Séquence 13 - Algorithmique des textes

Cette deuxième partie consacrée aux algorithmes des textes aborde la question de la compression des données à travers les algorithmes de Huffman et de Lempel–Ziv–Welch (LZW).

Table des matières

I. Manipulation des caractères et chaînes de caractères en C et en OCaml

I. 1. En C

En C, les chaînes de caractères sont de **simples tableaux de caractères**, donc **intrinsèquement mutables**, comme n'importe quel tableau. Une chaîne de caractères est donc une adresse vers le premier caractère d'un tableau de caractères, c'est un `char *`. Tout chaîne de caractères bien formée se termine par le caractère sentinelle `'\0 '` qui évite avoir à systématiquement passer la taille de la chaîne de caractère en plus de l'adresse `char *`. Attention toutefois, de bien vous assurer que l'espace alloué pour le stockage d'une chaîne comporte toujours 1 octet de plus pour le stockage du caractère sentinelle.
La bibliothèque `string` disponible de manière native sur la plupart des systèmes fournit des fonctions très utiles, qu'il est cependant nécessaire de savoir coder par vous-même car elles sont hors-programme :

— `strlen(char *s)` : renvoie le nombre de caractères de la chaîne `s`, sans le caractère sentinelle

— `strcpy(char *destination, const char *source)` : copie le contenu d'une chaîne pointée par `source` dans la zone mémoire pointée par `destination`. Attention, `destination` doit pointer vers un espace qui a été préalablement alloué, soit par une déclaration de tableau statique `char destination[10]` par exemple, dans l'un des blocs d'activation de la pile, soit dans le tas avec un `malloc`. Attention également, la taille allouée pour `destination` doit être suffisante pour accueillir tous les ca-

ractères de `source`, et `source` doit bien se terminer par le caractère sentinelle, sans quoi des erreurs de segmentations vont se produire.

— `strncpy(char *destination, const char *source, size_t nc)` : même fonctionnement que `strcpy`, mais plus sécurisé dans le sens où l'on indique explicitement le nombre `nc` de caractères que l'on copiera au maximum. Si `source` possède plus de caractères que `nc`, ils ne seront pas copiés.

— `char *strdup(const char *source)` : (*string duplication*) il s'agit là encore d'une fonction permettant de copier une chaîne de caractères, mais celle-ci gère l'allocation de l'espace mémoire de la nouvelle chaîne. Cette allocation est faite dans le tas : il faudra donc penser à désallouer cet espace avec `free`, même si le `malloc` est caché à l'intérieur de `strdup`. La fonction renvoie l'adresse du premier caractère de la chaîne dupliquée et dans laquelle on a copié les caractère de la chaîne source.

— `int strcmp(const char *s1, const char *s2)` : (*string comparison*) compare deux chaînes de caractères en utilisant la relation d'ordre lexicographique. La comparaison s'arrête lorsque l'on atteint le caractère sentinelle de l'une des deux chaînes. Cette fonction mesure une **différence** lexicographique entre les deux chaînes. Elle renvoie 0 si les deux chaînes sont identiques, une valeur négative si `s1` est plus petite au sens lexicographique, et supérieur à 0 si `s1` est plus grande.

— `strncmp(const char *s1, const char *s2, size_t nc)` : comparaison plus sécurisée, où l'on compare au maximum `nc` caractères.

— `strcat(char *destination, const char *source)` : concatène la chaîne `source` (qui reste inchangée) à la suite de la chaîne `destination`. Attention, là encore, à la présence de caractères sentinelle et à l'allocation préalable de l'espace mémoire pour la chaîne `destination`, qui doit pouvoir accueillir tous les caractères concaténés !

— `strncat(char *destination, const char *source, size_t nc)` : version de la concaténation où l'on concatène au maximum `nc` caractères.

Le mot-clé `const` devant un paramètre d'entrée indique qu'il sera impossible, une fois à l'intérieur de la fonction, de modifier le contenu de la variable locale au bloc d'activation contenant la copie du paramètre d'entrée. Elle est utilisée pour sécuriser une valeur d'entrée en s'assurant qu'elle ne sera pas modifiée par mégarde par la fonction alors qu'elle ne doit pas l'être.
Lorsque l'on souhaite analyser seulement une sous-chaîne d'une chaîne de caractères, on peut procéder de manière astucieuse en utilisant l'adresse du premier caractère de la sous-chaîne que l'on souhaite analyser, par exemple avec une syntaxe comme celle-ci, qui permet de considérer la sous-chaîne commençant au caractère d'indice `idx0` de la chaîne `text`.

```
char *souschaine = &(text[idx0]);
```

I. 2. En OCaml

En OCaml, il existe 3 types de chaînes de caractères :

Immuables (et donc évidemment de taille fixée) : type prédéfini `string`

Mutables de taille fixée : type prédéfini `bytes`

Mutables de taille variable : type `Buffer.t` défini dans le module `Buffer`

I. 2. a. Type `string` et module `String`

Les objets de type `string` des chaînes de caractères immuables en OCaml. Il était auparavant possible de modifier en place un caractère d'une `string`, mais cette syntaxe est obsolète et le type `string` OCaml est désormais immuable.

```
# let s = "danger";;
val s : string = "danger"
# s.[0] <- 'm';;
Characters 0-12:
  s.[0] <- 'm';;
  ^^^^^^^^^^^^
Warning 3: deprecated: String.set
Use Bytes.set instead.
- : unit = ()
```

Lecture d'un caractère de la chaîne : l'accès à un caractère d'une `string text` se fait avec la syntaxe `text.[i]`.

Comparaison de chaînes : La comparaison de deux chaînes de caractères s'effectue grâce à l'opérateur `=`, qui a été surchargé pour gérer la comparaison de chaînes en utilisant la relation d'ordre lexicographique.

Concaténation de chaînes : l'opérateur de concaténation de chaînes de caractères se note `^` (attention à ne pas le confondre avec l'opérateur de concaténation de liste, ne s'applique pas au même type d'objet et se note `@`). Cette opération préserve l'immuabilité des `string` données en entrée, notamment celle de la première chaîne. Tout comme pour les listes, la première chaîne de caractère est en fait recopiée dans un nouvel espace mémoire correspondant au résultat concaténé.

```
# let s = "danger";;
val s : string = "danger"
# let s2 = s^"osite";;
val s2 : string = "dangerosite"
# s;;
- : string = "danger"
```

On peut artificiellement créer des chaînes de caractères mutables en utilisant des références sur des objets de type string.

```
# let refs = ref "danger";;
val refs : string ref = {contents = "danger"}
# refs := !refs^"osite";;
- : unit = ()
# !refs;;
- : string = "dangerosite"
```

Mais ils s'agit bien ici d'une illusion de mutabilité car la structure n'est pas modifiée en place, dans la même zone mémoire : un nouvel espace mémoire est en fait crée dans le tas pour accueillir la nouvelle chaîne, et l'on modifie simplement la référence, pour qu'elle désigne le nouvel espace alloué.

I. 2. b. Type `bytes` implémentant des chaînes de caractères mutables de taille fixée

Pour faire de vraies chaînes de caractères mutables, modifiables en place, il faut créer et manipuler des objets de type `bytes`. *byte* signifie octet en anglais. Comme un caractère correspond à un octet, le type `bytes` correspond a une série d'octets, donc une chaîne de caractères. Ce type est prédéfini en OCaml et peut être utilisé comme n'importe quel

autre type, dans d'autres définitions de types (enregistrements, constructeurs pour les types sommes...etc)
Pour créer et manipuler des chaînes mutables `bytes`, on utilise généralement le module `Bytes`.

```
# let b = Bytes.of_string "danger";;
val b : bytes = "danger"
# Bytes.set b 0 'm';;
- : unit = ()
# b;;
- : bytes = "manger"
# let s = Bytes.to_string b;;
val s : string = "manger"
```

I. 2. c. Type `Buffer.t` implémentant des chaînes de caractères mutables de taille variable

Si l'on souhaite travailler sur de vraies chaînes mutables en place, et de taille pouvant varier, par exemple des chaînes subissant des concaténations, il existe un module `Buffer` implémentant des chaînes de caractères comme une structure de pile implémenté concrètement avec des tableau redimensionnables. Cette implémentation permet des coûts d'accès en temps constant à tout caractère de la chaîne, mais optimise aussi les modifications de la taille de la chaîne, car nous avons vu que le coût amorti d'une opération de type `push` (en fin de tableau dans cette implémentation) est constant, les pics de complexité liés aux éventuels redimensionnements étant suffisamment rares pour que leur coût sur une série d'appels puisse être considéré comme lissé. Le type OCaml implémentation la structure de donnée (contenant notamment le tableau redimensionnable) est noté `Buffer.t`. Il est rendu abstrait, c'est-à-dire qu'il n'est pas exposé dans les détails techniques à l'utilisateur : c'est une illustration de la notion de barrière d'abstraction entre implémentation concrète et interface abstraite. Nous n'avons accès qu'à l'interface abstraite en tant qu'utilisateur et n'avons nul besoin de nous plonger dans les méandres techniques de l'implémentation :

- `Buffer.create init_capacity`: créer une structure de chaîne en allouant un premier tableau de taille `init_capacity`, qui pourra être redimensionné par la suite en fonction des opérations effectuées et de l'allongement de la chaîne
- `Buffer.add_char buf c`: ajout un caractère `c` au bout de la chaîne mutable `buf`, équivalent de `append`
- `Buffer.add_string buf s`: ajoute une chaîne caractère `s` au bout de la chaîne mutable `buf`, équivalent à une concaténation
- `Buffer.contents buf`: renvoie un objet de type `string` correspondant au contenu du buffer de chaîne de caractère mutable.

Voici un petit exemple d'utilisation :

```
# let buf = Buffer.create 10;;
val buf : Buffer.t = <abstr>
# Buffer.add_string buf "danger";;
- : unit = ()
# let s = Buffer.contents buf;;
val s : string = "danger"
# Buffer.add_string buf "osite";;
- : unit = ()
# Buffer.contents buf;;
- : string = "dangerosite"
# Buffer.add_char buf 's';;
- : unit = ()
# Buffer.contents buf;;
- : string = "dangerosites"
```

II. Généralités sur les algorithmes de compression de données

II. 1. Objectifs et contexte

Définition 1 (Algorithme de compression)

> Tout algorithme de compression vise à transformer une donnée numérique originale, vue comme une série d'octets formatée, en une nouvelle série d'octets plus courte, appelée donnée compressée, occupant moins de place sur le disque dur, mais contenant suffisamment d'information pour permettre de retrouver, totalement ou partiellement, les informations contenues dans la donnée originale.

Un algorithme de compression peut travailler sur des données stockées sur le disque dur, dans un fichier, ou sur des flux de données issues d'un périphérique, typiquement un flux de données issues de la carte réseau.

Il existe des algorithmes de compression :

sans perte, qui garantissent la préservation de la totalité de l'information lors du processus de compression ; compresser la donnée puis à nouveau décompresser permet de retrouver exactement la donnée originale, sans aucune perte d'information ;

avec perte, qui autorisent une perte d'information lors du processus de compression, information qui sera définitivement perdue : il ne sera pas possible de reconstruire l'information originale dans sa totalité à partir de la donnée compressée

En MPII, on ne s'intéresse qu'aux algorithme de compression sans perte.
Tout algorithme de compression doit proposer deux fonctions :

Une fonction de compression, qui transforme la série d'octets originale en une série d'octets plus courte, appelée donnée compressée, contenant la même information ;

Une fonction de décompression, qui reconstitue la donnée originale à partir de la donnée compressée.

Remarque. Toute donnée numérique est une simple série de bits. Nous interpréterons dans ce TP cette série de bits en les rassemblant par blocs de 8 bits (1 octet), c'est à dire comme un texte, chaque octet étant interprété par un caractère de la table ASCII étendue.
Le fait de faire cette interprétation pour faciliter la compréhension et l'implémentation ne limite en rien la généralité des algorithmes de compression sans perte que nous allons implémenter car toute série de bits, pourvu que le nombre de bits soit un multiple de 8, peut être interprétée, comme un texte. Les algorithmes de compression sans perte que nous allons implémenter peuvent donc s'appliquer à tout type de données numérique, quitte à compléter les bits de cette donnée pour obtenir un nombre de bits qui est un multiple de 8.
Le cas des algorithmes avec perte est différent, car alors, le type de données encodée et le format vont être utilisés pour sélectionner les informations les plus pertinentes et retirer les informations jugées moins utiles.

II. 2. Taux de compression

Définition 2 (Taux de compression)

Le taux de compression est le rapport entre le nombre de bits initialement utilisés pour le stockage de la donnée originale et le nombre de bits nécessaires au stockage de la donnée compressée :

$$C = \frac{n_{\text{bits}}\,(\text{donnée originale})}{n_{\text{bits}}\,(\text{donnée compressée})}$$

Il représente le gain obtenu en terme de stockage, sous la forme d'un nombre réel indiquant le facteur de réduction par rapport à la donnée initiale. Si le facteur de compression est de 4.1, la compression a permis de réduire la taille de la donnée initiale d'un facteur 4.1

On parle parfois également en terme d'économie d'espace :

$$E = 1 - \frac{1}{C} = \frac{n_{\text{bits}}\,(\text{donnée originale}) - n_{\text{bits}}\,(\text{donnée compressée})}{n_{\text{bits}}\,(\text{donnée originale})}$$

Cette quantité est le taux d'évolution (en valeur absolue) de l'espace mémoire occupé entre la donnée originale et la donnée compressée.

III. Algorithme de compression de Huffman

Publié pour la première fois en 1952 [1], le codage de Huffman est un algorithme de compression sans perte de données.

III. 1. Principe de l'algorithme

L'idée de l'algorithme de Huffman est simple et intuitive : chaque caractère (motif de 8 bits) est associé à une nouvelle série de bits appelée **code**. Chaque caractère est associé à un unique code, chaque code existant n'est associé qu'à un seul caractère.
Pour réduire le nombre de bits total nécessaire pour stocker la donnée, l'astuce consiste à faire varier le nombre de bits d'un code en fonction de la fréquence d'apparition, dans la donnée originale, du caractère auquel ce code est associé : plus le caractère est fréquent dans la donnée originale, plus le nombre de bits de son code devra être petit. De la sorte, la taille de la donnée sera réduite, sans perte d'information.

Pour chaque nouvelle donnée à compresser, la table d'encodage, qui associe chaque caractère de la donnée à une série de bits en fonction de son nombre d'occurrences, va être différent et devra donc être recalculé.

Mais attention, si on choisit les codes sans plus de précaution, on peut se retrouver dans une situation où l'on ne sait pas décoder sans ambiguïté la donnée compressée, et donc

1. D.A. Huffman, "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., septembre 1952, pp 1098-1102

où l'on n'arrive plus à retrouver la donnée originale ! Donnons un exemple :

Exemple 1

Imaginons que l'on doive encoder le texte très court suivant : `alibaba`. La lettre `'a '` est la plus fréquente, on l'encode donc sur un unique bit `0`. La lettre `'b'` est la seconde lettre la plus fréquente, on l'encode avec un seul bit `1`. Les lettres `'l'` et `'i'` sont toutes deux présentent une seule fois, donc un peu moins fréquentes, on les encode sur 2 bits, par exemple `00` et `01`.
Voici donc notre table d'encodage

$$\texttt{'a'} \to \texttt{0},\ \texttt{'b'} \to \texttt{1},\ \texttt{'l'} \to \texttt{00},\ \texttt{'i'} \to \texttt{01}$$

et notre donnée originale, `alibaba`, qui est au départ encodée sur $7 \times 8 = 56$ bits est maintenant compressée en la série de bits : `000011010`, donc sur 9 bits. On a réduit la taille de la donnée originale d'un facteur 4 environ !
Oui, mais... impossible de retrouver le texte original à partir de cette série de bits ! Par exemple, doit-on interpréter le premier bit comme un code complet et l'associer à la lettre `'a '` ? Ou doit-on lire les deux premiers bits d'un coup et les interpréter comme le code `00` associé à la lettre `'l '` ?

Pour éviter ce problème, les codes doivent vérifier en plus la propriété suivante : **aucun code ne doit être un préfixe d'un autre code**. Autrement dit, aucun code ne doit correspondre au début d'un autre code plus long.
Il s'agit donc de trouver un algorithme qui construit la table d'encodage à partir de la donnée originale en créant des codes :

1. d'autant plus petits en nombre de bits que le caractère associé à ce code est fréquent dans la donnée originale ;
2. tels qu'aucun code n'est préfixe d'un autre code.

Pour construire une table d'encodage respectant cette propriété, l'algorithme de Huffman va créer un arbre binaire, appelé **arbre de Huffman**.

III. 2. Création de l'arbre de Huffman optimal associé à une donnée `s`

La création de la table d'encodage associée à une donnée `s` s'appuie sur la construction d'un *arbre binaire*, appelé *arbre de Huffman optimal* associé à `s`.

Définition 3 (Arbre de Huffman)

Un arbre de Huffman est un arbre binaire dont les nœuds internes ne sont pas étiquetés. Seules les feuilles sont étiquetées par l'un des caractères présents dans la donnée originale.

Définition 4 (Poids d'un arbre de Huffman)

Tout arbre de Huffman est associé à un entier appelé **poids**, qui correspond à la somme de toutes les occurrences de tous les caractères associés aux feuilles de l'arbre.

$\sum_{c \in C} \text{occ}(c)$ où C est l'ensemble des caractères présents dans les feuilles de l'arbre

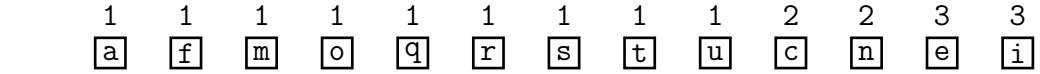
Un arbre de Huffman est optimal s'il ordonne les feuilles de sorte que plus le caractère associé à une feuille est rare, plus le niveau de profondeur de cette feuille dans l'arbre est important.

La construction d'un arbre de Huffman optimal suit une procédure particulière illustrée ci-dessous avec le texte `s = "scienceinformatique"`.

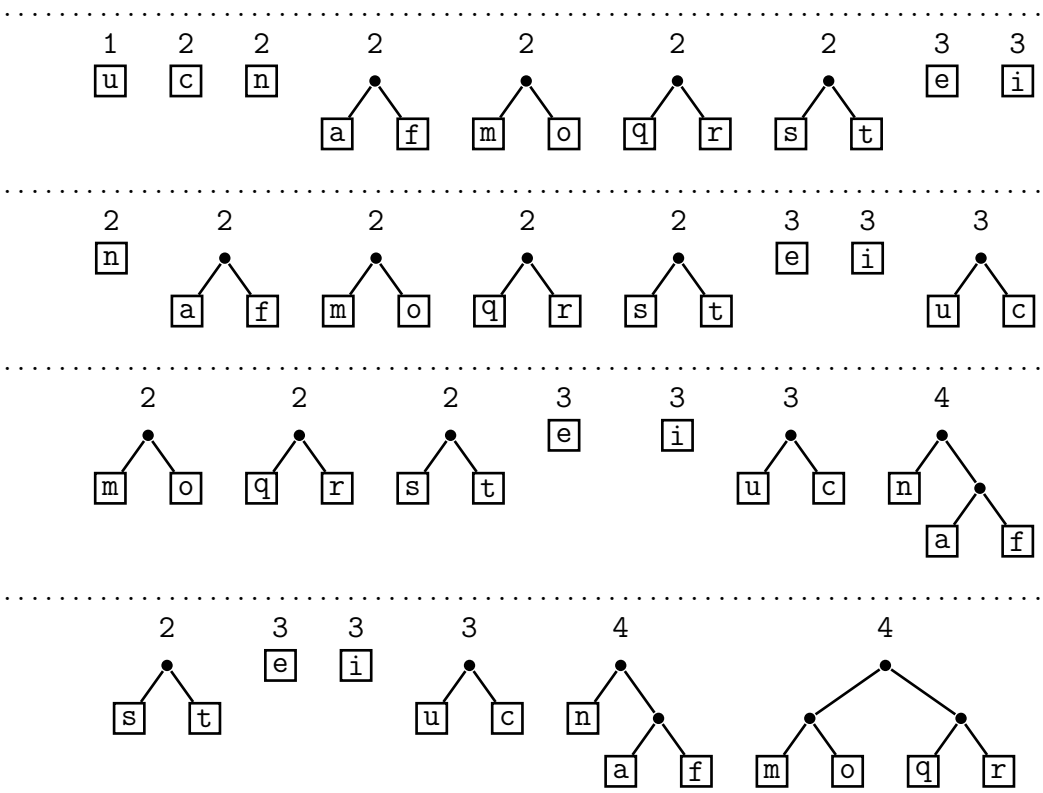
— Le *nombre d'occurrences* de chaque caractère est d'abord calculé.

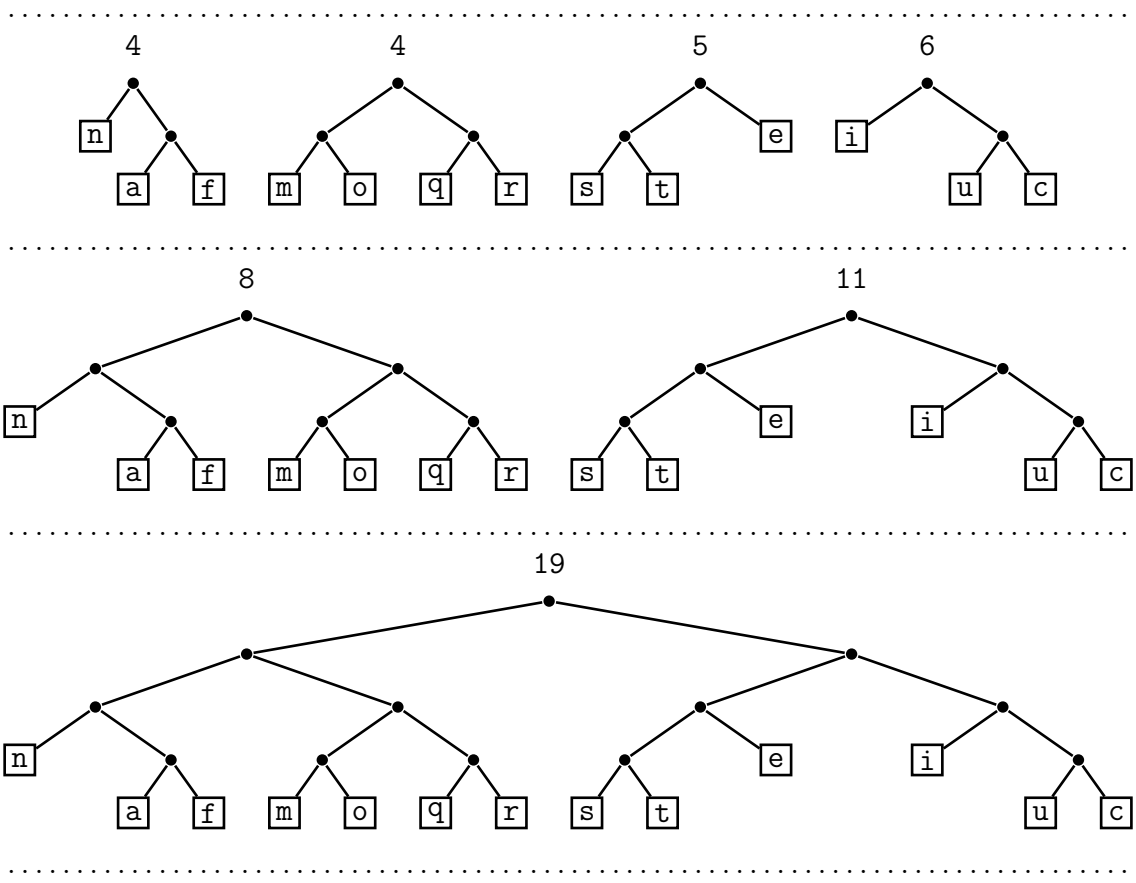
`a` → 1 `c` → 2 `e` → 3 `f` → 1 `i` → 3 `m` → 1 `n` → 2
`o` → 1 `q` → 1 `r` → 1 `s` → 1 `t` → 1 `u` → 1

— Chaque caractère `c` de `s` est ensuite associé à un arbre réduit à une feuille étiquetée par `c`, l'ensemble de ces arbres formant une *forêt*.



— Tant que la forêt contient au moins deux arbres, les deux arbres de plus petits poids sont fusionnés pour construire un arbre dont le poids est la somme des poids de ses deux sous-arbres. Les schémas suivants illustrent la construction progressive de l'arbre de Huffman optimal pour l'exemple `s = "scienceinformatique"`.





On remarque que les feuilles correspondant aux lettres les moins fréquentes sont situés sur les niveaux de profondeur les plus élevées, et les plus fréquentes au contraire sur les niveaux les moins élevés.

Remarque (Non unicité de l'arbre optimal). Il n'y a pas unicité de l'arbre de Huffman optimal associé à une donnée `s`. A la troisième étape de la création, on aurait pu choisir un autre arbre que celui constitué de la seule feuille `n`, car il y a d'autres arbres de poids 2 que nous aurions pu choisir à la place de celui-ci.

III. 3. Création de la table d'encodage à partir de l'arbre de Huffman

Pour créer la table d'encodage, on parcourt l'arbre de Huffman. Chaque code est construit par concaténation, en concaténant un 0 pour une branche de gauche, et un 1 pour une branche de droite. Lorsque l'on atteint une feuille associée à un caractère, cela signifie que le code de ce caractère a été reconstitué et qu'il peut être stocké dans la table. L'exemple donne ainsi les codages suivants.

`a` → 0010	`c` → 1111	`e` → 101	`f` → 0011	`i` → 110	`m` → 0100	`n` → 000
`o` → 0101	`q` → 0110	`r` → 0111	`s` → 1000	`t` → 1001	`u` → 1110	

III. 4. Fonction d'encodage et de décodage

III. 4. a. Compression

Une fois la table d'encodage établie, chaque caractère de la donnée originale est transformé, dans la donnée compressée, en son code. Dans notre exemple, la donnée `scienceinformatique` est transformée en le mot binaire suivant, de longueur 68 :

`10001111110101000111110111000000110101011101000010100111001101110101`

En pratique, l'**arbre de Huffman optimal ou la table d'encodage** est incorporée dans le fichier compressé de manière à permettre le décodage. Bien que cela augmente la taille du fichier, le bénéfice reste significatif pour de grandes données.
En codant chaque caractère sur 8 bits, $19 \times 8 = 152$ bits auraient été nécessaire. Le codage de Huffman en nécessite 68, auxquels il convient d'ajouter le codage de l'arbre. Toutefois, avec un texte suffisamment long, le gain reste important.

III. 4. b. Décompression

Le décodage de la donnée compressée s'effectue en « filtrant » (en tamisant) la série de bits du message compressé à travers l'arbre de Huffman. Chaque bit permet de prendre soit à gauche (s'il s'agit d'un 0) soit à droite (s'il s'agit d'un 1) dans l'arbre de Huffman. Si on aboutit à une feuille, c'est que l'on a trouvé un code complet et on l'associe donc au caractère étiquetant cette feuille.
Dans exemple ci-dessus, le mot débute par un `1` qui n'est le code d'aucun caractère. En lisant les caractères suivants du mot, le parcours de l'arbre aboutit à la feuille `s`, le mot lu étant `1011`. Ajoutons qu'il n'existe aucune lettre codée par le mot binaire `10110` ou `10111`. `1011` n'est préfixe d'aucun autre code. Ce qui permet la reconstruction non ambiguë du texte initial.

III. 5. Analyse théorique

III. 5. a. Optimalité de l'encodage de Huffman parmi tous les encodages préfixes

Théorème 1

L'arbre de Huffman est le meilleur que l'on puisse construire pour un code préfixe.
Soit n la taille du texte à compresser et n_i le nombre d'occurrences du caractères c_i. On note $f_i = \dfrac{n_i}{N}$ la fréquence du caractère c_i. Parmi tous les arbres binaires préfixes t possibles, un arbre binaire à n feuilles construit par l'algorithme de Huffman minimise la quantité :

$$S(t) = \sum_{i=0}^{n-1} f_i \times d_i(t)$$

où $d_i(t)$ est la profondeur de la feuille associée au caractère c_i dans l'arbre, c'est-à-dire la longueur du code du caractère c_i dans le texte compressé.

Démonstration

Montrons-le par récurrence sur le nombre de feuilles de l'arbre n (qui est égal au nombre de caractères différents présents dans le texte). On note la propriété $\mathcal{P}(n)$ suivante :

$\mathcal{P}(n)$: Un [a] arbre binaire $t_{\texttt{huffman}}$ à $n \geq 1$ feuilles construit par l'algorithme minimise la quantité :

$$S(t) = \sum_{i=0}^{n-1} f_i \times d_i(t)$$

c'est-à-dire :

$$S(t_{\texttt{huffman}}) = \min_{t \text{arbre préfixe à } n \text{ feuilles}} (S(t))$$

Initialisation : Pour $n = 1$, la propriété est triviale car on ne peut construire qu'un seul arbre, réduit à une feuille étiquetée par l'unique caractère présent dans le texte.

Hérédité : Soit $n \geq 2$ Supposons la propriété $\mathcal{P}(n-1)$ pour $(n-1)$ feuilles et montrons par l'absurde que $\mathcal{P}(n)$ est vraie.

Supposons donc que $\mathcal{P}(n)$ est fausse : il existe donc un arbre t, différent de n'importe quel arbre de Huffman $t_{\texttt{huffman}}$ construit avec l'algorithme, et tel que $S(t) < S(t_{\texttt{huffman}})$. Soit $t_{\texttt{huffman}}$ n'importe lequel de ces arbres possibles. Il a au moins 2 feuilles car $n \geq 2$. Du fait du processus de construction de l'algorithme, on peut toujours trouver deux feuilles étiquetées par deux caractères c_1 et c_2 appartenant à un même sous-arbre. Imaginons que l'on modifie l'arbre $t_{\texttt{huffman}}$ en remplaçant le sous-arbre constitué par ces deux feuilles par une unique feuille étiqueté par un unique caractère bidon noté c_{bidon}, non déjà présent dans la chaîne. On obtient un nouvel arbre de `Huffman` t'_{huffman} à (n-1) feuilles. Parallèlement, on doit être cohérent et remplacer dans la chaîne de caractère s toutes les occurrences de c_1 et c_2 par c_{bidon}, ce qui nous donne une nouvelle chaîne s'. Ainsi, le nouvel arbre obtenu t'_{huffman} est bien un arbre de Huffman que l'on aurait pu obtenir avec l'algorithme sur la chaîne s'.

On fait la même chose sur t en fusionnant les deux feuilles correspondant aux caractères c_1 et c_2, quitte à déplacer la feuille de hauteur plus importante vers la feuille de hauteur plus faible. Le déplacement éventuel d'une feuille, si les deux feuilles ne sont pas à côté, préserve l'optimalité de t car cela ne peut que faire diminuer la taille de l'un des codes d_i. Après la fusion, on obtient un arbre préfixe t' optimal à (n-1) feuilles associé à la chaîne s'.

On a donc un arbre t'_{huffman} à $n-1$ feuilles (on en a retiré deux, que l'on a remplacé par une unique feuille), qui peut être obtenu par l'algorithme de construction ci-dessous appliqué à la chaîne s', mais qui n'est pas optimal pour cette chaîne s' puisque l'on a réussi à construire un arbre optimal t' meilleur. On a donc une contradiction et on en déduit que $\mathcal{P}(n)$ est vraie

Conclusion : Par principe de récurrence, la propriété est vraie pour n'importe quel nombre de feuilles n

a. On a vu qu'il n'y avait pas unicité

III. 5. b. Complexité des fonction de compression et de décompression

Cf TP

III. 6. Améliorations de l'algorithme de Huffman

Dans l'algorithme de Huffman tel que nous l'avons présenté, la donnée est d'abord lue, de manière à calculer les occurrences de chaque octet, puis l'arbre est construit à partir des poids de chaque octet. Cet arbre restera le même jusqu'à la fin de la compression de cette donnée. Comme l'arbre de Huffman calculé est optimisée pour cette donnée spécifique, la compression est parfaitement optimael pour la donnée traitée, mais il sera nécessaire d'inclure les données de l'arbre dans la donnée compressée, pour permettre le décodage, ce qui pénalisera généralement le gain obtenu et va à l'encontre de l'objectif de compression des données.
Il existe plusieurs méthodes pour éviter ce stockage supplémentaire.

III. 6. a. Algorithme de Huffman statique

Si l'on dispose de **connaissances a priori sur le type de données que l'on souhaite compresser**, on peut se contenter d'un arbre de Huffman générique, standard, qui est supposé bien adapté à ce type de données, mais pas spécifiquement pour la donnée en question. Chaque octet est alors associé à un code prédéfini, choisi à partir de connaissances statistiques que l'on possède a priori sur la donnée à compresser.
Par exemple, si l'on sait que la donnée à compresser est un texte en français, où la fréquence d'apparition du `'e'` est élevée, on associera un code très court à l'octet représenté par un `'e'` en ASCII, un peu comme dans le code Morse.
L'avantage de cette méthode est que l'arbre n'a pas besoin d'être transmis avec la donnée compressée si l'arbre générique utilisé est connu du logiciel chargé de décodé la donnée. Cette méthode créera une compression peut-être moins optimisée pour la donnée spécifique que l'on traite, mais évite la transmission de l'arbre de Huffman. Toutefois, elle suppose que l'on possède des connaissances a priori sur les données traitées (par exemple qu'il s'agit d'un texte en français)

III. 6. b. Méthode adaptative

C'est la méthode qui offre a priori les meilleurs taux de compression. Elle utilise un arbre construit a priori (et ainsi non transmis) qui sera ensuite modifié de manière dynamique au fur et à mesure de la compression du flux de bits lu à partir de la donnée originale.
On part donc d'un arbre générique, qui va se modifier au fur et à mesure de la lecture de la donnée à compresser. On stockera donc non pas l'arbre de Huffman dans son intégralité, mais seulement les modifications par rapport à un arbre générique. Comme l'arbre de Huffman est adapté aux données et que l'on ne transmet que les modifications de l'arbre a priori, et non l'arbre en entier, la compression est toujours performante. De plus, il n'est pas nécessaire que le fichier soit connu avant de compresser. En particulier, l'algorithme est capable de travailler sur des flux de données (streaming), car il n'est pas nécessaire de connaître les symboles à venir, l'arbre de Huffman étant adapté « à la volée ». Cette méthode adaptative est cependant plus complexe et nécessite un temps d'exécution plus long.