# TP16: Textes

MP2I Lycée Pierre de Fermat

# OCaml: Codage de Huffman

On rappelle le principe du codage de Huffman: étant donné un texte source $t \in \Sigma^*$, on construit un arbre binaire strict dont les feuilles sont étiquetées par les lettres de $\Sigma$. Puis, on construit un code $C \in \{0,1\}^*$ en remplaçant chaque lettre de $t$ par le chemin qui y mène dans l'arbre. L'efficacité de l'arbre de Huffman vient du fait que les lettres les plus fréquentes sont proches de la racine, et sont donc encodées par peu de bits. Le pseudo code peut donc se décomposer en deux parties:

1. Création de l'arbre de Huffman:

**Algorithme 1 :** Arbre de Huffman

**Entrée(s) :** $t \in \Sigma^*$ un texte source
**Sortie(s) :** $A$ arbre de Huffman

$f \leftarrow$ dictionnaire des fréquences des lettres de $\Sigma$ dans $t$;
$L \leftarrow$ liste d'arbres binaires, initialement vide;
**pour** $a \in \Sigma$ **faire**
    Ajouter à $L$ une feuille $F(a, f[a])$;
**tant que** $L$ ***contient au moins 2 arbres*** **faire**
    Sélectionner $A_1$ et $A_2$ dans $L$ de fréquences minimales;
    Supprimer $A_1$ et $A_2$ de $L$ et les remplacer par un noeud $N(A_1.f + A_2.f, A_1, A_2)$;
**retourner** ***l'unique élément de*** $L$

2. Encodage du texte source:

**Algorithme 2 :** Codage de Huffman

**Entrée(s) :** $t \in \Sigma^*$ un texte source
**Sortie(s) :** Suite de bits encodant $t$, $A$ arbre de Huffman ayant servi à l'encodage

$A \leftarrow$ arbre de Huffman de $t$;
$c \leftarrow$ dictionnaire associant à chaque $a \in \Sigma$ son chemin $c[a]$ dans l'arbre $A$;
$res \leftarrow []$;
**pour** $a$ ***lettre de*** $t$ **faire**
    Ajouter $c[a]$ à $res$;
**retourner** $res$

## Création de l'arbre de Huffman

On propose d'utiliser le type suivant:

```
type huffman =
  | Leaf of int * char
  | Node of int * huffman * huffman ;;
```

L'attribut `int` de chaque constructeur correspond au nombre d'occurence dans le texte source des lettres contenues dans l'arbre, ce que l'on appellera abusivement la fréquence.

**Question 1.** Écrire une fonction `freq: huffman -> int` qui renvoie la fréquence d'un arbre de Huffman.

**Question 2.** Écrire une fonction `fusion : huffman -> huffman -> huffman` qui permet de fusionner deux arbres de huffman en un noeud.

On s'intéresse maintenant à l'initialisation de la liste des feuilles lors de la création de l'arbre de Huffman. Pour cela, on commence par compter le nombre d'occurences de chaque lettre dans le texte source. Afin de stocker cette informations, nous allons utiliser un dictionnaire, implémenté par table de hachage. En OCaml, le module `Hashtbl` implémente cette structure de donnée. Le type `('a, 'b) Hashtbl.t` représente ainsi les tables de hachages où les clés sont `'a` et où les valeurs sont `'b`. On souhaite donc construire à partir d'un texte un objet de type `(char, int) Hashtbl.t` qui à chaque lettre associe son nombre d'occurences dans le texte.

Voici quelques fonctions utiles du module `Hashtbl`:

- `Hashtbl.create n` crée une table de hachage à $n$ alvéoles. La taille est dynamique et change au fur et à mesure que la table grandit, il n'y a donc pas à s'inquiéter des performances, $n$ sert seulement à avoir une estimation initiale. On rappelle qu'idéalement, le nombre d'alvéoles d'une table de hachage doit être de l'ordre du nombre de clés distinctes que l'on utilise. Si l'on manipule des caractères ASCII, on pourra prendre par exemple $n = 128$.
- `Hashtbl.replace tb k v` met à jour la table de hachage $tb$ avec l'association $(k, v)$. Malgré son nom, cette fonction marche même si $k$ n'existe pas dans la table $tb$.
- `Hashtbl.find tb k` renvoie la valeur associée à $k$ dans la table $tb$, et lève une exception si la clé n'apparaît pas.
- `Hashtbl.mem tb k` renvoie un booléen indiquant si la clé $k$ apparaît dans la table $tb$.

Cette structure de table de hachage OCaml est mutable, et donc la fonction `Hashtbl.replace` ne renvoie rien (ou plutôt renvoie `()`). Vous pouvez trouver des informations plus précises sur les fonctions précédentes, ainsi que d'autres fonctions utilitaires, sur la page de documentation de ce module: v2.ocaml.org/api/Hashtbl.html.

**Question 3.** Écrire une fonction `count_freqs : string -> (char, int) Hashtbl.t` qui construit le dictionnaire des occurences à partir d'une chaîne de caractères (*Cette fonction s'écrit plus simplement en impératif qu'en fonctionnel*).

**Question 4.** Regarder la documentation de la fonction `Hashtbl.fold` et l'utiliser pour écrire une fonction `huffman_leaves : string -> huffman list` qui prend en entrée un texte $t$ et renvoie une liste contenant pour chaque caractère $c$ de l'alphabet utilisé dans $t$ une feuille étiquetée par ce caractère ainsi que sa fréquence dans le texte.

La prochaine étape est de réduire la liste des arbres (initialement tous des feuilles), en fusionnant à chaque fois ses deux éléments de fréquences minimales. Pour cela, nous allons garantir comme invariant que la liste des arbres est toujours triée par fréquence croissante. Au départ, il faudra donc trier la liste des feuilles.

**Question 5.** Écrire une fonction `merge_sort: huffman list -> huffman list` triant une liste d'arbres par ordre croissant de fréquences, suivant le principe du tri fusion.

**Question 6.** Écrire une fonction `insert_huffman : huffman -> huffman list -> huffman list` qui prend en entrée un arbre de Huffman $h$ et une liste d'arbres $l$ dont on suppose qu'ils sont triés par fréquence croissante, et qui insère $h$ dans $l$ au bon endroit.

**Question 7.** En déduire une fonction `fuse_list : huffman list -> huffman` qui prend en entrée une liste d'arbres de Huffman supposée triée, et qui renvoie l'arbre de Huffman obtenu en ayant itéré le processus de fusion jusqu'à ce que la liste soit réduite à un élément.

**Question 8.** Déduire des fonctions précédentes une fonction `huffman_tree : string -> huffman` qui construit l'arbre de Huffman d'un texte.

**Question 9.** Quelle est la complexité de cette fonction ?

## Codage de Huffman

Nous pouvons maintenant passer à la partie codage. Pour cela, il faut commencer par construire un dictionnaire, donnant pour chaque lettre le chemin qui y mène dans l'arbre de Huffman. On cherche donc à construire une table de type `(char, bool list) Hashtbl.t`. L'idée est d'effectuer un parcours de l'arbre, en profondeur pour plus de simplcité, en gardant à chaque fois en mémoire le chemin parcouru depuis la racine. Pour des raisons de complexité, **on stockera le chemin parcouru à l'envers**. Par exemple, si l'on a suivi le chemin (gauche, gauche, droite), on utilisera la liste `[true; false; false]`. Ceci permet de rajouter une arête au bout du chemin en temps constant.

**Question 10.** Écrire une fonction

```
build_path_table: (char, bool list) Hashtbl.t -> huffman -> bool list -> unit
```

qui prend en entrée une table de hachage, un arbre de Huffman, et une liste de booléens correspondant au chemin déjà parcouru (stocké à l'envers), et qui rajoute à la table les associations (caractère, chemin) dans l'arbre.

**Question 11.** En déduire une fonction `compression_huffman : string -> bool list * huffman` qui prend en entrée un texte et renvoie un couple $(C, h)$ où $h$ est l'arbre de Huffman calculé à partir du texte et $C$ le texte codé en utilisant $h$.

**Question 12.** Écrire une fonction `compare: string -> unit` qui calcule le nombre de bits utilisés pour encoder une chaîne via l'algorithme de Huffman, et sans encodage, et qui affiche les deux nombres ainsi que leur ratio. Pour calculer le nombre de bits utilisés sans encodage, on compte le nombre de lettres $p$ distinctes utilisées, et on considère que chaque lettre du texte prend $\lceil \log 2p \rceil$ bits.

**Question 13.** (Bonus) Renseignez-vous sur les fonctions `open_in: string -> in_channel` et `input_line: in_channel -> string` qui permettent de lire des lignes dans des fichiers. Utilisez

les pour tester l'efficacité de l'algorithme de Huffman sur le fichier "vingt_mille_lieues.txt" de l'archive. **Attention:** pour que votre code fonctionne, il faudra probablement coder votre fonction d'encodage en récursif terminal !

## Décodage de Huffman

La dernière étape de cette section est la mise en place de l'algorithme de décompression associé. On en rappelle le principe: étant donné une suite de bits $C$ et un arbre de codage $h$, on lit $C$ en suivant dans $h$ le chemin correspondant. Lorsque l'on arrive à une feuille dans $h$, on note le caractère lu et on revient à la racine.

**Question 14.** Écrire une fonction `read_path : huffman -> bool list -> char * bool list` prenant en entrée un arbre de Huffman $h$ et une suite de booléens $C$, qui renvoie le caractère lu sur la feuille atteinte en suivant le chemin indiqué par $C$ dans $h$, ainsi que la liste des booléens non lus restants. Par exemple, si la liste $C$ est $[1, 0, 0, 1, 0]$ et que l'on atteint une feuille en allant à droite (1) puis à gauche (0), la liste restante sera $[0, 1, 0]$.

**Question 15.** En déduire une fonction `decompression_huffman : huffman -> bool list -> string` qui décode une liste de booléens en utilisant un arbre de huffman, et qui renvoie le texte ainsi reconstruit. On pourra utiliser la fonction `String.make` pour transformer un caractère en un string de longueur 1, et utiliser l'opérateur de concaténation `^`.

**Question 16.** Testez la correction de votre fonction.

## Bonus: LZW

**Question 17.** En utilisant le pseudo-code donné dans le cours, implémentez l'algorithme de compression de LZW. On pourra utiliser une fonction auxiliaire pour simuler un tour de boucle, sa signature sera:

```
1 let rec step_lzw (i: int) (w: string) (next_code: int) (res: int list) =
```

Avec $i$ l'indice actuel dans le texte source, $w$ le mot en cours de lecture, *next_code* le prochain code à utiliser pour le dictionnaire, et *res* un accumulateur avec la liste des codes générés jusqu'à maintenant, à l'envers.

**Question 18.** Modifier votre implémentation de l'algorithme LZW pour pouvoir spécifier un code maximal, au delà duquel on ne rajoute plus de mots au dictionnaire.

**Question 19.** Avec différents codes maximaux (idéalement des puissances de 2), comparer les performances de l'algorithme de Ziv-Lempel-Welch avec celles de l'encodage de Huffman.

# C: Recherche de motif

L'archive du TP contient quatre fichiers texte:

- "gene.txt" qui contient la séquence ADN du gène CTFR [1]. Une mutation de ce gène peut entrainer la mucoviscidose;
- 'ADN.txt" qui contient un morceau de l'ADN d'une personne, dont on souhaite déterminer si son gène CTFR a subi une mutation;
- "vingt_mille_lieues.txt", qui contient le texte de Vingt Mille Lieues sous les mers;
- "kraken.txt", qui contient un extrait de Vingt Mille Lieues sous les mers, dans lequel le narrateur décrit l'apparition d'un calamar géant.

Nous allons utiliser ces 4 fichiers pour tester les performances de différents algorithmes de recherche de motif. On a donc deux tests de performance différents:

- Chercher le gène dans le fragment de séquence ADN donné.
- Chercher l'extrait de Vingt Mille Lieues sous les mers.

Cependant, il faudra aussi écrire vos propres tests, plus courts, pour vous aider à débugger vos programmes au fur et à mesure.

**Question 20.** Implémenter une fonction `int recherche_naive(char* texte, char* motif)` qui recherche selon une méthode naïve la présence d'un motif dans un texte. On renverra le premier indice où le motif commence dans le texte, et $-1$ si le motif n'apparaît pas.

**Question 21.** Mesurer les temps d'exécutions de cet algorithme sur les deux tests proposés.

## Rabin-Karp

Afin d'accélérer un peu la recherche, nous allons implémenter l'algorithme de Rabin-Karp. Cet algorithme est une amélioration de la méthode naïve, qui consiste à calculer un hash du motif, et à le comparer au hash de chaque fenêtre du texte, afin de ne faire la comparaison que si les hash sont identiques. Pour que cela soit efficace, il faut pouvoir recalculer le hash d'une fenêtre facilement à partir du hash de la fenêtre précédente. Pour l'algorithme de Rabin-Karp, on prend $q$ un nombre premier assez grand, et $B$ le nombre de lettres possibles (on prendra $B = 256$ pour les caractères ASCII), et l'on considère un mot $m_0, \dots m_{p-1}$ comme un nombre en base $B$ pris modulo $q$. Autrement, la fonction de hachage est:

$$h : u_0 \dots u_{p-1} \mapsto \sum_{i=0}^{p-1} u_{p-1-i} B^i \mod q$$

**Question 22.** On considère un texte $t = t_0 \dots t_{n-1}$. Donner la formule liant $h(t_{i+1} \dots t_{i+p})$ et $h(t_i \dots t_{i+p-1})$.

**Question 23.** Implémenter l'algorithme de Rabin-Karp, et comparer ses performances avec l'algorithme naïf.

[1] Les données ont été modifiées pour mieux s'adapter au TP

## Boyer-Moore

Implémentons maintenant l'algorithme de Boyer-Moore-Horspool, version plus simple de Boyer-Moore, qui consiste à utiliser seulement la règle du mauvais caractère. Cette règle utilise la définition suivante:

**Définition 1.** Soit $m \in \Sigma^*$ un motif de taille $p$ et $a \in \Sigma$. La dernière occurence non-finale de $a$ dans $m$, notée $d_m(a)$, est définie par:

$$d_m(a) = \max\{i \in [\![0, p-2]\!] \mid m_i = a\}$$

en prenant comme convention $\max \emptyset = -1$.

En utilisant $d_m$, lorsque l'on compare le motif à une fenêtre du texte, en cas de non-concordance, on peut éliminer directement plusieurs autres positions successives.

**Proposition 1** (Règle du mauvais caractère)**.** Soit $t$ un texte et $m$ un motif. Supposons que $m$ ne se trouve pas à la position $i$ dans $t$ et que $j$ est le premier indice en partant de la droite pour lequel $m_j \neq t_{i+j}$. Alors, la prochaine position où $m$ peut apparaître est $i' = i + j - d_m(t_{i+j})$.

Cette règle donne lieu à l'algorithme de Boyer-Moore-Horspool:

**Algorithme 3 :** Recherche de motif: Boyer-Moore-Horspool

**Entrée(s) :** $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
**Sortie(s) :** $i$ tel que $m$ apparaît à la position $i$ de $t$

```
1  i ← 0;
2  tant que i < n − p + 1 faire
3      j ← p − 1;
4      tant que j ≥ 0 et t_{i+j} = m_j faire
5          j ← j − 1;
6      si j = −1 alors
7          retourner i
8      sinon
9          i ← i + max(1, j − d_m(t_{i+j}));
10 retourner Pas d'occurence
```

**Question 24.** Écrire une fonction `int* construire_d(char* m)` qui renvoie un tableau de taille 256 donnant les valeurs de $d_m(a)$ pour $a$ dans l'alphabet ASCII. Cette fonction devra être de complexité linéaire en la taille de $m$.

**Question 25.** Implémenter l'algorithme de Boyer-Moore-Horspool, et le comparer aux algorithmes précédents sur les fichiers donnés.

## Bonus: Règle du bon suffixe

On s'intéresse enfin à l'implémentation de l'algorithme de Boyer-Moore en entier. Autrement dit, nous allons implémenter la règle du bon suffixe. On rappelle que pour cette règle, on précalcule sur le motif $m$ de taille $p$ deux tableaux $p_m$ et $s_m$:

- Pour $j \in [\![1, p-1]\!], p_m(j) = \max\{k \in [\![0, p-j]\!] \mid m_0 \dots m_{k-1} = m_{p-k} \dots m_{p-1}\}$
- $p_m(0) = p_m(1)$
  Pour $j \in [\![1, p-1]\!], s_m(j) = \max\{k \in [\![0, j-1]\!] \mid m_k \dots m_{k+p-j-1} = m_j \dots m_{p-1}$ et $m_{k-1} \neq m_{j-1}\}$
- $s_m(p) = p - 1$

L'algorithme de Boyer-Moore utilise ces deux tableaux pour accélérer la recherche naïve:

**Algorithme 4 :** Recherche de motif: Boyer-Moore

**Entrée(s) :** $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
**Sortie(s) :** Liste des indices $i$ tel que $m$ apparaît à la position $i$ de $t$

```
1  D, P, S ← tableaux stockant d_m, p_m et s_m;
2  L ← [] // résultat de l'algorithme, liste des positions
3  i ← 0;
4  tant que i ≤ n − p faire
5      j ← p − 1;
6      tant que j ≥ 0 et t_{i+j} = m_j faire
7          j ← j − 1;
8      si j = −1 alors
9          Ajouter i à L;
10         i ← i + p − P[1];
11     sinon
12         si S[j + 1] ≥ 0 alors
13             i ← i + max(1, j − D[t_{i+j}], j + 1 − S[j + 1]);
14         sinon
15             i ← i + max(1, j − D[t_{i+j}], p − P[j + 1]);
16     retourner L
17 retourner Pas d'occurence
```

**Question 26.** Écrire deux fonctions `construire_p` et `construire_s` prenant en entrée une chaîne de caractères et construisant les deux tableaux supplémentaires utilisés pour l'algorithme de Boyer-Moore. **Attention: pour que la complexité de l'algorithme soit convenable, ces deux fonctions doivent être en temps quadratique au pire. C'est même possible de construire les tables en temps linéaire.**

**Question 27.** Implémentez l'algorithme de Boyer-Moore, et comparez ses performances avec les algorithmes précédents.