TD XLV

CODE DE HUFFMAN

Définitions

- Si $\mathcal{A}$ est un ensemble fini (dit *alphabet*), on appelle ensemble des mots sur $\mathcal{A}$ et l'on note $\mathcal{A}^*$ l'ensemble des suites finies d'éléments de $\mathcal{A}$. Si $a_1, \dots, a_n \in \mathcal{A}$, l'élément $(a_1, \dots, a_n)$ de $\mathcal{A}^*$ sera simplement noté $a_1 a_2 \dots a_n$.
- On notera $|\mathcal{A}|$ le cardinal de $\mathcal{A}$, que l'on supposera systématiquement supérieur ou égal à 2.
- Si $u = a_1 \dots a_n$, où $a_1, \dots, a_n \in \mathcal{A}$, la longueur n de u sera notée $|u|$.
- Pour $a \in \mathcal{A}$ et $u \in \mathcal{A}^*$, on notera $|u|_a := \mathrm{Card}\{i \in [\![1 \dots |u|]\!] \mid u_i = a\}$. Autrement dit, $|u|_a$ désigne le nombre d'occurrences de la lettre a dans le mot u.
- Si $a_1, \dots, a_n, b_1, \dots, b_p \in \mathcal{A}$ et si l'on a $u = a_1 \dots a_n$ et $v = b_1 \dots b_p$, on notera $u \cdot v$ le mot $a_1 \dots a_n b_1 \dots b_p$, appelé *concaténation* de u et de v.
  On notera souvent uv pour $u \cdot v$.
- On note ε l'unique élément de $\mathcal{A}^*$ de longueur 0 (mot vide). Pour tout u, on a $u \cdot \varepsilon = \varepsilon \cdot u = u$.
- Si u et v sont des éléments de $\mathcal{A}^*$, on dit que u est un *préfixe* de v si $v = u \cdot w$ où $w \in \mathcal{A}^*$. Si $w \neq \varepsilon$, on dit que u est un *préfixe strict* de v.
- On appelle *code binaire* sur $\mathcal{A}$ une application f injective de $\mathcal{A}$ dans $\{0,1\}^* \setminus \{\varepsilon\}$: à chaque lettre de $\mathcal{A}$, on associe une suite finie (non vide) de 0 et de 1.
  Tous les codes considérés dans le sujet seront des codes binaires (et ce ne sera pas précisé à chaque fois).
- Si f est un code binaire sur $\mathcal{A}$, son *extension* $\overline{f}$ (que l'on notera souvent f pour alléger) est l'application :

$$\begin{array}{rccc} \overline{f} : & \mathcal{A}^* & \to & \{0,1\}^* \\ & a_1 \dots a_n & \mapsto & f(a_1) \cdot \dots \cdot f(a_n) \end{array}$$

  Autrement dit, le codage d'un mot est obtenu en concaténant les codages des caractères qui le composent.
- Un code binaire est dit *uniquement déchiffrable* si son extension est injective, *ambigu* sinon.
- Un code binaire f est dit *préfixe*[1] s'il n'existe pas de couple (a, b) d'éléments de $\mathcal{A}$ tels que $a \neq b$ et $f(a)$ soit un préfixe de $f(b)$.
- Un code binaire f est dit *à longueur fixe* si tous les $f(a)$ pour $a \in \mathcal{A}$ sont de même longueur, *à longueur variable* sinon.

1 Exemples et premières propriétés

► **Question 1** On considère dans cette question l'alphabet $\mathcal{A} = \{a, b, c\}$ et le code f défini par $f(a) = 01$, $f(b) = 010$ et $f(c) = 1$. Calculer $\overline{f}(abc)$ et $\overline{f}(bca)$. Le code f est-il préfixe ? uniquement déchiffrable ?

► **Question 2** Donner un exemple de code non préfixe uniquement déchiffrable. On justifiera (brièvement) le caractère uniquement déchiffrable du code.

► **Question 3** Soient $\mathcal{A}$ un alphabet et $u, u', v, v' \in \mathcal{A}^*$, on suppose que $uu' = vv'$. Montrer que u est un préfixe de v ou v est un préfixe de u.

► **Question 4** Montrer que tout code préfixe est uniquement déchiffrable.

1. on dit aussi *sans préfixe*, ce qui est quelque part plus logique...

2 Arbre d'un code préfixe

2.1 Arbre binaire associé à un code préfixe

Dans cette partie (et uniquement dans cette partie), on considère des arbres binaires dont :

- chaque feuille est étiquetée par un caractère (type `char` en OCaml) ;
- chaque nœud interne a soit un, soit deux fils, et ne porte pas d'étiquette.

On utilise le type OCaml suivant :

```
type arbre =
  | Vide
  | Feuille of char
  | Noeud of arbre * arbre
```

▶ **Question 5** Le type OCaml ci-dessus permet d'avoir des nœuds de la forme `Noeud (Vide, Vide)` (nœud interne n'ayant aucun fils) qui ne sont pas autorisés par la définition. Écrire une fonction `bien_forme : arbre -> bool`, qui renvoie `true` si et seulement si l'arbre reçu en argument ne comporte aucun nœud de ce type.

Quand on représentera graphiquement un tel arbre, on omettra les fils `Vide` :

```
let t1 =
  Noeud (
      Noeud (
          Feuille 'a',
          Noeud (Feuille 'b', Vide)),
      Noeud (
          Noeud (Vide, Feuille 'c'),
          Vide))
```

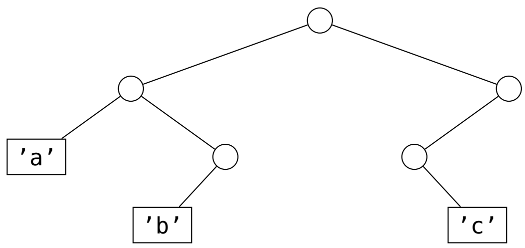


Figure XLV.1 – Définition en OCaml et représentation graphique de l'arbre t_1.

Dans un arbre t, on définit l'*adresse* $\mathrm{add}(x)$ d'un nœud x (interne on non) de la manière suivante :

- l'adresse de la racine est ε (le mot vide) ;
- si x est le fils gauche de y, alors $\mathrm{add}(x) = \mathrm{add}(y)0$;
- si x est le fils droit de y, alors $\mathrm{add}(x) = \mathrm{add}(y)1$.

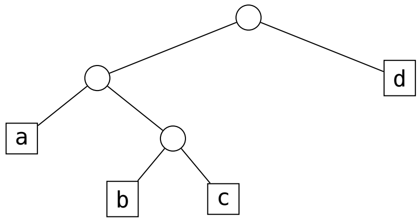
Soit $\mathcal{A}$ un alphabet de cardinal n. À un arbre binaire t ayant exactement n feuilles non vides étiquetées par les n lettres de $\mathcal{A}$, on associe un code préfixe f_t de la façon suivante :

$$\text{pour toute lettre } a \in \mathcal{A},\ f_t(a) \text{ est l'adresse de l'unique feuille de } t \text{ étiquetée par } a.$$

En reprenant l'arbre t_1 donné plus haut, on obtient alors :

Lettre	**Code**
a	00
b	010
c	101

▶ **Question 6** Déterminer le code associé à l'arbre suivant :



Inversement, tout code préfixe sur $\mathcal{A}$ peut être représenté par un arbre dont les feuilles sont exactement les lettres de $\mathcal{A}$.

▶ **Question 7** Dessiner l'arbre associé au code suivant :

Lettre	Code
a	010
b	011
c	001
d	10
e	11

2.2 Poids d'un code préfixe

Considérons un mot $s = s_1 \dots s_n$ sur un alphabet $\mathcal{A}$; **on supposera toujours que toutes les lettres de $\mathcal{A}$ apparaissent au moins une fois dans** s (ou, ce qui revient au même, que l'on restreint $\mathcal{A}$ pour ne garder que les lettres apparaissant dans s). Ce mot correspond en fait à la totalité du texte à traiter : pour nous, les espaces et les retours à la ligne sont des caractères comme les autres. On cherche à compresser ce texte en trouvant un code préfixe f pour lequel l'image de s peut être stockée sur un petit nombre de bits. On définit donc le *poids* d'un code préfixe f, que l'on note $w_s(f)$, comme la longueur totale de l'image du mot s par le code :

$$\begin{aligned} w_s(f) &:= |f(s)| \\ &= \sum_{i=1}^{n} |f(s_i)| \\ &= \sum_{a \in \mathcal{A}} |s|_a \, |f(a)| \end{aligned}$$

Un code préfixe f est dit *optimal pour un mot* s si $w_s(f)$ est minimal parmi tous les codes préfixes. On notera $\mathrm{opt}(s)$ le poids d'un code préfixe optimal pour s.

▶ **Question 8** On définit $\mathrm{opt}_{\mathrm{fixe}}(s)$ comme le poids minimal d'un code préfixe *à longueur fixe* pour le mot s. Exprimer $\mathrm{opt}_{\mathrm{fixe}}(s)$ en fonction de $|s|$ et de $|\mathcal{A}|$.

▶ **Question 9** Donner un exemple de mot s sur l'alphabet $\{a, b, c, d\}$ pour lequel on a $\mathrm{opt}(s) < \mathrm{opt}_{\mathrm{fixe}}(s)$ (on justifiera cette inégalité).

▶ **Question 10** Montrer que l'arbre associé à un code préfixe optimal ne contient aucun nœud n'ayant qu'un seul fils (non vide).

Comme on s'intéresse dans la suite à la construction d'un code optimal, on peut donc simplifier le type de nos arbres pour se limiter aux arbres binaires entiers (où chaque nœud interne a exactement deux fils). Le type obtenu, que nous utiliserons dans toute la suite du problème, est alors :

```
type arbre_code =
  | F of char
  | N of arbre_code * arbre_code
```

2.3 Fonctions de codage et décodage

On choisit les types suivants pour les différents objets :

- le texte que l'on souhaite compresser (le mot s) est représenté par une chaîne de caractères (type `string`);
- l'alphabet $\mathcal{A}$ est constitué des caractères ASCII (type `char`) apparaissant au moins une fois dans s ;
- le texte compressé (c'est-à-dire le résultat $f(s)$ de l'application du code au texte s de départ) est une suite de zéros et de uns ; il sera représenté comme une liste de booléens, où `false` correspond à 0 et `true` à 1 :

```
type bitstream = bool list
```

- le code f aura deux représentations :
  — une de type `arbre_code` qui sera utilisée pendant le décodage (et aussi la construction du code en fin de problème)
  — une autre utilisée pour l'encodage, détaillée dans la partie 2.3.b.

2.3.a Décodage

▸ **Question 11** Écrire une fonction `decode_caractere : arbre_code -> bitstream -> char * bitstream` prenant en entrée un code préfixe f sous forme d'arbre et le codage $u = f(s)$ d'un certain mot s, et renvoyant le couple (a, u') tel que $u = f(a) \cdot u'$. Autrement dit, cette fonction doit renvoyer le premier caractère du texte décodé et le reste du texte à décoder.

▸ **Question 12** On donne la fonction suivante pour convertir une `char list` en `string` :

```
let string_of_char_list u = String.of_seq (List.to_seq u)
```

Écrire une fonction `decode_texte (f : arbre_code) (u : bitstream) : string` prenant un code préfixe f sous forme d'arbre et le codage $u = f(s)$ d'un certain mot s, et renvoyant s.

2.3.b Encodage

Pour réaliser le codage, un arbre n'est pas très pratique : on préfère avoir un tableau `t` permettant d'obtenir directement le code associé à un caractère. On définit donc :

```
type table_code = bitstream array
```

Une `table_code` sera toujours de longueur 256, et contiendra dans sa case `i` le code du caractère `char_of_int i` ; pour les caractères n'apparaissant pas dans le texte (et n'ayant donc pas de code associé), la case contiendra la liste vide.

Par exemple, le code

Lettre	Code
a	010
b	011
c	00
d	1

serait représenté par un `t : table_code` avec :

- `t.(0) = ... = t.(96) = t.(101) = ... = t.(255) = []` (car tous ces caractères n'ont pas de code) ;
- `t.(97) = [false; true; false]` (car `int_of_char 'a' = 97`) ;
- `t.(98) = [false; true; true]` (car `int_of_char 'b' = 98`) ;
- `t.(99) = [false; false]` et `t.(100) = [true]`, de même.

▸ **Question 13** Écrire une fonction `cree_table : arbre_code -> table_code` permettant d'obtenir la représentation d'un code sous forme de table à partir de sa représentation sous forme d'arbre.

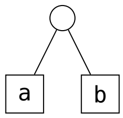
▸ **Question 14** Écrire la fonction `encode (t : table_code) (s : string) : bitstream`, qui prend en entrées la table `t` représentant un code préfixe f et le texte `s` et renvoie $f(s)$ sous la forme d'une liste de booléens.

3 Algorithme de Huffman

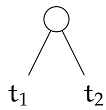
3.1 Principe de l'algorithme

L'algorithme de Huffman permet de construire un code préfixe optimal pour un mot s donné.

- On calcule $|s|_a$ pour chaque $a \in \mathcal{A}$. On crée une feuille étiquetée a pour chaque a apparaissant dans s, et on crée une liste $q = \left[\left(\boxed{a_1}, |s|_{a_1}\right), \left(\boxed{a_2}, |s|_{a_2}\right), \dots, \left(\boxed{a_p}, |s|_{a_p}\right)\right]$.
- On détermine les deux feuilles a et b ayant les plus petits nombres d'occurrences (*i.e.* les deux couples ayant les plus petites deuxièmes composantes), on les sort de la liste et l'on met à la place le couple $(t, |s|_a + |s|_b)$, où t est l'arbre

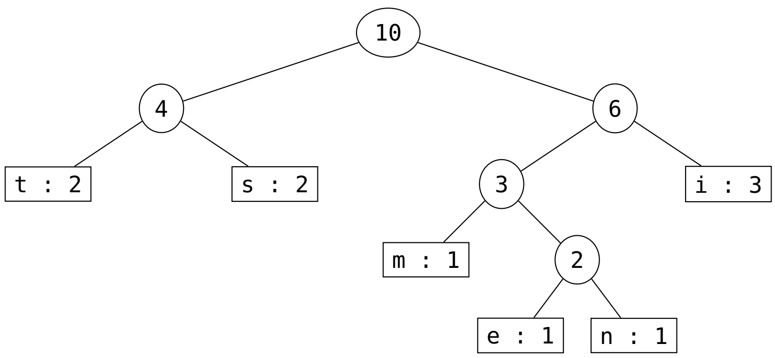


- On recommence l'étape précédente, en prenant les deux couples (t_1, n_1) et (t_2, n_2) ayant les plus petites deuxièmes composantes, et en les remplaçant par le couple $(t, n_1 + n_2)$ où t est l'arbre



  Ici, t_1 et t_2 ne sont pas nécessairement des feuilles.
- On continue jusqu'à ce qu'il n'y ait plus qu'un arbre dans la liste : cet arbre est le code de Huffman associé à s.

▸ **Question 15** Vérifier que, appliqué au mot `"intimistes"`, l'algorithme de Huffman produit (ou plutôt peut produire, suivant comment l'on tranche en cas d'égalité) l'arbre :



Les étiquettes entières des nœuds et des feuilles sont « virtuelles » : elles ont servi à la construction mais ne sont en fait pas stockées dans l'arbre.

▸ **Question 16** Pour l'exemple ci-dessus, calculer :

- le nombre de bits qu'occupe la chaîne de départ ;
- le poids qu'aurait un code à longueur fixe (où l'on restreint l'alphabet aux caractères effectivement présents) ;
- le poids du code de Huffman.

Quelle caractéristique du texte initial le codage de Huffman exploite-t-il pour obtenir un poids inférieur à celui d'un code à longueur fixe ?

▶ **Question 17** On considère un mot s sur un alphabet $\mathcal{A} = \{a_0, \dots, a_{n-1}\}$ vérifiant $|s|_{a_i} = 2^i$ pour $0 \leqslant i < n$. Donner (en justifiant) la forme d'un arbre de Huffman possible pour s et montrer que son poids vaut $2^{n+1} - n - 3$.

▶ **Question 18** On définit le *facteur de compression* d'un code f pour le mot s comme le quotient $\frac{\mathrm{opt}_{\mathrm{fixe}}(s)}{w_s(f)}$. En reprenant le mot s de la question précédente, déterminer un équivalent simple de ce facteur de compression pour le code de Huffman quand n tend vers $+\infty$.

3.2 Construction de l'arbre

▶ **Question 19** Écrire une fonction `occurrences (s : string) : int array`. Cette fonction prend en entrée une chaîne s et renvoie un tableau t de taille 256 tel que $t.(i)$ contienne le nombre d'occurrences du caractère dont le numéro ASCII est i (c'est-à-dire de `char_of_int i`) dans la chaîne s. On demande une complexité en $\mathcal{O}(|s|)$.

```
# let t = occurrences
"So we beat on, boats against the current, borne back ceaselessly into the
↪  past.";;
(* OCaml affiche le tableau de taille 256, omis ici... *)
# t.(97);;
- : int = 7
(* int_of_char 'a' vaut 97, et il y a sept 'a' dans la chaine donnée en argument.
↪  *)
```

▶ **Question 20** Écrire une fonction `foret (s : string) : (arbre_code * int) list` qui prend une chaîne de caractères et renvoie la liste des `(Feuille c, f)`, où le caractère c apparaît f fois dans s. Les caractères n'ayant aucune occurrence dans s seront omis. L'ordre des éléments de la liste n'a pas d'importance.

```
utop[13]> foret "inimity";;
- : (arbre_code * int) list =
[(F 'i', 3); (F 'm', 1); (F 'n', 1); (F 't', 1); (F 'y', 1)]
```

▶ **Question 21** Écrire une fonction `huffman (s : string) : arbre_code` qui renvoie un arbre de Huffman associé à la chaîne s.

Remarque

On pourra utiliser une structure de données (que vous devriez bien connaître, et pour laquelle vous devriez pouvoir copier-coller du code) adaptée à cette construction.

```
utop[11]> huffman "des dodos font dodo";;
- : arbre_code =
N (N (N (F 's', N (F 'n', F 'e')), N (N (F 'f', F 't'), F ' ')),
 N (F 'd', F 'o'))
```

▶ **Question 22** Écrire une fonction `compresse : string -> (arbre_code * bitstream)` qui prend en entrée une chaîne et renvoie le code de Huffman correspondant, sous forme d'arbre, et le texte compressé sous forme de flux binaire.

3.3 Optimalité

Pour démontrer le caractère optimal du code de Huffman, nous allons modifier légèrement nos notations. On remarque que le code de Huffman associé à un mot s ne dépend pas de l'ordre des lettres dans s, et qu'il en est de même pour $\mathrm{opt}(s)$: pour un code f, on aura toujours $w_f(\mathrm{edredon}) = w_f(\mathrm{ddeeorn})$.

On laisse donc de côté la notion de mot pour se concentrer sur celle d'alphabet, que l'on étend pour inclure les fréquences d'apparition des différentes lettres :

- dans la suite, on appellera *alphabet* un ensemble fini de couples $\mathcal{A} = \{(a_1, n_1), \dots, (a_p, n_p)\}$ où les a_i sont des lettres (deux à deux distinctes) et les n_i des entiers vérifiant $1 \leqslant n_1 \leqslant n_2 \leqslant \dots \leqslant n_p$ (n_i représente le nombre d'occurrences de a_i dans le mot sous-jacent) ;
- on définit $h(\mathcal{A})$ comme l'arbre de Huffman associé à un mot constitué de n_1 lettres a_1, ..., n_p lettres a_p et $w_h(\mathcal{A})$ son poids ;
- on définit également $\mathtt{opt}(\mathcal{A})$ comme le poids minimal d'un code pour ce même mot ;
- l'objectif est donc de montrer que $\mathtt{opt}(\mathcal{A}) = w_h(\mathcal{A})$.

On rappelle qu'on suppose systématiquement $|\mathcal{A}| \geqslant 2$.

► **Question 23** Montrer qu'on peut toujours trouver un code optimal pour $\mathcal{A}$ dans lequel la feuille étiquetée a_1 a pour sœur la feuille étiquetée a_2.
On rappelle que l'on a numéroté les lettres de manière à avoir $n_1 \leqslant n_2 \leqslant \dots \leqslant n_p$.

► **Question 24** Pour un alphabet $\mathcal{A}$ vérifiant $|\mathcal{A}| \geqslant 3$, on définit $\mathcal{A}' = \{(b, n_1+n_2), (a_3, n_3), \dots, (a_{|\mathcal{A}|}, n_{|\mathcal{A}|})\}$, où b est une nouvelle lettre, distincte de toutes les autres.
Montrer que $\mathtt{opt}(\mathcal{A}) \geqslant \mathtt{opt}(\mathcal{A}') + n_1 + n_2$.

► **Question 25** Montrer que le code construit par l'algorithme de Huffman est optimal.

Solutions

▶ **Question 1** $f(a)$ est un préfixe de $f(b)$, donc $\boxed{f \text{ n'est pas préfixe.}}$

De plus, $f(abc) = 010101 = f(bca)$ et $abc \neq bca$, donc $\boxed{f \text{ n'est pas uniquement déchiffrable.}}$

▶ **Question 2** Prenons $\mathcal{A} = \{a, b\}$ et f défini par $\boxed{f(a) = 0 \text{ et } f(b) = 01.}$

f n'est $\boxed{\text{pas préfixe, et est pourtant uniquement déchiffrable.}}$ En effet, chacun des 1 présents dans l'image doit nécessairement être précédé d'un 0 : on sait que ces blocs 01 correspondent à des b. Ensuite, il ne reste que des 0, donc chacun correspond à un a.

Remarque

En pratique, on programmerait comme suit (en codant les 0 et 1 par des booléens) :

```
let rec decode_exemple = function
  | [] -> []
  | false :: true :: xs -> 'b' :: decode_exemple xs
  | false :: xs -> 'a' :: decode_exemple xs
  | true :: xs -> failwith "pas l'image d'un mot"
```

▶ **Question 3** On suppose que $|u| \leqslant |v|$, et l'on prouve par récurrence sur $|u|$ que u préfixe de v :

- si $|u| = 0$, alors $u = \varepsilon$ est un préfixe de v ;
- sinon, $u = ax$ et $v = by$ avec $a, b \in \mathcal{A}$ (car $|v| \geqslant |u| \geqslant 1$).
  On a donc $axu' = byv'$, on en déduit $a = b$ et $xu' = yv'$, avec $|x| = |u| - 1$.
  En appliquant l'hypothèse de récurrence on obtient x préfixe de y et donc u préfixe de v.

Par symétrie des rôles de u et v, on conclut que $\boxed{u \text{ est un préfixe de } v \text{ ou } v \text{ est un préfixe de } u.}$

▶ **Question 4** Soient f un code préfixe sur $\mathcal{A}$, $u = a_1 \dots a_n \in \mathcal{A}^*$, $v = b_1 \dots b_p \in \mathcal{A}^*$; on suppose $f(u) = f(v)$. Montrons que $u = v$ par récurrence sur $n = |u|$:

- si $n = 0$, alors $f(u) = f(\varepsilon) = \varepsilon$, donc $f(v) = \varepsilon$ et $v = \varepsilon$.
- sinon, on a $f(a_1)f(a_2 \dots a_n) = f(b_1)f(b_2 \dots b_p)$. D'après la question précédente, on a donc $f(a_1)$ préfixe de $f(b_1)$ ou inversement ; comme f est préfixe, cela signifie que $a_1 = b_1$. On a donc $f(a_2 \dots a_n) = f(b_2 \dots b_p)$, on conclut en utilisant l'hypothèse de récurrence.

Ainsi, $\boxed{\text{tout code préfixe est uniquement déchiffrable.}}$

1 Arbre d'un code préfixe

1.1 Arbre binaire associé à un code préfixe

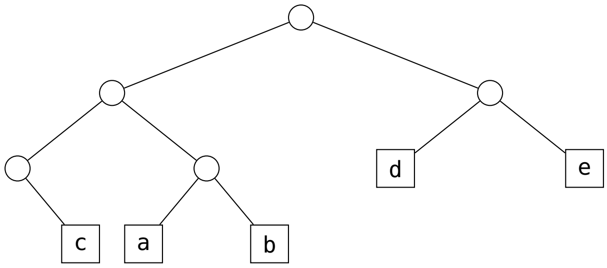
▶ **Question 5**

```
let rec bien_forme = function
  | Vide -> true
  | Feuille _ -> true
  | Noeud (Vide, Vide) -> false
  | Noeud (g, d) -> bien_forme g && bien_forme d
```

▸ Question 6

Lettre	Code
a	00
b	010
c	011
d	1

▸ Question 7



▸ Question 8 Considérons un code de longueur fixe égale à k. Son poids est $|s| \cdot k$, et minimiser ce poids revient donc à minimiser k. Or une longueur de k permet de coder un maximum de 2^k symboles différents : il faut donc prendre le plus petit entier k tel que $2^k \geqslant |\mathcal{A}|$, c'est-à-dire $k \geqslant \lceil \log_2 |\mathcal{A}| \rceil$. Finalement, $\boxed{\mathtt{opt}_{\mathtt{fixe}}(s) = \lceil \log_2 |\mathcal{A}| \rceil \cdot |s|.}$

▸ Question 9

Considérons $s = aaaabcd$, et f le code préfixe défini ci-contre.
On a $w_f(s) = 4 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 12$, donc $\mathtt{opt}(s) \leqslant 12$. Or $\mathtt{opt}_{\mathtt{fixe}}(s) = \lceil \log_2 4 \rceil \cdot 7 = 14$, donc $\boxed{\mathtt{opt}_{\mathtt{fixe}}(s) > \mathtt{opt}(s).}$

Lettre	Code
a	0
b	10
c	110
d	111

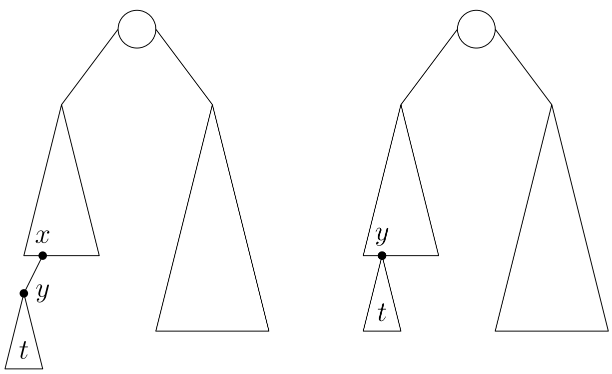
▸ Question 10

Considérons un mot s sur un alphabet $\mathcal{A}$ et l'arbre d'un code préfixe f sur $\mathcal{A}$, et supposons qu'il contienne un nœud x n'ayant qu'un fils y. Notons $\mathcal{B}$ l'ensemble des lettres de $\mathcal{A}$ dont la feuille se trouve dans le sous-arbre enraciné en x et f' le code dont l'arbre est obtenu en supprimant le nœud x et en le remplaçant par y.

- Pour $a \in \mathcal{B}$, on a $|f'(a)| = |f(a)| - 1$ puisque ces feuilles ont été remontées d'un niveau.
- Pour $a \in \mathcal{A} \setminus \mathcal{B}$, on a $|f'(a)| = |f(a)|$.

On a donc $w_{f'}(s) = w_f(s) - \sum_{a \in \mathcal{B}} |s|_a < w_f(s)$ car $\mathcal{B}$ est non vide et toutes les lettres de $\mathcal{A}$ apparaissent dans s. Donc f n'est pas optimal.

Arbre de f Arbre de f'



$\boxed{\text{Ainsi, l'arbre d'un code optimal ne contient aucun nœud n'ayant qu'un seul fils.}}$

▸ Question 11

```
let rec decode_caractere arbre u =
  match arbre, u with
  | F x, _ -> (x, u)
  | N (ga, _), false :: u' -> decode_caractere ga u'
  | N (_, dr), true :: u' -> decode_caractere dr u'
  | _ -> failwith "erreur de décodage"
```

▶ **Question 12**

```
let decode_texte (arbre : arbre_code) (flux : bitstream) : string =
  let rec aux u =
    match u with
    | [] -> []
    | _ -> let (c, u') = decode_caractere arbre u in
           c :: aux u' in
  string_of_char_list (aux flux)
```

1.1.a Codage

▶ **Question 13** `remplit_tab noeud pref` explore le sous arbre `noeud` et remplit les cases du tableau `t` correspondant aux feuilles qui y apparaissent. L'argument `pref` est le préfixe commun à tous les codes du sous-arbre (qui correspond à l'adresse de `noeud`, à l'envers).

```
let cree_table arbre =
  let t = Array.make 256 [] in
  let rec remplit_tab noeud pref =
    match noeud with
    | F c ->
        t.(int_of_char c) <- List.rev pref
    | N(ga, dr) ->
        remplit_tab ga (false :: pref);
        remplit_tab dr (true :: pref) in
  remplit_tab arbre [];
  t
```

▶ **Question 14**

```
let encode (t : table_code) (s : string) : bitstream =
  let rec encode_aux k =
    if k = String.length s then []
    else t.(int_of_char s.[k]) @ encode_aux (k + 1) in
  encode_aux 0
```

▶ **Question 15** On le vérifie...

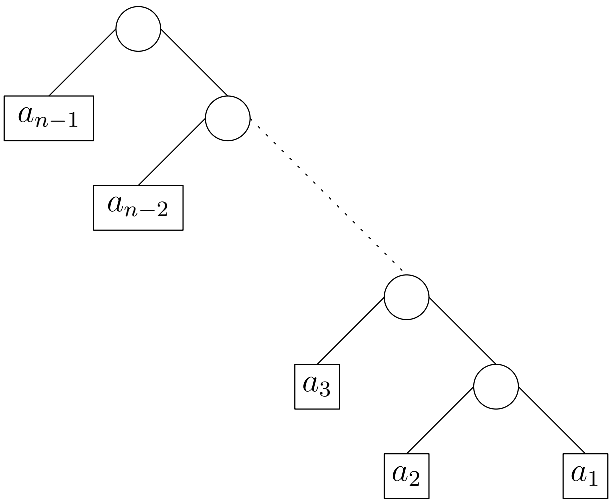
▶ **Question 16**

- Une chaîne de caractères occupe un octet par caractère[2], donc ici dix octets soit $\boxed{\text{80 bits.}}$
- $|\mathcal{A}| = 6$, donc $\lceil \log_2 |\mathcal{A}| \rceil = 3$ et $\boxed{\text{un code à longueur fixe aurait un poids de 30 bits.}}$
- Pour le code de Huffman, on obtient $2 \cdot 2 + 2 \cdot 2 + 1 \cdot 3 + 1 \cdot 4 + 1 \cdot 4 + 3 \cdot 2 = \boxed{\text{25 bits.}}$

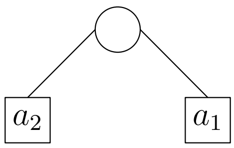
Le codage de Huffman stocke de manière plus compacte les caractères qui sont plus fréquents dans le texte : plus les fréquences d'apparition seront différentes, plus il sera efficace.

2. Plus quelques octets pour stocker, entre autres, la longueur de la chaîne, mais on va le négliger ici.

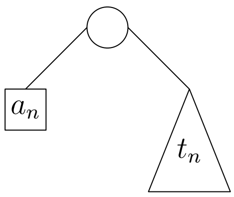
▶ **Question 17** On montre par récurrence sur $n \geqslant 2$ que la formule demandée est respectée, et que l'arbre peut être un *peigne droit* :



- Pour $n = 2$, on obtient



  et le poids vaut $2^0 \cdot 1 + 2^1 \cdot 1 = 3$. Or $2^3 - 2 - 3 = 3$, donc la propriété est initialisée.
- On suppose la propriété vérifiée pour n et l'on considère $\mathcal{A} = \{a_0, \dots, a_n\}$.
  Comme $|s|_{a_n} = 2^n > \sum_{i=0}^{n-1} |s|_{a_i} = 2^n - 1$, la feuille a_n ne sera fusionnée qu'à la dernière étape.
  Par hypothèse de récurrence, la liste q contiendra alors $\left(\boxed{a_n}, 2^n\right)$ et $(t_n, 2^n - 1)$, où t_n est l'arbre dessiné plus haut. On obtiendra alors un arbre t_{n+1} ayant la bonne forme :



  Chaque feuille a_i présente dans t_n a vu sa profondeur augmentée de 1, donc le poids de t_{n+1} vaut

$$\begin{aligned}\mathtt{poids}(t_{n+1}) &= |s|_{a_n} \cdot 1 + \mathtt{poids}(t_n) + \sum_{i=0}^{n-1} |s|_{a_i} \\ &= 2^n + 2^{n+1} - n - 3 + 2^n - 1 \\ &= 2^{n+2} - (n+1) - 3\end{aligned}$$

  ce qui achève la récurrence.

▶ **Question 18** En notant s_n le mot défini plus haut, on a $\mathtt{opt}_{\mathtt{fixe}}(s_n) = \lceil \log_2 n \rceil \cdot (2^n - 1) \sim 2^n \log_2 n$ et $w_s(f_n) = 2^{n+1} - n - 3 \sim 2^{n+1}$. Le facteur de compression est donc équivalent à $\boxed{\frac{\log_2 n}{2}.}$
Dans cet exemple (essentiellement le meilleur cas pour le code de Huffman), on arrive à utiliser une moyenne de 2 bits par caractère pour un alphabet de taille n (au lieu de $\log_2 n$ bits par caractère pour un code à longueur fixe).

▶ **Question 19**

```
let occurrences (s : string) : int array =
  let tab_freq = Array.make 256 0 in
  for k = 0 to String.length s - 1 do
    let x = int_of_char s.[k] in
    tab_freq.(x) <- tab_freq.(x) + 1
  done;
  tab_freq
```

▶ **Question 20**

```
let foret (s : string) : (arbre_code * int) list =
  let t = occurrences s in
  let rec aux k =
    if k >= Array.length t then []
    else if t.(k) > 0 then (F (char_of_int k), t.(k)) :: aux (k + 1)
    else aux (k + 1) in
  aux 0
```

▶ **Question 21** On transforme la forêt en file de priorité (en utilisant le nombre d'occurrences comme priorité). Ensuite, tant qu'elle contient au moins deux éléments, on récupère les deux arbres à fusionner, on les fusionne et on insère le résultat (avec la bonne priorité).

```
let huffman s =
  let file = PrioQ.of_list (foret s) in
  while PrioQ.length file > 1 do
    let (c, fr) = PrioQ.extract_min file in
    let (c', fr') = PrioQ.extract_min file in
    PrioQ.insert file (N(c, c'), fr + fr')
  done;
  fst (PrioQ.extract_min file)
```

▶ **Question 22** Il s'agit juste de combiner les fonctions déjà écrites.

```
let compresse (s : string) : (arbre_code * bitstream) =
  let arbre = huffman s in
  let table = cree_table arbre in
  arbre, encode table s
```

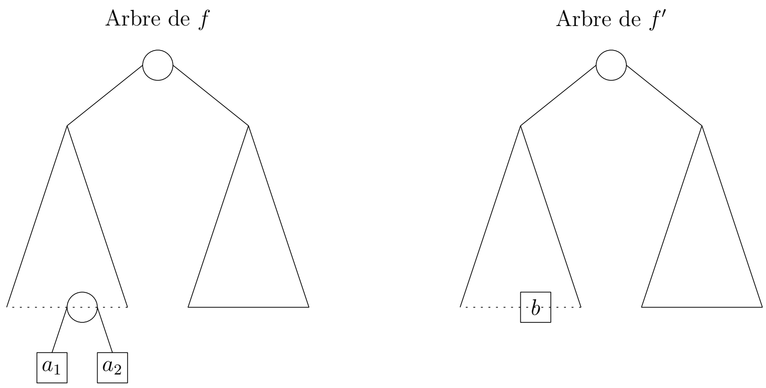
▶ **Question 23** Partons d'un arbre f optimal pour un alphabet $\mathcal{A}$ et montrons que l'on peut le transformer en un arbre g dont les feuilles a_1 et a_2 sont sœurs et qui vérifie $w_g(\mathcal{A}) \leqslant w_f(\mathcal{A})$.

- Soit a_i telle que $|f(a_i)|$ soit maximal. On échange les feuilles a_1 et a_i, on obtient un arbre f' avec :

$$\begin{aligned} w_{f'}(\mathcal{A}) - w_f(\mathcal{A}) &= n_1\left(|f'(a_1)| - |f(a_1)|\right) + n_i\left(|f'(a_i)| - |f(a_i)|\right) \\ &= n_1\left(|f(a_i)| - |f(a_1)|\right) + n_i\left(|f(a_1)| - |f(a_i)|\right) \\ &= \underbrace{(n_1 - n_i)}_{\leqslant 0}\underbrace{\left(|f(a_i)| - |f(a_1)|\right)}_{\geqslant 0} \\ &\leqslant 0 \end{aligned}$$

- D'après la question 10, on sait que a_1 n'est pas une « fille unique » dans f'. Comme c'est la feuille la plus profonde, sa sœur est forcément une feuille a_j.
- On obtient g en échangeant les feuilles a_j et a_2 de f'. Comme $n_j \geqslant n_2$, on obtient $w_g(\mathcal{A}) \leqslant w'_f(\mathcal{A}) \leqslant w_f(\mathcal{A})$ comme dans le premier point.
- Comme f était optimal, g est encore optimal.

► **Question 24** Soit f un arbre optimal pour $\mathcal{A}$ tel que les feuilles a_1 et a_2 soient sœurs. Considérons l'arbre f' dans lequel on a remplacé ces deux feuilles ainsi que leur père par une unique feuille b :



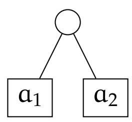
Cet arbre f' définit un code sur $\mathcal{A}'$, et son poids est

$$\begin{aligned}
w_{f'}(\mathcal{A}') &= w_f(\mathcal{A}) - n_1|f(a_1)| - n_2|f(a_2)| + n_b|f'(b)| \\
&= w_f(\mathcal{A}) - (n_1 + n_2)|f(a_1)| + (n_1 + n_2)\left(|f(a_1)| - 1\right) \\
&= w_f(\mathcal{A}) - n_1 - n_2 \\
&= \mathtt{opt}(\mathcal{A}) - n_1 - n_2
\end{aligned}$$

Or par définition $\mathtt{opt}(\mathcal{A}') \leqslant w_{f'}(\mathcal{A}')$, donc $\boxed{\mathtt{opt}(\mathcal{A}) \geqslant \mathtt{opt}(\mathcal{A}') + n_1 + n_2.}$

► **Question 25** On procède par récurrence sur le cardinal p de l'alphabet.

- Pour $p = 2$, le code de Huffman est clairement optimal.
- Soient $p \geqslant 3$, $\mathcal{A} = \{(a_1, n_1), \ldots, (a_p, n_p)\}$ et $\mathcal{A}' = \{(b, n_1+n_2), (a_3, n_3), \ldots, (a_p, n_p)\}$ comme au-dessus. On peut reformuler l'algorithme de construction du code de Huffman comme suit :
  — remplacer $\mathcal{A}$ par $\mathcal{A}'$;
  — calculer l'arbre de Huffman pour $\mathcal{A}'$;
  — remplacer dans cet arbre la feuille $\boxed{b}$ par



  On a donc (le calcul est le même qu'à la question précédente) $w_h(\mathcal{A}) = w_h(\mathcal{A}') + n_1 + n_2$.
  Or par hypothèse de récurrence on a $w_h(\mathcal{A}') = \mathtt{opt}(\mathcal{A}')$, donc $w_h(\mathcal{A}) = \mathtt{opt}(\mathcal{A}') + n_1 + n_2$.
  D'après la question précédente, cela implique $w_h(\mathcal{A}) \leqslant \mathtt{opt}(\mathcal{A})$ et donc $w_h(\mathcal{A}) = \mathtt{opt}(\mathcal{A})$.

$\boxed{\text{Le code de Huffman est donc optimal.}}$