# TP de théorie de l'information

2A informatique (2018-2019)

*Les TPs sont à faire en binome (écrire vos deux noms en début de fichier). Déposez le(s) fichier(s) source(s) au format c (ou dans une archive au format zip ou tar.gz ; autres formats = -2pts) sur la plateforme, au plus tard 10 minutes après le TP (-2pts si retard constaté). Les TP sont à faire en langage C, avec des commentaires directement dans le fichier source.*

## 1 TP1 : Codage de Huffman (2H)

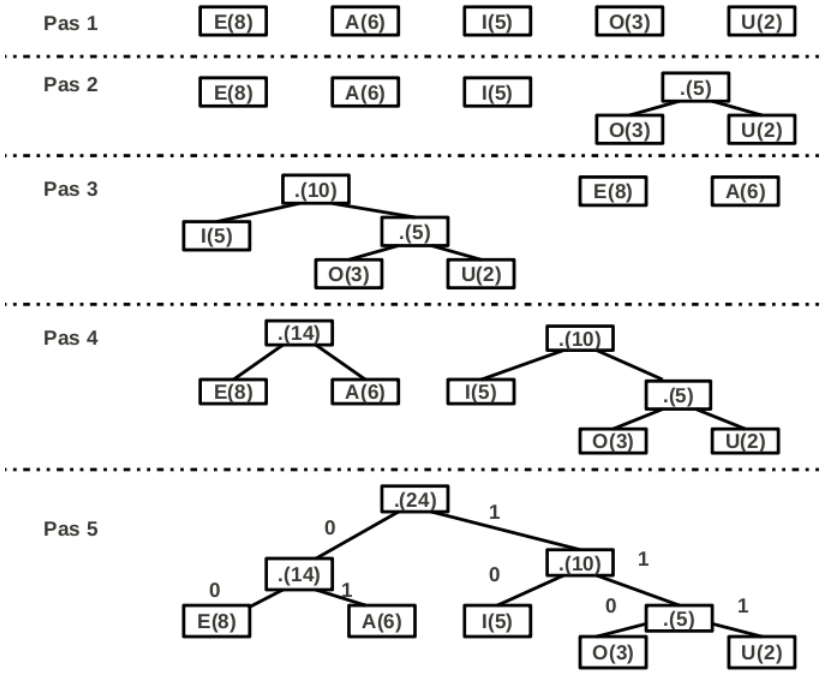
Le codage de Huffman est une technique de compression, sans perte permettant de coder les lettres et les symboles d'un texte en fonction de leur nombre d'occurrences dans le texte, afin de proposer un système de codage plus court que le code ASCII standard. Pour cela, on construit un arbre de Huffman de la manière suivante (voir exemple ci-dessous) :

1. Les feuilles (nœuds à l'extrémité) de l'arbre contiennent chaque lettre (ou symbole) avec un poids associé correspondant à leur nombre d'occurrences, triées de gauche à droite par ordre décroissant.
2. L'arbre est ensuite créé progressivement en groupant chaque couple de nœuds ayant le poids le plus faible pour donner naissance à une nouveau nœud père dont le poids est la somme des deux poids des fils.
3. L'arbre est ainsi construit jusqu'à obtenir un seul nœud qui sera la racine de l'arbre.

Le codage de Huffman consiste ensuite à associer un 0 à chaque branche de gauche et un 1 à chaque branche de droite. Le codage de chaque lettre (ou symbole) se lit alors en partant de la racine jusqu'à arriver à la lettre ou le symbole en question.

**Exemple :** On considère les 5 voyelles de l'alphabet A (6), E (8), I (5), O (3), U (2) où le chiffre entre parenthèses est leur nombre d'occurrences. On construit l'arbre de Huffman comme sur la figure ci-dessous. A la fin de l'algorithme, le codage de Huffman obtenu est le suivant : le codage de A est 01, celui de E est 00, celui de I est 10, celui de O est 110 et celui de U est 111. Les lettres les plus fréquentes sont codées avec le moins de bits possible.

**Objectif du TP :** L'objectif du TP est de faire un programme en langage C qui prend en argument un nom de fichier (les fichiers `test.txt` ou `Huffman.txt` dans notre cas) et réalise le codage de Huffman de ce fichier. Le programme évaluera l'entropie du fichier source (en bits), le taux de compression obtenu, ainsi que la longueur moyenne du code. Le fichier source (`Huffman_initial.c`) à compléter et les fichiers textes sont sur la plateforme.




**Présentation du code source :** Pour simplifier, on considère que le nombre maximal de caractères différents dans le texte est 128, c'est-à-dire uniquement les caractères anglo-saxons dont le code ASCII est compris entre 0 et 127. En particulier, cela ne prend pas en compte les accents (qui ont ainsi été enlevés du fichier `Huffman.txt`). Ceci est pris en compte dans la déclaration de `#define NB_max 256` en début de programme. L'explication de cette déclaration (à priori `#define NB_max 128` semble plus naturel puisqu'il n'y a que 128 caractères différents) devra être justifiée à la fin de la question 2.

**Structure de données :** On définit un nœud de l'arbre par la structure suivante (où les champs `nb_occurence` et `frequence` désignent respectivement le nombre de fois que le symbole apparait dans le texte à compresser et la fréquence correspondante, calculée à partir de `nb_occurence` et du nombre total de caractères) :

```
typedef struct noeud{
  unsigned char symbole ;
  int nb_occurences ;
  float frequence ;
  struct noeud *f_gauche, *f_droit ;
} noeud ;
```

Le programme principal commence par créer une table de symboles `table` définie par la structure de donnée `noeud *table[NB_max]` qui est indexée par le code ASCII du symbole correspondant à partir d'un fichier en entrée de la fonction. Ceci est réalisé par la fonction `initialiser_table` qui

retourne le nombre de caractères du fichier.

*Décrire et expliquer à quoi correspond* `table` *si on lui rentre le fichier* `test.txt`.

Le caractère *retour* sera désormais considéré comme faisant partie du fichier (vérifier le avec la commande `hexdump` sous console). Écrire l'arbre de Huffman correspondant au fichier `test.txt` (en prenant en compte le caractère *retour*).

Cette table ainsi créée comporte de nombreux symboles d'occurrence nulle et qui ne servent pas. On enlève donc ensuite ces symboles, puis on trie la table par ordre décroissant (en fonction du nombre d'occurrences de chaque symbole) par la fonction `trier_table` qui renvoie le nombre de symboles non nuls de cette table. Cette fonction utilise une autre fonction `inserer_noeud` qui insère un nœud dans une table déjà triée par ordre décroissant entre les indices `0` à `l-1` en respectant l'ordre décroissant de la table. A partir du nombre `nb_non_nul` d'éléments non nuls de la table qui vient d'être renvoyé ci-dessus, on peut ensuite calculer la fréquence de chaque symbole, ce qui est réalisé par la fonction `remplir_frequence_table`.

**Travail à réaliser :**

1. Écrire une fonction qui prend en entrée une table de symboles (sous forme de `noeud`) et le nombre de symboles non nuls de cette table et qui calcule l'entropie (binaire) correspondante. On calcule le logarithme en base 2, à l'aide de la fonction `log` (attention, c'est le logarithme néperien), il faudra donc compiler votre programme avec l'option `-lm`. Vous devez trouver une entropie de 2.75 bits pour `test.txt` et 4.38 bits pour le fichier `Huffman.txt`. Comment interpréter la différence d'entropie entre les deux fichiers ?
2. On définit un arbre par la structure `noeud` qui correspond à un pointeur sur la racine de l'arbre. Écrire une fonction qui prend en entrée la table des symboles (triée et débarrassée des symboles d'occurrence nulle) et le nombre de symboles non nuls et qui retourne l'arbre de Huffman correspondant. Il est conseillé d'utiliser les fonctions `nouveau_noeud` et `inserer_noeud`.
3. La fonction `codage` permet de remplir une table `code` qui correspond au code de Huffman de chaque symbole obtenue à partir de l'arbre de Huffman entré en paramètre. Utiliser cette fonction ainsi que la fonction `afficher_codage` pour faire afficher le code de Huffman des fichiers `test.txt` et `Huffman.txt` sur la sortie standard (`stdout`). Vérifier visuellement que dans les deux cas les symboles les plus courants sont codés de manière plus courte que les autres (remarque : cette question consiste simplement à enlever les commentaires dans le fichier source et vérifier que l'arbre de la question précédente est correct).
4. Le nombre de caractères du fichier a été retourné par la fonction `initialiser_table` en début de programme et chaque caractère non compressé est codé en ASCII sur 8 bits. La fonction `nb_bits` retourne le nombre bits du texte codé avec le code de Huffman correspondant. Vérifier que le taux de compression de l'algorithme est de 2.91 pour le fichier `test.txt` et 1.82 pour le fichier `Huffman.txt`.
5. Étudier attentivement et modifier la fonction la fonction `nb_bits` (en lui donnant un autre nom) pour que celle-ci renvoie la longueur moyenne du code de Huffman. Tester la sur le petit fichier test (contenant le terme Huffman) puis sur `Huffman.txt` et comparer le résultat avec l'entropie binaire du texte correspondant. Comment expliquer qu'il y a égalité dans le cas du premier fichier et pas dans le deuxième cas (4.41 vs 4.38) ?

## 2 TP2 : Codage de Hamming (2H)

**Présentation du code de Hamming (7,4) :** Le code de Hamming (7, 4) est un code binaire de cardinal $K = 2^4 = 16$, de longueur $n = 7$ et de distance minimale $d = 3$ (le taux de transmission de ce code est donc $4/7 \simeq 0.57$ et la capacité de correction $e$ du code est 1).

**Encodage :** Pour coder le vecteur binaire $m = (m_1, m_2, m_3, m_4)$, on construit le vecteur binaire $x = (x_1, \ldots, x_7)$, tel que $x_3 = m_1$, $x_5 = m_2$, $x_6 = m_3$, $x_7 = m_4$, et les bits $x_1, x_2, x_4$ sont calculés par les sommes de contrôle suivantes : $x_1 + x_3 + x_5 + x_7 = 0 \bmod 2$, $x_2 + x_3 + x_6 + x_7 = 0 \bmod 2$ et $x_4 + x_5 + x_6 + x_7 = 0 \bmod 2$ (les vecteurs $x$ envoyés ont donc un nombre pair de 1 dans les positions 1,3,5,7 ainsi que 2,3,6,7 et 4,5,6,7). L'ensemble des $2^4 = 16$ vecteurs $x$ possibles calculés comme ci-dessus forment le code de Hamming (7,4) et sont appelés mots de code.
Exemple : Soit $m = 8 = (1, 0, 0, 0)$ le message à coder. Alors $x = 112 = (1, 1, 1, 0, 0, 0, 0)$ (vérifier les indices et les trois sommes de contrôle).

**Décodage :** Soit $y = (y_1, \ldots, y_7)$ le mot binaire reçu après la transmission de $x$. On suppose qu'il y a une erreur dans la composante d'indice $i$ (avec $i = i_0 + 2i_1 + 4i_2$ la décomposition de $i$ en base 2). Si il y a un nombre impair de 1 dans les composantes d'indices 1, 3, 5, 7, alors l'erreur se situe sur un indice $i$ tel que $i_0 = 1$. De même, si il y a un nombre impair de 1 dans les composantes d'indices respectifs 2, 3, 6, 7, et 4, 5, 6, 7, alors l'erreur se situe sur un indice $i$ tel que $i_1 = 1$ et $i_2 = 1$ (respectivement). Ainsi pour décoder, on calcule le *syndrome* $s = (s_2, s_1, s_0)$ de $y$ défini par $s_0 = y_1 + y_3 + y_5 + y_7 \bmod 2$, $s_1 = y_2 + y_3 + y_6 + y_7 \bmod 2$ et $s_2 = y_4 + y_5 + y_6 + y_7 \bmod 2$. Si $s = (0, 0, 0)$ alors il n'y a pas d'erreur et si il y a une erreur, son indice est $s_0 + 2s_1 + 4s_2$.
Exemple : Supposons que l'on reçoit le mot $y = 48 = (0, 1, 1, 0, 0, 0, 0)$. Son syndrome est $s = 1 = (0, 0, 1)$, l'erreur est donc à la place $1 + 0 * 2 + 0 * 4 = 1$ et on corrige $y_1$.

**Structures de données utilisées :** Tous les mots (binaires) de code sont représentés sous forme d'entiers de type `int` de la manière suivante : le mot $x = (x_1, \ldots, x_n)$ est représenté par l'entier dont la décomposition binaire est $\sum_{i=1}^{n} x_i 2^{n-i}$ (le bit de poids faible est donc $x_n$). Un code (est un ensemble de mots de code) est représenté par un tableau d'entier de type `int *`, dont la taille correspond au cardinal du code (16 pour le code de Hamming (7, 4), que l'on définit donc par l'instruction `int *code = (int *) malloc(16*sizeof(int)) ;`).

On obtient le $i$ eme bit du mot de code $x = (x_1, \ldots, x_n)$ par `(x>>(n-i)) & 1`. De même on modifie le $i$ eme bit de $x$ par `x^(1<<(n-i))`. Il est conseillé d'écrire deux petites fonctions `bit(x,i,n)` et `flip(x,i,n)` qui retournent respectivement le $i$ eme bit de $x$ et l'entier $x$ où le $i$ eme bit est modifié. Ainsi, pour vérifier la somme de contrôle $x_1 + x_2 + x_3 = 0 \bmod 2$, on calcule la somme `bit(x,1,n)^bit(x,2,n)^bit(x,3,n)` (où `^` correspond au xor).

**Travail à réaliser :**

1. Écrire une fonction d'encodage qui prend en entrée $m = (m_1, \ldots, m_4)$ de type `int` et retourne le mot de code $x = (x_1, \ldots, x_7)$ correspondant (de type `int`). Vérifier la fonction avec l'exemple de la page 1.
2. Écrire une fonction modélisant un canal symétrique binaire en prenant en entrée deux entiers $x$ et $n$ ($n$ est le nombre de bits du mot $x$, donc $n = 7$ pour le code de Hamming) et un nombre réel $p$, qui est une probabilité vérifiant $0 \leq p \leq 1$ (il est aussi possible de représenter $p$ comme un entier entre 0 et 100 et de s'en servir de seuil). La fonction retourne le vecteur de 7 bits

correspondant à $x$ où chaque bit est modifié par son complémentaire avec la probabilité $p$. On utilise la fonction `srand(time(NULL))` en début de programme pour initialiser le générateur pseudo aléatoire, puis `rand() % N` permet de générer un nombre pseudo aléatoire entre 0 et $N-1$.

3. Écrire une fonction qui prend en entrée un entier $y = (y_1, \ldots, y_7)$ et qui calcule son syndrome $s = (s_2, s_1, s_0)$, de type `int`, puis une fonction `correction` qui prend en entrée un mot $y$ de taille $n$ et de syndrome $s$ et qui retourne un mot corrigé (si le syndrome $s > 0$, on peut alors corriger le mot $y$ par `y = y^(1<<(7-s))`, qui correspond au mot envoyé à travers le canal s'il y a eu au plus une seule erreur durant la transmission). Vérifier la fonction avec l'exemple de la page 1.
4. Écrire une fonction sans paramètre en entrée et qui retourne le code de Hamming (7,4) sous forme d'un tableau `int *`. Écrire une fonction qui prend en entrée $n$ et deux vecteurs $x$ et $y$ de longueur $n$ et retourne la distance de Hamming $d_H(x, y)$ (nombre de composantes différentes entre $x$ et $y$). Écrire une fonction qui prend en entrée $n$, $K$ et un code de longueur $n$ et de cardinal $K$ et qui calcule la distance minimale du code. Vérifier la distance minimale du code de Hamming (7,4).
5. Le décodage est correct si le nombre d'erreurs est inférieur ou égal à la capacité de correction $e$ du code. La probabilité qu'il y ait au plus $e$ erreurs à travers le canal symétrique (de probabilité $p$) est $\sum_{i=0}^{e} C_n^i p^i (1-p)^{n-i}$, où $n$ est la longueur du mot envoyé et le coefficient binomial $C_n^i = \frac{n!}{(n-i)!i!}$. Après avoir écrit les fonctions qui calculent $n!$, un coefficient binomial et une fonction puissance d'un nombre réel, écrire une fonction qui calcule la probabilité que le décodage soit correct à partir de $n$, $p$ et $e$.
   Vérifier que cette probabilité est correcte en générant aléatoirement 10000 entiers (de 4 bits), puis en calculant les mots de code correspondants et en regardant combien ont été décodés correctement après le passage par le canal symétrique binaire de probabilité $p$ (si $p = 0.25$, la probabilité que le décodage soit correct est 0.445 pour le code de Hamming (7,4)).