Université Claude Bernard Lyon 1

Compression

Nicolas Pronost

LIRIS

Principe

- Les algorithmes de compression permettre de réduire la taille des données afin de faciliter le stockage, aider à la transmission à distance etc.
- Il existe des algorithmes
  - sans perte où toute l'information originale est conservée
    - taux de compression faible à moyen, temps de compression rapide à moyen
  - avec perte où une partie de l'information est perdue (on essaye de faire tout de même en sorte que ça soit la moins significative)
    - taux de compression haut, temps de compression lent

Compression sans perte

- Imaginons que l'on veuille compresser un message texte composé de caractères, c'est-à-dire des codes de la table ASCII (ou UTF-8)
- Prenons le texte « je pense, donc je suis. » composé de 23 caractères (on inclut les caractères spéciaux d'espacement, la virgule et le point)
- Chaque code ASCII est codé sur 1 octet (8 bits)
- Une version non compressée de ce texte prendra donc 23 octets, c'est-à-dire 184 bits

Compression sans perte

- Le texte ne comporte pourtant que 13 symboles différents donc on n'a besoin que de 4 bits pour les représenter
  - $2^3 = 8 < 13 < 16 = 2^4$
- En posant par exemple 0000='j', 0001='e', ... , 1100='.'
- Chaque symbole est représenté sur 4 bits donc le texte peut se coder sur $23 \times 4 = 92$ bits (contre 184 sans compression)
- **MAIS** il faut connaître la table de correspondance

Compression par dictionnaire

- C'est le principe de la compression par dictionnaire
  - Il associe une suite de bits à une information (qui peut être une séquence de symboles et pas un symbole isolé)
- On appelle référence une suite de bits associée à une information dans un dictionnaire
- Comme le dictionnaire est construit en fonction de l'information à compresser, il faut l'ajouter à l'information compressée
- Ca n'est donc intéressant que s'il y a beaucoup d'information (message long) et beaucoup de symboles / suites de symboles répétés

Code de Huffman

- Pour améliorer les performances de la compression par dictionnaire, on peut utiliser les références les plus courtes pour les symboles fréquents et les références les plus longues pour les moins fréquentes
- Dans notre exemple, il y a
  - 3 occurrences du symbole ‘s’
  - 2 occurrences des symboles ‘n’, ‘ ‘, ‘e’ et de la séquence ‘je ’
  - 1 occurrence des autres symboles

Code de Huffman

- On peut donc utiliser le dictionnaire suivant (toujours à fournir avec l'information compressée)

's'	000		','	1010
'e'	010		'.'	1011
'je '	011		'c'	00100
'n'	110		'd'	00101
' '	111		'i'	00110
'p'	1000		'o'	00111
'u'	1001			

- Et la phrase sera compressée en une suite de 69 bits:

011100001011000001010101110010100111110001001110110001001001100001011

Que se passe-t-il à la décompression?

- Dans la version de compression par dictionnaire avec des longueurs de référence toujours égales, c'est facile
  - Si cette longueur est de 4 bits, on lit l'information compressée 4 bits par 4 bits, et on converti chaque séquence de bits en son symbole correspondant dans le dictionnaire
- Mais quand la longueur est variable?? Comment sait-on combien de bits il faut lire à chaque fois?
- Il peut apparaître des ambiguïtés
  - imaginons que le symbole 's' soit codé par 000 et le symbole 'p' par 0001, et que l'on lise la suite 0001
  - on ne sait pas s'il faut lire 1 's' puis continuer avec la suite (qui commencerait par un 1) ou bien si c'est 1 'p'
  - 's' est dit un préfixe de 'p'

Décodage de Huffman

- Lorsque l'on créer le code de Huffman, on s'assure qu'aucune référence n'est le préfixe d'une autre

's'	000		','	1010
'e'	010		'.'	1011
'je '	011		'c'	00100
'n'	110		'd'	00101
' '	111		'i'	00110
'p'	1000		'o'	00111
'u'	1001			

011100001011000001010101110010100111110001001110110001001001100001011

Décodage de Huffman

- Lorsque l'on créer le code de Huffman, on s'assure qu'aucune référence n'est le préfixe d'une autre

's'	000		','	1010
'e'	010		'.'	1011
'je '	011		'c'	00100
'n'	110		'd'	00101
' '	111		'i'	00110
'p'	1000		'o'	00111
'u'	1001			

01110000101100000101010111001010011111000100111011000100100110000101‌1

je

Décodage de Huffman

- Lorsque l'on créer le code de Huffman, on s'assure qu'aucune référence n'est le préfixe d'une autre

's'	000		','	1010
'e'	010		'.'	1011
'je '	011		'c'	00100
'n'	110		'd'	00101
' '	111		'i'	00110
'p'	1000		'o'	00111
'u'	1001			

0111000010110000010101011100101001111100010011101100010010011000010 11

je p

Décodage de Huffman

- Lorsque l’on créer le code de Huffman, on s’assure qu’aucune référence n’est le préfixe d’une autre

‘s’	000		‘,’	1010
‘e’	010		‘.’	1011
‘je ’	011		‘c’	00100
‘n’	110		‘d’	00101
‘ ‘	111		‘i’	00110
‘p’	1000		‘o’	00111
‘u’	1001			

0111000010110000010101011100101001111100010011101100010010011000001011

je pe

Décodage de Huffman

- Lorsque l’on créer le code de Huffman, on s’assure qu’aucune référence n’est le préfixe d’une autre

‘s’	000		‘,’	1010
‘e’	010		‘.’	1011
‘je ’	011		‘c’	00100
‘n’	110		‘d’	00101
‘ ‘	111		‘i’	00110
‘p’	1000		‘o’	00111
‘u’	1001			

011100001011000001010101110010100111110001001110110001001001100001011

je pense, donc je suis.

Code de Huffman

- Comment générer les codes de Huffman?

Code de Huffman

- Chaque référence est une feuille d'un arbre binaire
- Le nombre d'occurrence de la référence est associée à la feuille

‘s’ , 3

‘e’ , 2

‘n’ , 2

‘p’ , 1

‘u’ , 1

‘,’ , 1

‘.’ , 1

‘je ’ , 2

‘ ’ , 2

‘c’ , 1

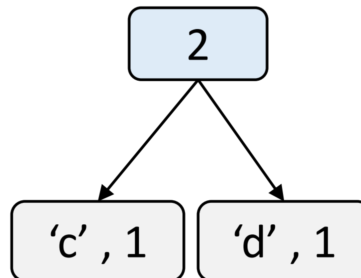
‘d’ , 1

‘i’ , 1

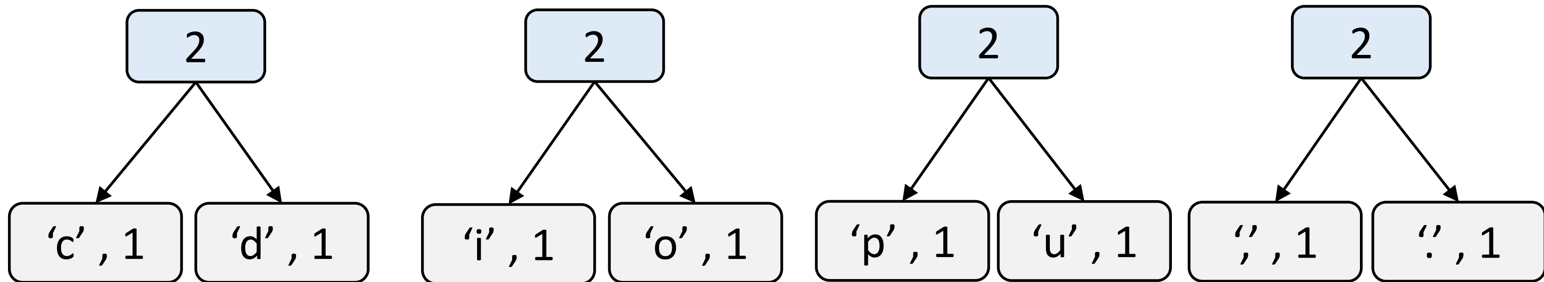
‘o’ , 1

Code de Huffman

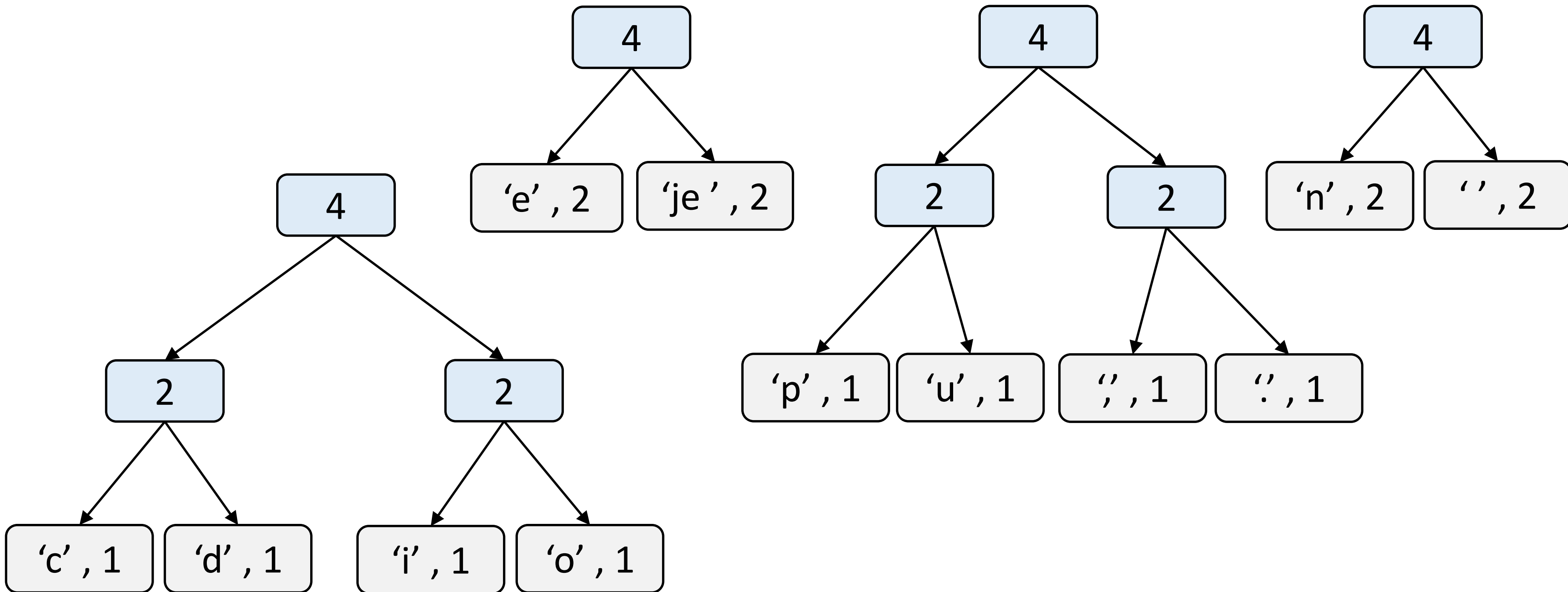
- On associe deux des nœuds de plus faible occurrence pour donner un nœud dont la valeur est la somme des occurrences des deux fils



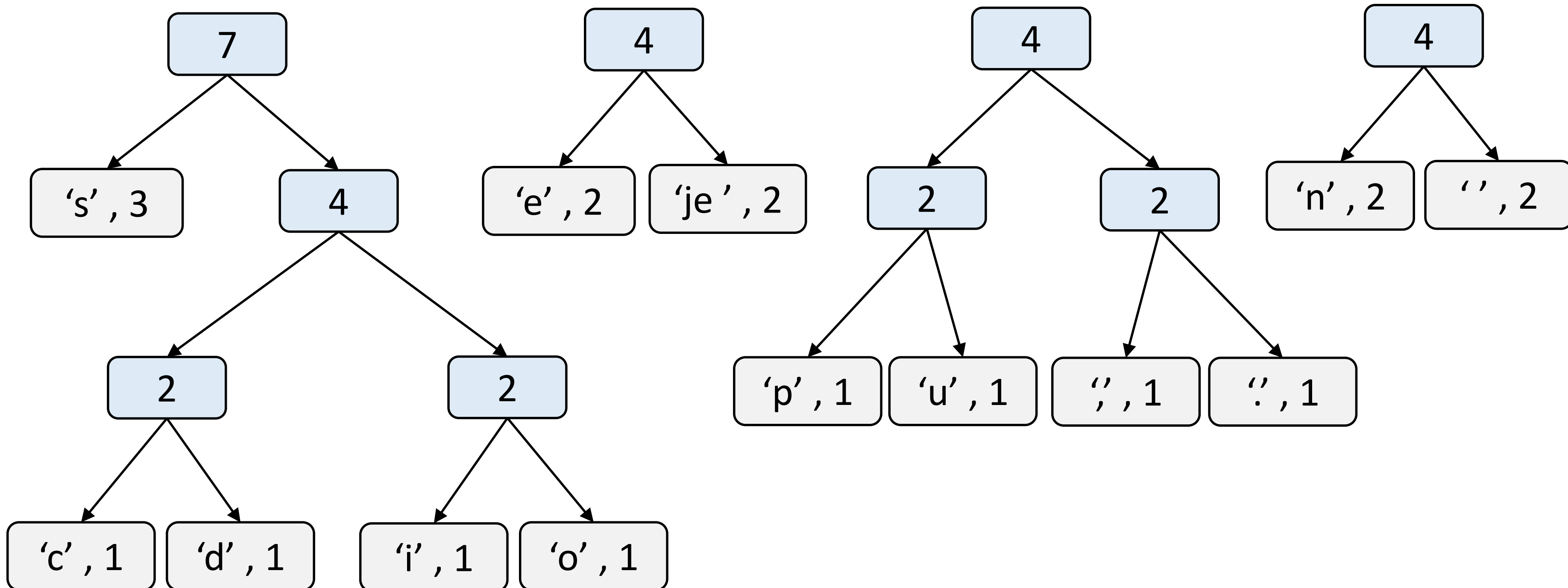
Code de Huffman



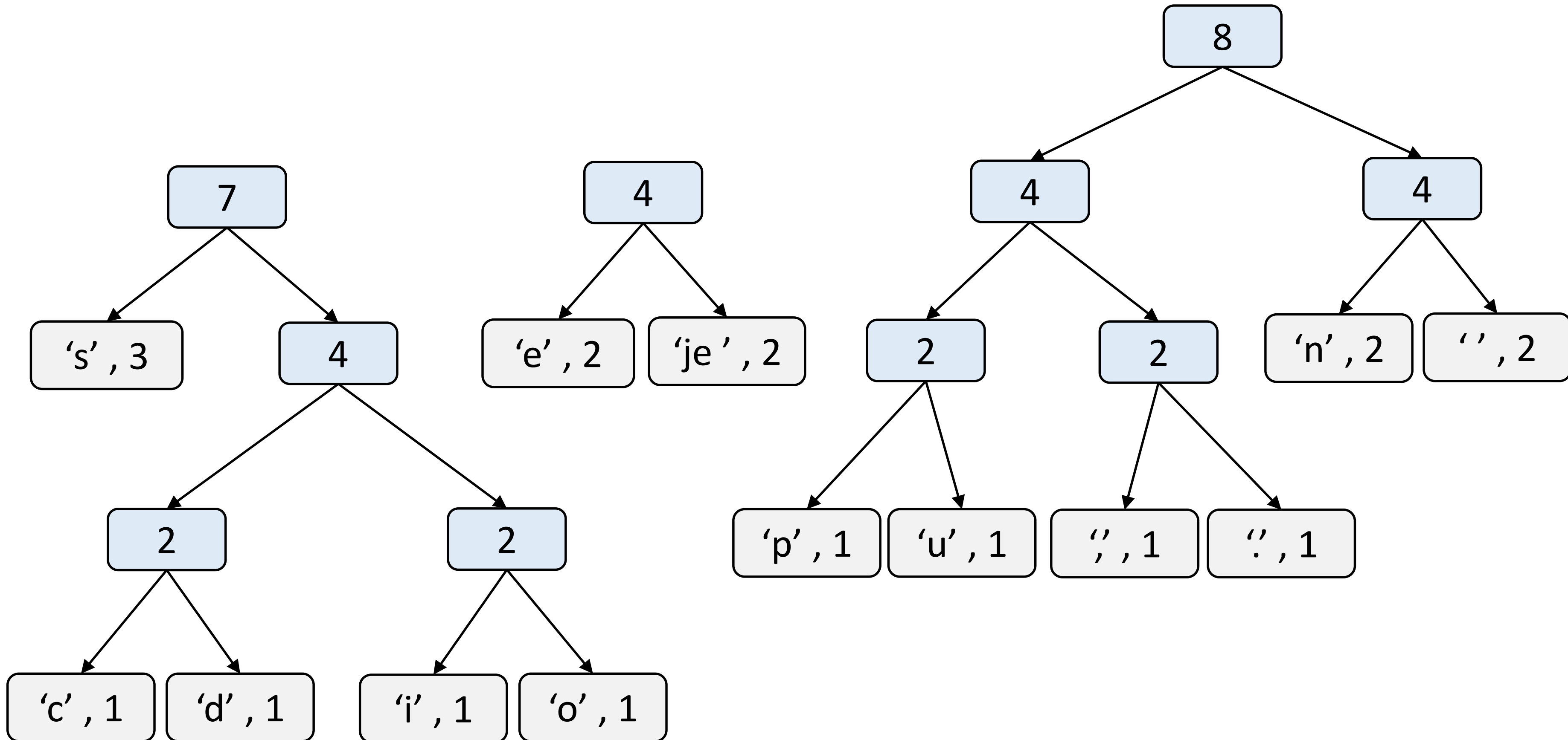
Code de Huffman



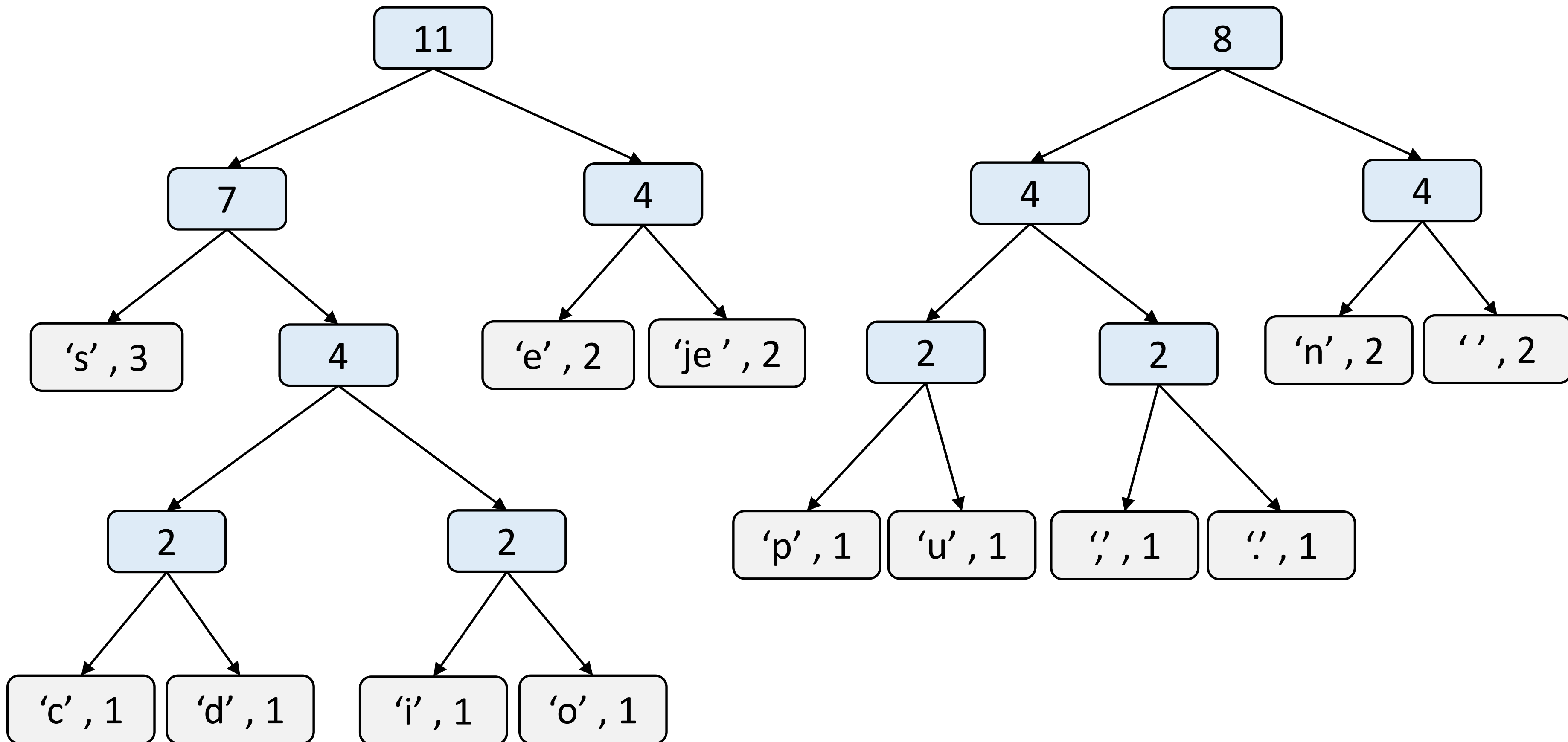
Code de Huffman



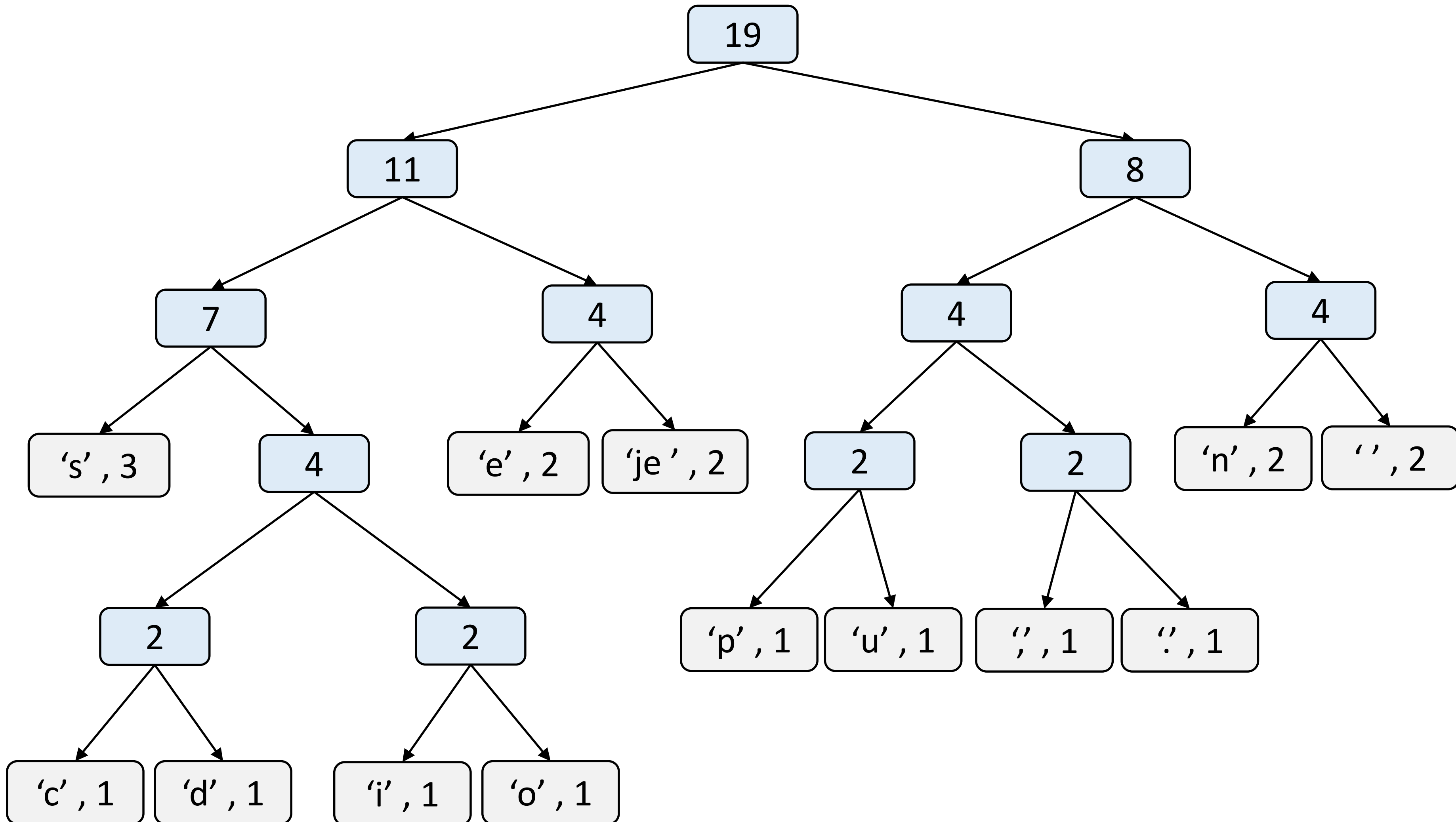
Code de Huffman



Code de Huffman



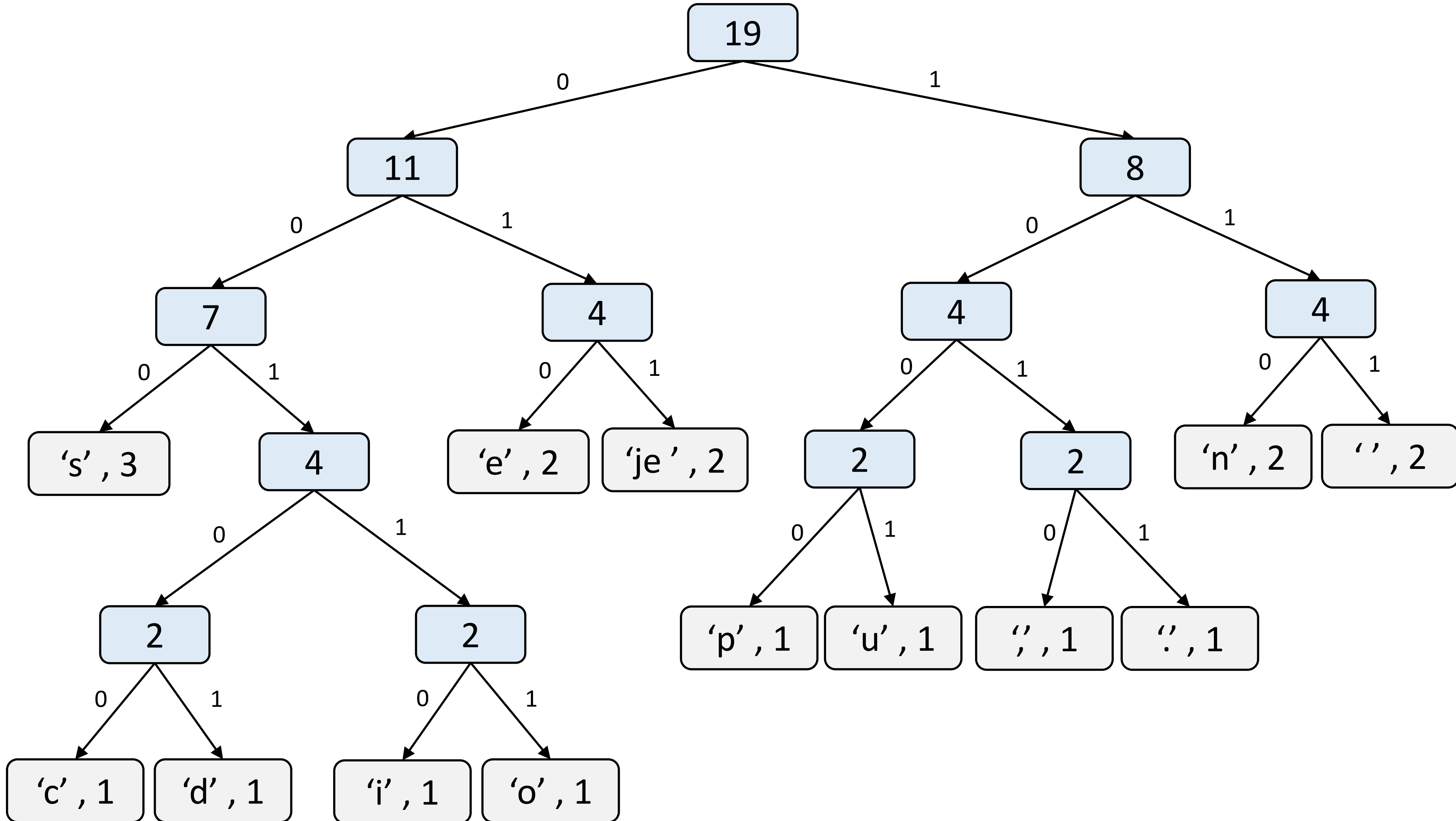
Code de Huffman



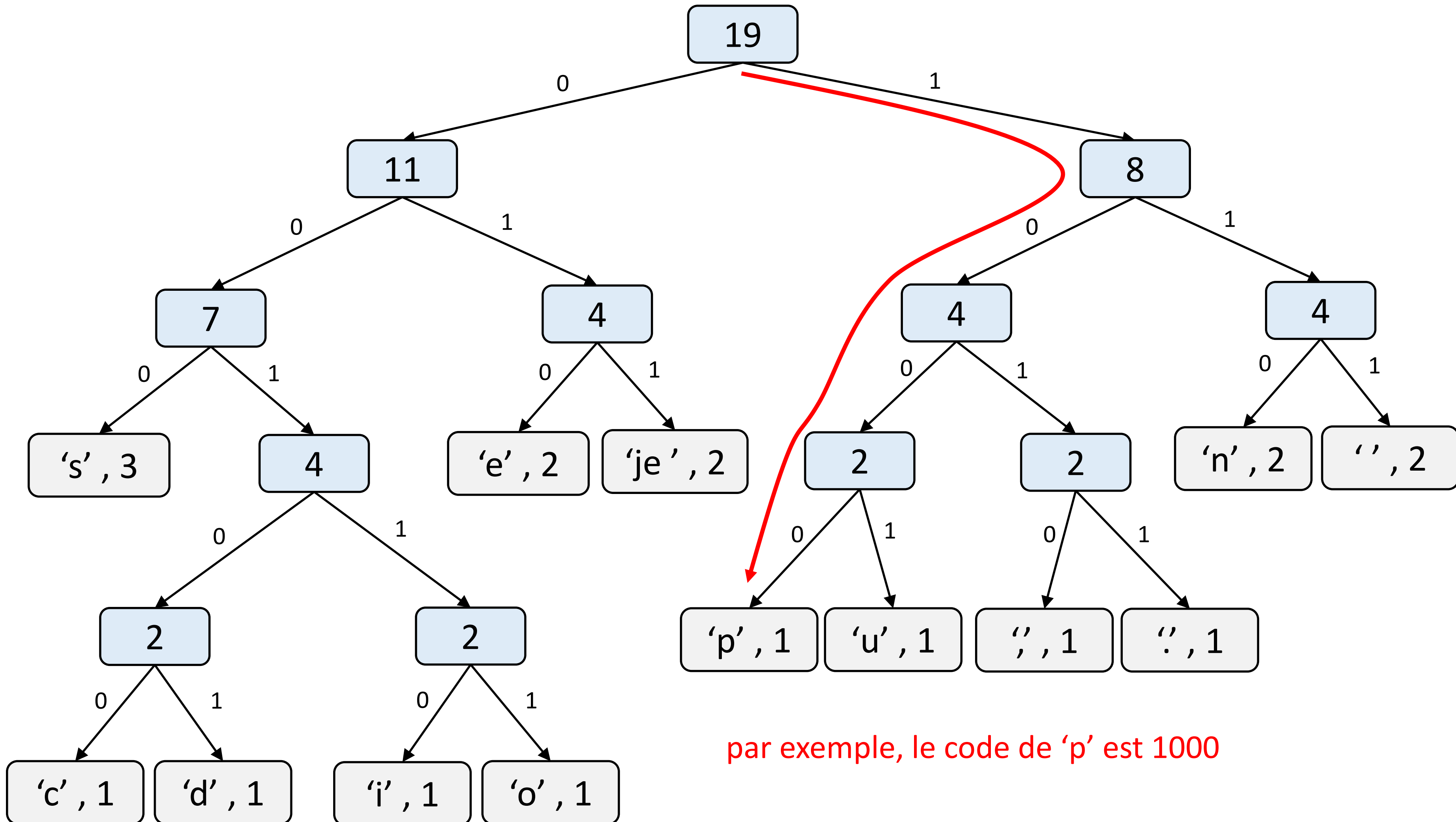
Code de Huffman

- Finalement, on associe le code 0 (ou 1) aux branches gauches (ou droites) et le code 1 (ou 0) aux branches droites (ou gauches)
- Le code de Huffman se lit depuis la racine jusqu'à la feuille référence en concaténant les codes des branches

Code de Huffman



Code de Huffman



Code de Huffman

- Pourquoi cet arbre assure qu'aucune référence n'est un préfixe d'une autre ? Et que les références les plus fréquentes ont les codes les plus courts ?
- Parce que c'est un arbre et que les références sont des feuilles
  - pour qu'une référence soit un préfixe il faudrait qu'elle est une autre référence en tant que fils, or une référence est une feuille
- Parce que les références les moins fréquentes ont été ajoutées à l'arbre en premier et donc sont au bout des branches les plus profondes
- Cet algorithme est utilisé dans la compression ZIP
- Le principe est appliqué dans d'autres contextes tel que la distribution des indicatifs téléphoniques par pays

Compression RLE

- Cf. exercice 16
- Consiste à écrire pour chaque entier d'une séquence d'entiers le nombre de répétitions supplémentaires
  - problème : toutes les séquences non répétitives vont être dilatées
- Amélioration : transformer une séquence d'entiers que si un entier est immédiatement répété au moins une fois
- Lors de la décompression, c'est le fait d'avoir deux entiers identiques à la suite qui indique que l'entier suivant est un nombre d'occurrences supplémentaires et non un entier du fichier de départ
  - favorise les successions de plus de trois caractères identiques
  - défavorise les successions de deux
  - laisse inchangé les simples occurrences

Compression avec perte

- Dans les algorithmes de compression avec perte, certains bits seront exclus (et donc perdus) dans l'information compressée
- C'est souvent le cas des algorithmes de compression d'images et de sons
  - JPG, MP3 etc.
- Exemples d'algorithmes
  - compression par ondelettes
  - transformation de Fourier

Correction

- Des erreurs (au niveau du codage binaire de l'information) est inévitable dues au bruit sur les lignes de transmissions ou les composants électroniques défaillants
- D'où l'utilité d'algorithmes de détection et de correction d'erreur dans les données (compressées ou non)

Correction par redondance

- Une première technique consiste à répéter l'information plusieurs fois (ex. 3 fois)
- Si la suite de bits à transmettre (stocker) est 10110110, alors on transmettra 111000111111000111111000 à la place
- On lit les bits 3 à 3, et si autre chose que 000 ou 111 est lu, alors il y a une erreur
- On peut essayer de corriger, en admettant que 100, 010 ou 001 était en fait originalement 000 (idem pour une majorité de 1)
- Fonctionne à condition qu'il y ait au plus une erreur par triplet
- Méthode couteuse car nécessite une information trois fois plus grande

Détection par parité

- Si la détection d'erreur suffit, on peut utiliser un bit supplémentaire de parité (bit de contrôle) de l'information
  - souvent 1 bit de contrôle par paquet de 100 bits
- Si le bit de contrôle vaut 1, il y a normalement un nombre pair de 1, sinon il y en a un nombre impair
- La taille est augmentée de 1% uniquement
- Permet de détecter mais pas de corriger et si un nombre pair d'erreurs est introduit, l'erreur n'est pas détectée
- De plus si l'erreur est sur le bit de contrôle, on aura détecté une erreur alors qu'il n'y en avait pas...

Correction par parité 2D

- On organise par exemple les données d'un paquet de 100 bits en une liste de 10 lignes et 10 colonnes
- On ajoute 1 bit de contrôle sur chaque ligne et chaque colonne => 10+10=20 bits de contrôle
- On augmente l'information de 20% (mieux que tripler, moins bien que la détection par parité simple)
- Si on détecte une erreur dans une ligne et une colonne, on peut exactement corriger l'erreur (0⇔1)
- Si on détecte que sur une ligne ou une colonne, on suppose que l'erreur est sur le bit de contrôle, on ne fait rien
- Fonctionne à condition qu'une erreur au plus se produise dans chaque suite de 120 bits