

1 Algorithmes de compression d'un texte

1.1 Algorithme de Huffman

On rappelle rapidement le principe de l'algorithme de Huffman :

- on compte les occurrences de chaque caractère du texte à compresser ;
- chaque caractère donne un arbre à 1 nœud, dont la priorité est le nombre d'occurrences de ce caractère ;
- on ajoute tous ces arbres dans une file de priorité (implémentée par un tas-min par exemple) ;
- tant qu'il reste au moins deux arbres dans la file de priorité :
 - on sort les deux arbres de priorités minimales ;
 - on les fusionne ;
 - on rajoute l'arbre obtenu dans la file, avec comme priorité la somme des priorités des deux arbres qu'on a sorti ;
- à la fin, il ne reste qu'un arbre : c'est l'arbre de Huffman pour ce texte ;
- l'encodage d'un caractère est donné par le chemin depuis la racine vers la feuille qui contient le caractère ;
- le nombre de bits du texte compressé vaut $\sum_i p_i \times occ_i$, où p_i est la profondeur du caractère i , et occ_i est son nombre d'occurrences ;
- cet encodage est minimal (les encodages des caractères les plus utilisés dans le texte sont les plus courts) ;
- on a la garantie qu'il n'y a jamais d'ambiguïté dans cet encodage : l'encodage d'un caractère ne peut pas être le préfixe de l'encodage d'un autre caractère ;
- on peut donc sans problème décompresser le texte, à condition d'avoir sauvegarder l'arbre de Huffman qu'on a construit.

1. Compresser la chaîne "charabiabaragouin" à l'aide de l'algorithme de Huffman.

2. Décompresser la chaîne :

1101111100101110111010010110110011100101110100111001101010001101110111100101000

sachant qu'elle a été compressée à l'aide de l'algorithme de Huffman et que la clé de codage est la suivante :

Lettre	a	d	e	i	n	o	s	t	u
Code	0010	11	0100	011	101	100	000	0011	0101

1.2 Algorithme de Lempel-Ziv-Welch

On rappelle rapidement le principe de l'algorithme de LZW :

- on lit petit à petit le texte à compresser ;
- on garde une table de compression, dans laquelle on stocke toutes les sous-chainés qu'on croise, en les numérotant dans l'ordre dans lesquelles elles sont rencontrées pour la première fois ;
- au début, dans cette table, on a ajouté chaque lettre possible ;
- quand on trouve un nouveau sous-mot de la forme wa , c'est que w était déjà présent dans la table, donc :
 - on écrit le numéro de w dans l'encodage compressé ;
 - on ajoute wa à la table :
 - on reprend le processus à partir de la lettre a ;

On obtient une moins bonne compression qu'avec les arbres de Huffman, mais on a deux avantages :

- il suffit de parcourir le texte à compresser une seule fois : pratique si les données à compresser sont reçues en streaming : on peut traiter le flux de données à la volée ;
- une fois la compression terminée, pas besoin de sauvegarder la table de compression : elle peut être reconstruite à la volée et on aura toujours les informations nécessaires à l'instant t pour décoder le caractère courant ;
- il y a une (et une seule) exception à cette propriété : dans ce cas, s'il nous manque une valeur dans la table reconstituée à l'instant t , on sait qu'on est forcément dans le cas suivant :
 - on a construit la table jusqu'au numéro $n - 1$, et il nous faut le numéro n ;
 - la valeur manquante est alors forcément awa , où aw est le dernier morceau qu'on a décodé à l'étape précédente !

3. Compresser la chaîne "AABAACBAAAABBCBAC" à l'aide de l'algorithme LZW en prenant comme dictionnaire de base celui qui suit :

Chaîne	A	B	C
Code	1	2	3

4. Sachant que le dictionnaire initial est constitué des lettres latines minuscules codées par leur position dans l'alphabet (ainsi, la lettre a est associée au code 1 et la lettre z au code 26), décompresser via l'algorithme LZW la chaîne suivante :

16 1 16 15 21 19 27 27 19 28 30 24 33 1 19

2 Algorithmes de recherche d'un mot dans un texte

2.1 Algorithme de Rabin-Karp

Remarque : dans cette partie, on fournit un fichier `revisions_rabin_karp.c` sur la page web du cours.

L'algorithme naïf de recherche d'un mot dans un texte est le suivant :

- on a une **fenêtre glissante** de la longueur du mot recherché, qu'on fait avancer dans le texte ;
- pour chaque position possible de cette fenêtre glissante, on teste si le sous-mot du texte contenu dans cette fenêtre est égal au mot recherché.

5. Écrire une fonction `char* search(char* txt, char* word)` ; implémentant cet algorithme naïf.

Le défaut de cette approche est que, en notant m la longueur du mot recherché, chaque test d'égalité se fait en $\mathcal{O}(m)$ dans le pire des cas. L'algorithme de Rabin-Karp essaye d'améliorer cela (en fait, le cas pire en $\mathcal{O}(m)$ existera toujours, mais il se produira beaucoup moins souvent en pratique).

Le principe de l'algorithme de Rabin-Karp est le suivant :

- on se donne une fonction de hachage sur les chaînes de caractères ;
- on calcule le hash du mot recherché ;
- on reprend l'algorithme précédent, mais on commence par calculer le hash du sous-mot contenu dans la fenêtre glissante, et on compare les hash : s'ils sont égaux (et seulement dans ce cas), on teste l'égalité des chaînes.

Il reste encore un petit problème à régler : calculer le hash du sous-mot contenu dans la fenêtre glissante se fait en $\mathcal{O}(m)$ si on le recalcule naïvement à chaque fois...

Pour éviter cela, on utilise alors une fonction de hachage bien particulière, pour qu'on puisse calculer en $\mathcal{O}(1)$ le hash de la nouvelle fenêtre glissante en fonction du hash de l'ancienne fenêtre glissante.

Pour cela, on utilise la numérotation ASCII pour faire correspondre une valeur entière à chaque caractère, et on calcule le hash d'une chaîne $s = s_0s_1 \cdots s_{m-1}$ via la formule suivante :

$$h(s) = \sum_{i=0}^{m-1} s_i \times d^{m-1-i} \pmod q$$

où $d = 256$ et où q est un entier très grand (mais travailler modulo q permet d'éviter les dépassement d'entiers).

6. Écrire une fonction `int expo(int d, int n, int q)` ; calculant $d^n \pmod q$ en utilisant l'exponentiation rapide.

On fixe dans le fichier deux variables globales `d` et `q`.

7. Écrire une fonction `int convert(char* s, int m)` ; prenant en argument une chaîne de caractères s et sa longueur m , et renvoyant $h(s)$.

8. Donner une formule calculable en $\mathcal{O}(1)$ exprimant $h(s_1 \dots s_n)$ en fonction de $h(s_0 \cdots s_{m-1})$.

9. Compléter la fonction `char* rabin_karp(char* txt, char* word)` ; fournie dans le fichier C.

2.2 Algorithme de Boyer-Moore

On rappelle ici rapidement le principe de l'algorithme de Boyer-Moore :

- on fonctionne toujours avec le même principe de fenêtre glissante ;
- cette fois-ci, pour tester l'égalité entre le mot recherché et le sous-mot contenu dans la fenêtre glissante, on commence par tester le dernier caractère, et on avance petit à petit vers la gauche dans la fenêtre ;
- en cas d'échec, cette fois-ci, on ne décale pas naïvement la fenêtre glissante d'un cran vers la droite, mais on procède comme suit :
 - on regarde quelle est la lettre de la fenêtre glissante pour laquelle on a trouvé le premier **mismatch** ;
 - en analysant le mot recherché, on peut alors calculer de combien de crans d'un coup on peut décaler la fenêtre glissante vers la droite sans risquer de rater le mot ;
 - cette analyse peut être effectuée une seule fois via un précalcul concernant uniquement le mot recherché : on calcule alors ce qu'on appelle la **table des sauts**.

10. Implémenter l'algorithme de Boyer-Moore dans le langage de votre choix.