Chapitre 11 : Algorithmique du texte

Table des matières

1 Recherche dans un texte

1.1 Introduction

1.1.1 Problème

Étant donné un texte (un fichier / une chaîne de caractères) t, et un motif (une chaîne de caractères) x, on veut trouver toutes les occurrences de x dans t (*i.e* on veut les positions (indices)).

1.1.2 Algorithme naïf

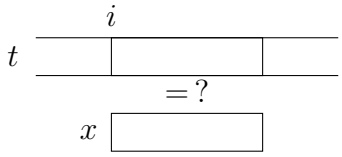
On teste toutes les positions dans t pour déterminer si ce sont des positions d'occurrences de x :

```
let recherche_naive (x : string) (t : string) : int list =
    let l = ref []in
    let n = String.length t
    and m = String.length x in

    for i = 0 to n - m do
        let j = ref 0 in
        while !j < m && x.[!j] = t.[i + !j] do
            incr j
        done;

        if !j = m then l := i::!l
    done;
    !l
```

Complexité : dans le pire cas, la boucle `while` s'arrête au dernier indice dans x (exemple : $t = a^n = \underbrace{a \cdots a}_{n \text{ fois } a}$, et $x = a^{m-1}b$) : $\mathcal{O}((n-m+1)m) = \mathcal{O}(nm)$ si m est petit devant n.

1.1.3 Remarque

Cet algorithme fait partie d'une famille d'algorithmes, dits de *fenêtre glissante* : on fait glisser une fenêtre sur le texte en notant toutes les positions auxquelles la fenêtre contient le motif.



Le côté naïf de cet algorithme vient du fait que l'on fait systématiquement glisser la fenêtre d'un rang, quel que soit son contenu. En analysant les raisons de l'échec d'une comparaison, on peut espérer décaler la fenêtre de plusieurs rangs.

1.2 Algorithme de Boyer-Moore

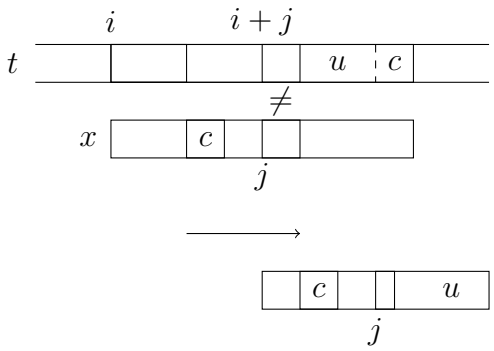
1.2.1 Introduction

L'algorithme de Boyer-Moore est un algorithme de fenêtre glissante, mais la comparaison entre le contenu de la fenêtre et le motif se fait de la droite vers la gauche, afin de repérer en cas d'échec le caractère le plus à droite qui ne correspond pas au motif. À partir de cette position qui entraîne l'échec de la comparaison, on peut calculer un décalage pour la fenêtre.

En pratique le décalage est précalculé pour toutes les positions du motif et il existe de nombreuses variantes suivant la manière dont le décalage est calculé.

1.2.2 Algorithmique de Boyer-Moore-Horspool

Dans cette variante, si la comparaison de $x_0 \dots x_{m-1}$ et $t_i \dots t_{i+m-1}$ (la fenêtre) échoue à l'indice j, *i.e* $x_{j+1} \dots x_{m-1} = t_{i+j+1} \dots t_{i+m-1}$ et $x_j \neq t_{i+j}$, on cherche à aligner t_{i+m-1} avec son occurrence la plus à droite dans x (sauf x_{m-1})



Exemple : $x = aababab$ et $t = aabbbababacaabbaba$

$aababab$
×
$aababab$
×
$aababab$
×

Algorithme : pour toute lettre a, on note $d(a)$ le décalage à effectuer pour aligner cette lettre avec don occurrence la plus à droite dans x (sauf la dernière lettre) en cas d'échec d'une comparaison avec une fenêtre dont la dernière lettre est a.

$$d(a) = \begin{cases} |u| & \text{si } u \text{ est les plus petit suffixe non vide de } x \text{ tq } au \text{ est suffixe de } x \\ |x| & \text{si } u \text{ n'existe pas, } i.e \text{ si } x \text{ ne contient pas } a, \text{ ou alors seulement en dernière position} \end{cases}$$

Pseudo-code :

```
i ← 0
Tant que i ⩽ n − m :
    j ← m − 1
    Tant que j ⩾ 0 et x_j = t_{i+j} :
```

$j \leftarrow j - 1$

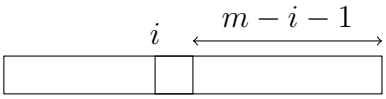
Si $j = -1$:
i est la position d'une occurrence de x
$i \leftarrow i + 1$

Sinon :
$i \leftarrow i + d(t_{i+m-1})$

Précalcul de d :

Pour toute lettre a, $d(a) \leftarrow m$

Pour i de 0 à $m - 2$:
$d(x_i) \leftarrow m - 1 - i$



Complexité :

– Précalcul : $\mathcal{O}(|A| + m)$ où A est l'alphabet, *i.e* l'ensemble des symboles possibles dans un texte.

– Algorithme : dans le pire cas, on décale toujours d'un rang (exemple : $t = a^n$ et $x = ba^{m-1}$) : même complexité que l'algorithme naïf.

En pratique, c'est plus efficace : $\mathcal{O}(n)$ en moyenne (admis).

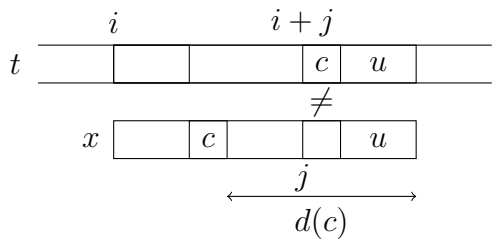
1.2.3 Algorithme de Boyer-Moore (simplifié)

L'algorithme de Boyer-Moore-Horspool ne tient pas compte de ce qu'il se passe lors de la lecture de la fenêtre, mais seulement de son dernier caractère.

Par exemple : si $x = aababab$ et la fenêtre vaut $aabcbab$, le décalage calculé vaut 2 alors que l'absence de c dans x permet un décalage de 4 rangs.

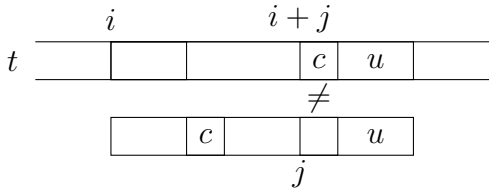
L'idée de Boyer et Moore est d'aligner plutôt de caractère qui provoque l'échec de la comparaison avec son occurrence la plus à droite dans x (sauf la dernière lettre).

Plus précisément, si $x_j \neq t_{i+j}$ et $x_{j+1} \dots t_{i+j+1} \dots t_{i+m-1}$, alors on décale la fenêtre de $d(t_{i+j}) - (m - 1 - j)$



$$\text{décalage} = d(c) - u = d(c) - (m - 1 - j)$$

Attention : cela ne fait pas toujours progresser la recherche :



On obtient un décalage négatif ! Dans ce cas, on décale seulement d'un rang par sécurité.
Algorithme :

```
i ← 0
Tant que i ⩽ n − m :
    j ← m − 1
    Tant que j ⩾ 0 et x_j = t_{i+j} :
        j ← j + 1
    Si j = −1 :
        Occurrence de x à la position i
        i ← i + 1
    Sinon :
        i ← i + max(1, d(t_{i+j}) − (n − 1 − j))
```

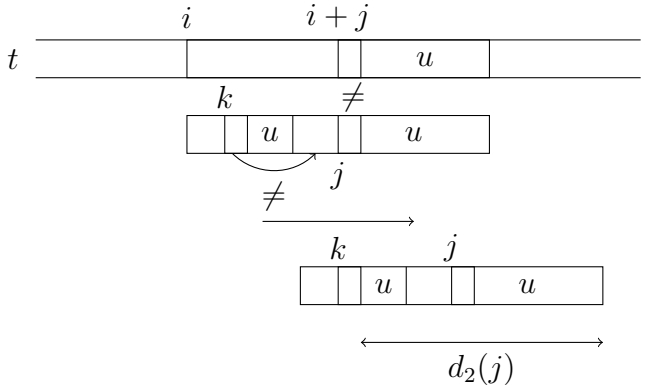
Complexité : dans le pire cas, $\mathcal{O}(nm)$ avec le même exemple qu'en 1.2.2 (page 3).

1.2.4 Algorithme de Boyer-Moore

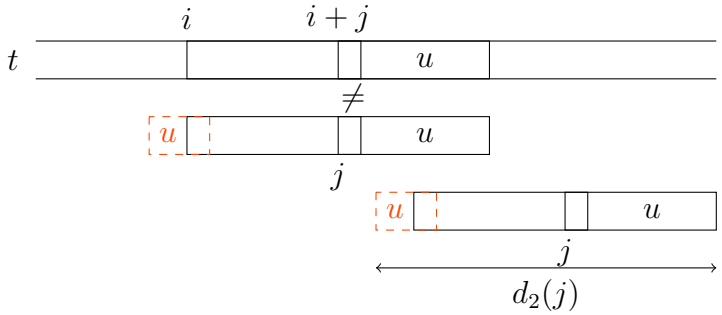
La version complète de l'algorithme de Boyer-Moore utilise une seconde fonction de décalage.

Celle de 1.2.3 (page 4) correspond à la règle du *mauvais caractère* : si un caractère de la fenêtre fait échouer la comparaison avec le motif, on essaie de l'aligner avec sa dernière occurrence dans le motif (sauf la dernière lettre).

Il y a aussi la règle du *bon suffixe* : lorsqu'un caractère fait échouer la comparaison, on a réussi à lire un suffixe de x dans la fenêtre. On peut essayer d'aligner ce suffixe dans le texte avec son occurrence la plus à droite dans x à condition qu'elle soit précédée d'un caractère différent.



$d_2(j)$ = longueur du plus court suffixe de x qui a $x_{j+1} \dots x_{m-1}$ comme suffixe et préfixe et qui n'est pas précédé dans x de x_j.
Si un tel suffixe n'existe pas, on peut chercher le plus long suffixe de u qui est préfixe de x et l'aligner avec le u de la fenêtre.



$d_2(j)$ = longueur de plus court mot w qui a u comme préfixe et x comme suffixe.
Remarque : $|w| \leqslant |u| + |x|$.
Remarque : $\forall j,\ d_2(j) \geqslant 1 + m - j - 1 = m - j$

L'algorithme de Boyer-Moore utilise le décalage maximal entre ceux calculés à partir de d et d_2.
Remarque : si on trouve une occurrence de x ($u = x$), alors on tombe dans le deuxième cas du calcul de d_2 : on cherche le plus long suffixe de x qui est aussi préfixe de x, que l'on appelle le *bord* de x.
Algorithme :

```
i ← 0
Tant que i ⩽ n + m :
    j ← m − 1
    Tant que j ⩾ 0 et x_j = t_{i+j} :
        j ← j − 1
    Si j = −1 :
        Occurrence de x à la position i
        i ← i + d2(j) − m
    Sinon :
        i ← i + max(d(t_{i+j}) − (m − j − 1), d2(j) − (m − j − 1))
```

Exemple :

$x = aababab$

j	-1	0	1	2	3	4	5	6
x		a	a	b	a	b	a	b
$d_2(j)$	14	13	12	6	10	6	8	1

$$
\begin{array}{lcccccccccccccccc|}
t = & a & a & b & b & b & a & b & a & b & a & c & a & a & b & b & a & b & a \\
 & a & a & b & \underset{\times}{a} & b & a & b & & & & & & & & & & & \\
 & & & & & & & & a & a & b & a & \underset{\times}{b} & a & b & & & & \\
 & & & & & & & & & & & & a & a & b & a & b & a & \underset{\times}{b} \\
\end{array}
$$

Complexité : on admet que d_2 peut être précalculée en temps $\mathcal{O}(n)$. Dans le pire des cas, la recherche est encore en $\mathcal{O}(mn)$ (exemple : $x = a^m$ et $t = a^n$)

Remarque : il existe une version plus complexe avec un prétraitement qui permet une complexité $\mathcal{O}(n+m)$ (H.P).

1.3 Algorithme de Karp-Rabin

1.3.1 Principe

L'idée de l'algorithme de Karp-Rabin est d'éviter les comparaisons en temps linéaire entre le motif et le contenu d'une fenêtre en utilisant une fonction de hachage. On parle d'empreinte pour désigner le haché d'une fenêtre d'une chaîne de caractère par la fonction de hachage choisie et on compare les empreintes du motif et du contenu de la fenêtre. Pour contrer le problème des collisions, en cas d'égalité des empreintes, on compare aussi les deux chaînes de caractères.

Algorithme :

Entrées : Texte $t = t_0 \dots t_{n-1}$; Motif $x = x_0 \dots x_{m-1}$ (+ fonction de hachage h choisie à l'avance)

$h_x \leftarrow h(x_0 \dots x_{m-1})$
$h_t \leftarrow h(t_0 \dots t_{m-1})$

Pour i de 0 à $n-m$:
    Si $h_x = h_t$ et (paresseux) $x_0 \dots x_{m-1} = t_i \dots t_{i+m-1}$:
        Occurrence de x à la position i

    Si $i < n-m$:
        $h_t \leftarrow h(t_{i+1} \dots t_{i+m})$

1.3.2 Choix de la fonction de hachage

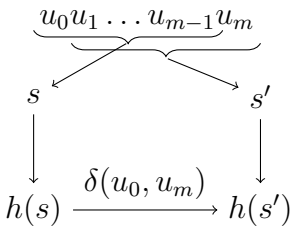
Plusieurs contraintes :

• On doit pouvoir comparer des empreintes en temps constant : en pratique, les empreintes sont des entiers machine, donc ce n'est pas un problème.

• Il doit y avoir peu de collisions avec le haché du motif.

• On doit pouvoir calculer l'empreinte de la fenêtre à l'itération suivante sans lire tous ses caractères (sinon on retrouve la complexité de l'algorithme naïf).

On peut par exemple utiliser une fonction de hachage déroulante, *i.e* une fonction h telle qu'il est possible de calculer en temps constant $h(u_1 \dots u_m)$ à partir de $h(u_0 \dots u_{m-1})$



Exemple : on considère que les caractères sont codés sur un octet, *i.e* on les assimiles à des entiers compris entre 0 et $r-1$ (avec $r=2^8$). On peut voir une chaîne de caractères de longueur m comme l'écriture en base r d'un entier compris entre 0 et r^m-1.

$$P(u_0\ldots u_{m-1}) = \sum_{i=0}^{n-1} u_i r^{m-1-i} = u_0 r^{m-1} + u_1 r^{m-2} + \cdots + u_{m-1}$$

On choisit un nombre premier p et on définit $h(u) = P(u) \mod p$

$$\begin{aligned}
h(u_1\cdots u_m) &= \sum_{i=1}^{m} u_i r^{m-1-i+1} \mod p \\
&= \left(\sum_{i=1}^{m-1} u_i r^{m-i} + u_m\right) \mod p \\
&= \left(r\sum_{i=1}^{m-1} u_i r^{m-1-i} + u_m\right) \mod p \\
&= \left(r\left(\sum_{i=0}^{m-1} u_i r^{m-1-i} - u_0 r^{m-1}\right) + u_m\right) \mod p \\
&= \left(r\left(h(u_0\ldots u_{m-1}) - u_0 r^{m-1}\right) + u_m\right) \mod p
\end{aligned}$$

Donc

$$\delta(u_0,u_m) : e \longmapsto \left(r\left(e - u_0 r^{m-1}\right) + u_m\right) \mod p$$

Remarque : si on précalcule r^{m-1} alors $\delta(u_0,u_m)$ est bien calculable en temps constant.
Complexité du précalcul : $\mathcal{O}(\log m)$ avec l'exponentiation rapide (modulaire).

1.3.3 Implémentation en OCaml

```
let hash (r : int) (p : int) (s : string) : int =
    let e = ref 0 in
    for i = 0 to String.length s - 1 do
        e := (r * !e + (Char.code s.[i])) mod p
        (*Schema de Horner*)
    done;
    !e;;

let delta (r : int) (p : int) (rm : int) (u0 : char) (um : char) (
   e : int) : int =
    (r * (e - (Char.code u0) * rm) + Char.code um) mod p;;
```

```
11
12 let harp_rabin (t : string) (x : string) : int list =
13     let n = String.length t
14     and m = String.length x in
15     let l = ref [] in
16     let r = 256
17     and p = 0x7fffffff in (*p = 2^31 - 1*)
18     let rm = fast_exp_mod r (m - 1) p in (*r^(m - 1) mod p*)
19     let hx = hash r p x
20     and e = ref (hash r p (String.sub 0 m t)) in
21
22     for i = 0 to n - m do
23         if hx = !e && x = String.sub i n t then
24             l := i::!l;
25
26         if i < n - m then
27             e := delta r p rm t.[i] t.[i + m] !e
28     done;
29     !l
```

1.3.4 Étude de la complexité

- Complexité de l'initialisation :
  - Complexité de r_m (`rm`) : $\mathcal{O}(\log m)$
  - Calcul de h_x et $e = h(t_0 \dots t_{n-1})$: $\mathcal{O}(m)$

  $\rightarrow \mathcal{O}(m)$ au total pour l'initialisation ;

- Complexité de la boucle :
  - À chaque itération : une comparaison et un calcul de delta en $\mathcal{O}(1)$;
  - En cas d'égalité d'empreintes : une extraction et une comparaison de chaîne de taille m : $\mathcal{O}(m)$

Au total : $\mathcal{O}(m) + (n-m)\mathcal{O}(1) + \#\text{collisions} \cdot \mathcal{O}(m) = \mathcal{O}(n + m \cdot \#\text{collisions})$
Avec $t = a^n$ et $x = a^m$, on a égalité à chaque itération et on obtient une complexité $\mathcal{O}((n-m)m)$.

Exemple de cas où le pire cas est atteint sans occurrence de x dans t : $x = aa$, $t = arar \dots ar$, $p = 17$, $r = 26$.
$h(aa) = 0 = h(ar) = h(ra)$.

Remarque : dans la publication de Karp et Rabin, l'idée est de choisir un nombre premier p au hasard parmi un ensemble prédéfini pour limiter le risque de collisions.

Admis : si on choisit deux chaînes de taille m aléatoirement, uniformément, la probabilité de collision avec la fonction de hachage choisie est de l'ordre de $\frac{1}{p}$ (qui est $< 10^{-9}$ si $p = 2^{31} - 1$)
Attention, en pratique les chaînes étudiées ne sont pas du tout choisies uniformément.

1.3.5 Extension à la recherche de plusieurs motifs

Même si cet algorithme est moins efficace que l'algorithme de Boyer-Moore pour la recherche d'un seul motif, il devient plus intéressant pour la recherche de plusieurs motifs car on peut l'adapter pour éviter de lancer une recherche par motif.

En choisissant une structure d'ensemble adaptée, il est possible de vérifier en temps constant si l'empreinte d'une sous-chaîne est égale à l'empreinte de l'un des motifs (test d'appartenance).

Il suffit de remplacer le test d'égalité par un test d'appartenance à l'ensemble des empreintes des motifs (précalculées) dans l'algorithme de Karp-Rabin

2 Algorithmes de compression

2.1 Contexte

2.1.1 Principe

On dispose d'un texte (en fait, d'un fichier, quel que soit son contenu, puisque c'est une séquence finie d'octets) et on souhaite réduire l'espace mémoire qu'il occupe *via* un changement d'encodage. On s'intéresse ici à la compression sans perte, *i.e* on doit pouvoir retrouver l'intégralité de l'information initiale à partir du texte compressé.

Formellement, on note Σ l'ensemble des symboles autorisés dans le texte, appelé alphabet (en pratique, $\Sigma = \{0, 1\}$), et Σ^* l'ensemble des séquences finies d'éléments de Σ. On cherche alors une fonction de Σ^* dans Σ^*, nommée `compression`, telle qu'il existe $\texttt{decompression} : \Sigma^* \longrightarrow \Sigma^* \mid \forall t \in \Sigma^*, \texttt{decompression}(\texttt{compression}(t)) = t$

2.1.2 Remarque

On ne peut pas avoir $\forall t \in \Sigma^*,\ |\texttt{compression}(t)| < |t|$.

En effet, la condition $\texttt{decompression} \circ \texttt{compression} = \text{id}$ impose que `compression` soit injective. Mais dans ce cas, on aurait une fonction injective de l'ensemble des textes d'une taille donnée vers l'ensemble des textes de taille strictement inférieure : impossible car le cardinal du second ensemble est strictement inférieur à celui du premier.

Il existe donc toujours des textes dont la version compressée est plus grande que la version décompressée.

L'important est alors de trouver des algorithmes qui compressent efficacement les textes qui sont pertinents pour l'utilisateur.

2.1.3 Algorithmes au programme

On va étudier l'algorithme de Huffman, qui exploite des données sur la fréquence d'apparition des caractères dans le texte pour déterminer un encodage, et l'algorithme de Lenpel-Ziv-Welch, qui ne nécessite pas la connaissance de l'intégrité du texte

pour calculer un encodage car il construit incrémentalement un dictionnaire de codes pour les motifs apparaissant dans le texte.

2.2 Algorithme de Huffman

2.2.1 Principe

L'idée principale de cet algorithme est d'associer un code plus court aux caractères les plus fréquents pour diminuer la taille du texte.

Exemple : `abaabc` nécessite 12 bits avec un code de taille fixe (3 lettres avec 2 bits par caractère) mais seulement 9 bits avec un code de taille variable (ex : `a` = 0, `b` = 10, `c` = 11).

Remarque : pour pouvoir décompresser sans ambiguïté un encodage de taille variable, one ne peut pas encoder un caractère avec un préfixe du code d'un autre caractère. On appelle cela un *code préfixe*.

L'algorithme de Huffman est un algorithme qui, étant donné un texte, produit un code préfixe optimal pour la compression de ce texte, en termes de la taille du texte compressé.

2.2.2 Représentation de l'encodage des symboles

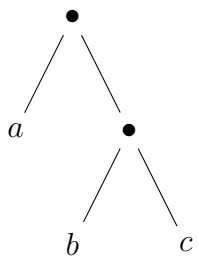
On s'intéresse uniquement à des encodages binaires.

Un code préfixe binaire peut être représenté à l'aide d'un arbre binaire (strict) : dans cet arbre, les caractères sont placés aux feuilles, et le chemin de la racine vers une feuille donne le code du caractère correspondant (descendre à gauche correspond au bit 0, et descendre à droite au bit 1).

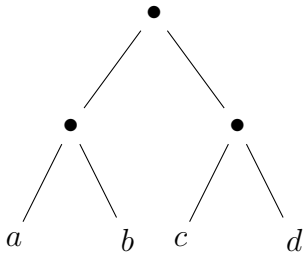
Exemple : le code

a	b	c
0	10	11

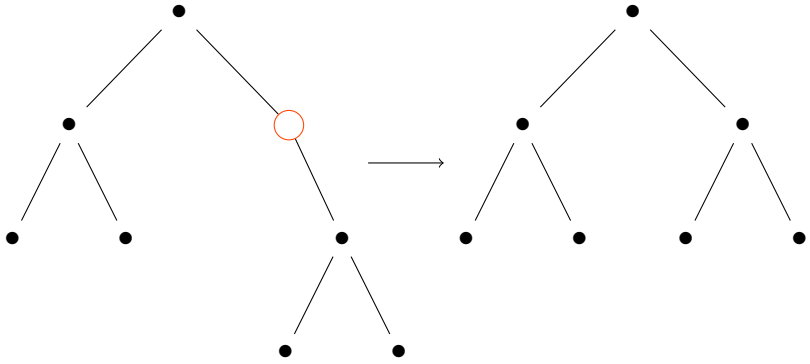
est représenté par :



Autre exemple :



a	b	c	d
00	01	10	11



Pour avoir un code optimal, l'arbre doit être binaire strict.

Remarque : la longueur du code d'un caractère est la profondeur de la feuille correspondante dans l'arbre représentant l'encodage. Déterminer un code préfixe optimal revient à chercher un minimum pour

$$\varphi(t) = \sum_{x \in \text{feuilles}(t)} f(x) \cdot p(x,t)$$

où $p(x,t)$ est la profondeur de la feuille x dans l'arbre t, et $f(x)$ est la fréquence du caractère x dans le texte.

2.2.3 Encodage et décodage

On considère un arbre binaire t représentant un code préfixe et on suppose que l'on travaille sur l'alphabet des octets (type `char`).

• Pour décoder une séquence de bits, il suffit de descendre dans l'arbre selon les bits lus et, lorsque l'on atteint une feuille, de produire le caractère correspondant et de poursuivre la lecture en repartant de la racine.

Code (on représente un bit par un booléen (0 : `false`, 1 : `true`)) :

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

```
let decode (l : bool list) (t : arbre) : char list =
    let rec aux (l : bool list) (a : arbre) : char list =
        match l, a with
        | _, Feuille c -> c :: aux l t
        | [], _ -> [] (*match order is important here*)
        | b::q, Noeud(g, d) -> if b then aux q d else aux q g
    in aux l t
```

Complexité : $\mathcal{O}$(nombre de bits dans le texte compressé).

• Pour encoder un texte, étant donné un arbre représentant un encodage pour chacun des caractères du texte, on ne veut pas parcourir l'arbre à chaque caractère pour déterminer son code. On calcule d'abord une table d'encodage associant à chaque caractère son code. On peut la calculer avec un unique parcours de l'arbre :

```
let codes (t : arbre) : boll list array =
    let a = Array.make 256 [] in (*on travaille sur les octets*)
    let rec aux (t : arbre) (acc : bool list) : unit =
        match t with
        | Feuille c -> a.(Char.code c) <- List.rev acc
        | Noeud(g, d) -> aux g (false::acc) ; aux d (true::acc)
    in aux t [];
    a
```

Complexité : chaque nœud interne est traité en $\mathcal{O}(1)$, et chaque feuille x en $\mathcal{O}(p(x,t))$.
En notant n le nombre de caractères encodés ($n \leqslant |\Sigma|$), il y a n feuilles, et $n-1$ nœuds internes car l'arbre est binaire strict (*cf* chapitre 6, §1.1.10). La hauteur de l'arbre est comprise entre $\lceil \log_2(n) \rceil$ et n.
Donc la complexité est en $\Omega(n \log n)$ et $\mathcal{O}(n^2)$.

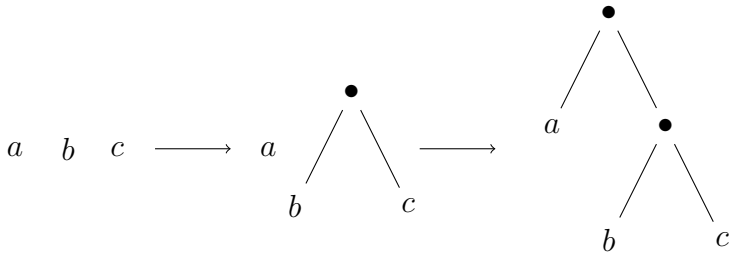
Une fois la table d'encodage calculée, il suffit de remplacer chaque caractère par son code :

```
let encode (l : char list) (t : arbre) : bool list =
    let a = codes t in
    List.flatten (List.map (fun c -> a.(Char.code c)) l)
```

2.2.4 Arbre de Huffman

Il reste à déterminer comment calculer un arbre représentant un code préfixe adapté à la compression d'un texte donné. L'algorithme de Huffman est un algorithme glouton déterminant un arbre optimal.
Cet algorithme part d'une forêt de feuilles et les fusionne deux à deux jusqu'à l'obtention d'un unique arbre. La fusion de deux arbres la construction d'un nouveau nœud dont les deux arbres sont les fils.

Exemple :



L'idée du choix glouton est la suivante : la fusion incrémente la longueur du code des feuilles des deux arbres concernés donc les arbres contenant les caractères les moins fréquents doivent subir le plus de fusion donc être fusionnés en priorité.

On généralise la fonction f de 2.2.2 (page 11) aux arbres en définissant :

$$f(t) = \sum_{x \in \text{feuilles}(t)} f(x)$$

L'algorithme de HUFFMAN s'écrit alors ainsi :

Entrée : texte s

> `occ` $\leftarrow$ table des occurrences des caractères de s
> $F \leftarrow$ ensemble des Feuille c où c est tel que $\texttt{occ}(c) > 0$
>
> Tant que $|F| \geqslant 2$:
> Extraire t_1 de F tel que $f(t_1)$ est minimale
> Extraire t_2 de F tel que $f(t_2)$ est minimale
> $F \leftarrow F \cup \{\texttt{fusion}(t_1, t_2)\}$
>
> Renvoyer l'unique élément de F

Implémentation : la forêt F se comporte comme une file de priorité min où la priorité d'un arbre t est $f(t)$.

On suppose donné un type `fp` associé aux primitives suivantes :

- `create : unit -> fp` qui crée une file de priorité vide ;
- `add : arbre -> int -> fp -> unit` qui insère un arbre avec la priorité (entière) donnée dans la file ;
- `take_min : fp -> arbre*int` qui extrait un arbre de priorité min et qui renvoie aussi sa priorité ;
- `size : fp -> int` qui renvoie la taille de la file