'c'),(5,
'e')]
Taux de compression: 1.8
Décompression réussie: abababcdeabababcde

Input initial: aaaaabbbbbccccc
Input compressé:
[(0,'a'),(1,'a'),(2,'b'),(0,'b'),(4,'b'),(4,'c'),(0,'c'),(7,'c'),(7,'\NUL')]
Taux de compression: 1.6666666666666667
Décompression réussie: aaaaabbbbbccccc

Input initial:
aaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaab
bbbbaaaaabbbbb
Input compressé:
[(0,'a'),(1,'a'),(2,'b'),(0,'b'),(4,'b'),(4,'a'),(2,'a'),(1,'b'),(5,'b'),(6,
'a'),(7,'b'),(9,'b'),(7,'a'),(8,'b'),(9,'a'),(13,'b'),(12,'a'),(16,'b'),(15,
'a'),(11,'b'),(19,'a'),(3,'b'),(21,'a'),(14,'b'),(4,'b')]
Taux de compression: 3.6
Décompression réussie:
```

```
aaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaab
bbbbaaaaabbbbb

Input initial:
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx
yz
Input compressé:
[(0,'a'),(0,'b'),(0,'c'),(0,'d'),(0,'e'),(0,'f'),(0,'g'),(0,'h'),(0,'i'),(0,
'j'),(0,'k'),(0,'l'),(0,'m'),(0,'n'),(0,'o'),(0,'p'),(0,'q'),(0,'r'),(0,'s')
,(0,'t'),(0,'u'),(0,'v'),(0,'w'),(0,'x'),(0,'y'),(0,'z'),(1,'b'),(3,'d'),(5,
'f'),(7,'h'),(9,'j'),(11,'l'),(13,'n'),(15,'p'),(17,'r'),(19,'t'),(21,'v'),(
23,'x'),(25,'z'),(27,'c'),(4,'e'),(6,'g'),(8,'i'),(10,'k'),(12,'m'),(14,'o')
,(16,'q'),(18,'s'),(20,'u'),(22,'w'),(24,'y'),(26,'\NUL')]
Taux de compression: 1.5
Décompression réussie:
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx
yz

Input initial: abcdefghijklmnopqrstuvwxyz1234567890
Input compressé:
[(0,'a'),(0,'b'),(0,'c'),(0,'d'),(0,'e'),(0,'f'),(0,'g'),(0,'h'),(0,'i'),(0,
'j'),(0,'k'),(0,'l'),(0,'m'),(0,'n'),(0,'o'),(0,'p'),(0,'q'),(0,'r'),(0,'s')
,(0,'t'),(0,'u'),(0,'v'),(0,'w'),(0,'x'),(0,'y'),(0,'z'),(0,'1'),(0,'2'),(0,
'3'),(0,'4'),(0,'5'),(0,'6'),(0,'7'),(0,'8'),(0,'9'),(0,'0')]
Taux de compression: 1.0
Décompression réussie: abcdefghijklmnopqrstuvwxyz1234567890
```

Résultats LZW :

```
Test de compression et décompression avec LZ78 :
Input initial: abcdefgh
Input compressé: [97,98,99,100,101,102,103,104]
Taux de compression: 1.0
Décompression réussie: abcdefgh

Input initial: abababab
Input compressé: [97,98,256,258,98]
Taux de compression: 1.6
Décompression réussie: abababab

Input initial: abcabcabcabc
Input compressé: [97,98,99,256,258,257,259]
Taux de compression: 1.7142857142857142
Décompression réussie: abcabcabcabc
```

```
Input initial: abababcdeabababcde
Input compressé: [97,98,256,256,99,100,101,258,257,98,260,101]
Taux de compression: 1.5
Décompression réussie: abababcdeabababcde

Input initial: aaaaabbbbbccccc
Input compressé: [97,256,256,98,259,259,99,262,262]
Taux de compression: 1.6666666666666667
Décompression réussie: aaaaabbbbbccccc

Input initial:
aaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaab
bbbbaaaaabbbbb
Input compressé:
[97,256,256,98,259,259,257,258,260,98,262,97,264,265,262,268,266,271,270,264
,272,275,274,260,276,260]
Taux de compression: 3.4615384615384617
Décompression réussie:
aaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaabbbbbaaaaab
bbbbaaaaabbbbb

Input initial:
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx
yz
Input compressé:
[97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,11
6,117,118,119,120,121,122,256,258,260,262,264,266,268,270,272,274,276,278,28
0,282,259,261,263,265,267,269,271,273,275,277,279,122]
Taux de compression: 1.5
Décompression réussie:
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx
yz

Input initial: abcdefghijklmnopqrstuvwxyz1234567890
Input compressé:
[97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,11
6,117,118,119,120,121,122,49,50,51,52,53,54,55,56,57,48]
Taux de compression: 1.0
Décompression réussie: abcdefghijklmnopqrstuvwxyz1234567890
```

Comparaison et configurations optimales :

Pour LZW, en particulier, les configurations optimales sont celles où il y a de nombreux motifs répétitifs de longueur variable. Plus il y a de motifs répétitifs, plus la taille de la table de dictionnaire grandit, ce qui permet une meilleure compression.

Pour LZ78, les configurations optimales sont similaires, mais cet algorithme a tendance à mieux gérer les motifs répétitifs de longueur fixe ou variable.

En général, LZW a tendance à mieux fonctionner sur des ensembles de données avec des motifs répétitifs et des séquences de caractères plus longues. LZ78 peut être plus efficace pour les ensembles de données avec de nombreux petits motifs.