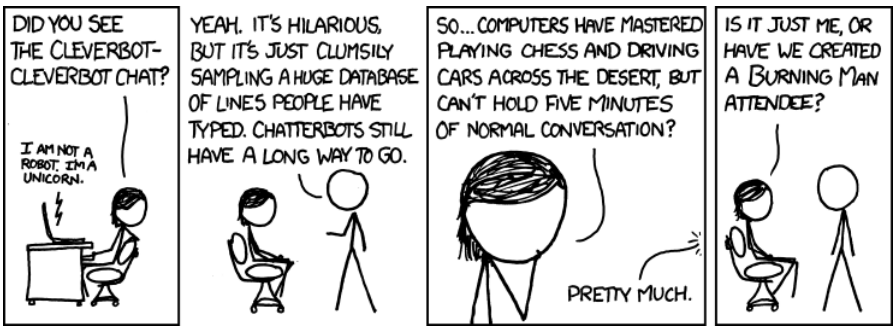
Chapitre Bonus

Algorithmes de texte



Olivier Caffier

1 Compression

1.1 Huffman

Définition - Code préfixe

Un code $c : \Sigma \to \{0;1\}^*$ est dit *préfixe* ssi :
$$\forall (a,b) \in \Sigma^2,\ c(a) \text{ n'est pas un préfixe de } c(b) \text{ et } c(b) \text{ n'est pas un préfixe de } c(a).$$

Problème

Soit $occ : \Sigma \to \mathbb{N}$, on veut construire $c : \Sigma \to \{0;1\}^*$ un code préfixe qui minimise
$$\sum_{a \in \Sigma} occ(a) \times \text{longueur}(c(a))$$

Algorithme de Huffman

Soit $m \in \Sigma^*$, on note $\text{occ}(a)$ le nombre d'occurrence de a dans m pour tout $a \in \Sigma$. On procède ensuite de la manière suivante :

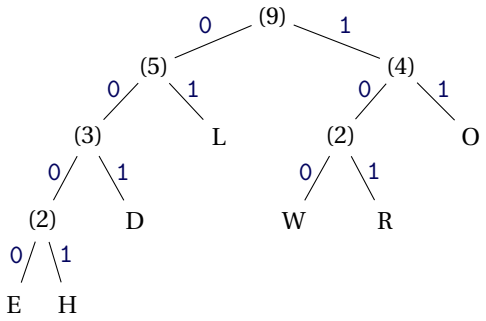
1. On construit la table d'occurrence, associant a et $\text{occ}(a)$ pour chaque lettre apparaissant dans m.
2. On initialise une forêt pondérée avec des arbres réduits à une feuille pour chaque elt a de Σ associé à $occ(a)$.
3. À chaque étape, on fusionne les deux arbres de poids minimal et on insère le résultat dans la forêt avec comme poids la somme des poids des 2 arbres fusionnés.
4. Quand la forêt est réduite à un seul arbre, alors on le renvoie. On en déduit alors un code binaire de manière assez classique : aller à gauche nous fait ajouter un 0, aller à droite nous fait ajouter un 1.

Un exemple vaut mieux qu'un long discours, compressons le mot "HELLOWORLD" :

1. On construit la table d'occurrence :

a	E	O	H	W	R	L	D
$\text{occ}(a)$	1	2	1	1	1	2	1

2. On construit l'arbre de Huffman. On dispose d'abord de arbres correspondant à des noeuds de chaque lettre, associés à leur poids. On fusionne ensuite les deux arbres avec les poids les plus faibles, le nouveau poids de cet arbre étant la somme des deux arbres.
3. On continue jusqu'à ce qu'il ne reste qu'un seul arbre :



4. On construit la table de compression à l'aide d'un parcours en profondeur :

E	H	D	L	W	R	O
0000	0001	001	01	100	101	11

5. On écrit dans le fichier le code de l'arbre, et en lisant le fichier texte lettre par lettre, on obtient le code suivant :

   0001 0000 01 01 11 100 11 101 01 001

   On comble le reste avec des 0 (en codant sur 8 bits par exemple), et on obtient : 00010000 01011110 01110101 00100000

Code texte	00010000	01011110	01110101	00100000
Nombre associé	16	94	117	32
Code de l'arbre	0001E1H101L001W1R10			

➢ Le code de l'arbre correspond au parcours préfixe de l'arbre, en notant `0` chaque noeud et `1X` chaque feuille (avec `X` la lettre en question). Ce code permet donc de représenter l'arbre de manière unique. D'où l'encodage recherché.

Proposition - Optimalité de l'algo de Huffman

L'arbre a obtenu à l'aide de cet algorithme minimise bien

$$\varphi(a) = \sum_{\alpha \in \Sigma} occ(\alpha) \times prof_a(\alpha)$$

DÉMONSTRATION.

- **1ère étape : Montrer qu'il existe un arbre qui minimise φ dans lequel deux des lettres d'occurrences minimales sont sœurs.**

  Soit $\mathcal{A}$ un arbre optimal, on fixe a et b deux lettres d'occurrences minimales avec $occ(a) \leq occ(b)$. Soit c de profondeur maximale dans $\mathcal{A}$, alors elle a une soeur d dans $\mathcal{A}$ (sinon, cf. dessin 1 à éviter) car sinon on aurait un bit inutile pour coder c, on suppose $occ(c) \leq occ(d)$.
  Considérons $\mathcal{A}'$ où on échange a et c ainsi que b et d. Ainsi,

$$\begin{aligned}\varphi(\mathcal{A}') &= \varphi(\mathcal{A}) - occ(a) \times p_{\mathcal{A}}(a) + occ(a) \times p_{\mathcal{A}}(c) - occ(b) \times p_{\mathcal{A}}(b) + occ(b) \times p_{\mathcal{A}}(d) \\ &\quad - occ(c) \times p_{\mathcal{A}}(c) + occ(c) \times p_{\mathcal{A}}(a) - occ(d) \times p_{\mathcal{A}}(d) + occ(d) \times p_{\mathcal{A}}(b) \\ &= \varphi(\mathcal{A}) + \underbrace{(occ(a) - occ(c))}_{\leq 0}\underbrace{(p_{\mathcal{A}}(c) - p_{\mathcal{A}}(a))}_{\geq 0} + \underbrace{(occ(b) - occ(d))}_{\leq 0}\underbrace{(p_{\mathcal{A}}(d) - p_{\mathcal{A}}(b))}_{\geq 0} \\ &\leq \varphi(\mathcal{A})\end{aligned}$$

  Or $\mathcal{A}$ étant minimal, on en déduit que $\varphi(\mathcal{A}) = \varphi(\mathcal{A}')$

- **2ème étape : Outil pour l'hérédité**.

  Soit $\mathcal{A}$ un arbre optimal pour (Σ, occ) ayant deux de ses lettres d'occurrences minimales sœurs notées α et β. Alors si on remplace le sous-arbre contenant les deux soeurs par une unique feuille γ pour obtenir $\mathcal{A}'$, $\mathcal{A}'$ est optimal pour le problème $((\Sigma \backslash \{\alpha, \beta\} \cup \{\gamma\}, occ')$ avec $\forall x \in \Sigma \backslash \{\alpha, \beta\}$, $occ'(x) = occ(x)$ et $occ(\alpha) + occ(\beta)$.

  On a

$$\begin{aligned}\varphi(\mathcal{A}') &= \varphi(\mathcal{A}) - occ(\alpha) \times p_{\mathcal{A}}(\alpha) - occ(\beta) \times p_{\mathcal{A}}(\beta) + (occ(\alpha) + occ(\beta))(p(\alpha) - 1) \\ &= \varphi(\mathcal{A}) - occ(\alpha) - occ(\beta)\end{aligned}$$

  Supposons par l'absurde qu'il existe $\mathcal{A}''$ tq $\varphi(\mathcal{A}'') < \varphi(\mathcal{A}')$ pour le problème (Σ', occ').

  On peut alors construire $\mathcal{A}'''$ en remplaçant γ par le sous-arbre $\mathcal{N}(\alpha, \beta)$ dans $\mathcal{A}'$ tel que :

$$\begin{aligned}\varphi(A''') &= \varphi(A'') + occ(\alpha) + occ(\beta) \\ &< \varphi(A') + occ(\alpha) + occ(\beta) = \varphi(A)\end{aligned}$$

  or $\mathcal{A}$ est optimal, absurde.

- **3ème étape : Preuve de l'optimalité de l'arbre de Huffman par rec. sur** $|\Sigma| \geq 1$

  Initialisation Si $|\Sigma| = 1$, c'est évident.

  Hérédité Soit $\mathcal{A}$ un arbre optimal pour (Σ, occ) avec deux lettres d'occ. minimales α et β sœurs.

  On a montré que $\mathcal{A}'$ où on remplace $\mathcal{N}(\alpha, \beta)$ par γ est optimal pour le problème (Σ', occ') défini à l'étape 2. Par H.R, l'algo glouton sur (Σ', occ') renvoie un arbre $\mathcal{A}''$ tq $\varphi(\mathcal{A}'') = \varphi(\mathcal{A})$.

  Or, sur (Σ, occ), l'algo renvoie $\mathcal{A}'''$ construit à partir de $\mathcal{A}''$ en remplaçant γ par $\mathcal{N}(\alpha, \beta)$. Il reste à montrer que $\varphi(\mathcal{A}''') = \varphi(\mathcal{A})$.

  Par les mêmes calculs qu'à la 2ème étape, on sait que

$$\begin{aligned}\varphi(\mathcal{A}''') &= \varphi(\mathcal{A}'') - occ(\alpha) - occ(\beta) \\ \text{et } \varphi(\mathcal{A}''') &= \varphi(\mathcal{A}') - occ(\alpha) - occ(\beta)\end{aligned}$$

  ce qui permet de conclure.

Algorithme de Huffman - Décompression

Pour décompresser un code issu d'un processus Huffman, on suit les deux étapes ci-contre :

1. On **reconstruit** l'arbre de Huffman ;
2. On **lit** chaque bit en parcourant en parallèle l'arbre et à chaque fois qu'on arrive sur une feuille, on **inscrit** la lettre trouvée et on retourne à la racine

➢ Le dernier octet du code contient des 0 supplémentaires qui ne font pas partie du codage, et pour pouvoir décompresser, on rajouter à la fin du fichier compressé un entier de $[\![0;7]\!]$ qui nous informe du nombre de zéros ajoutés.

1.2 Lempel-Ziv-Welch

Intérêt de cette nouvelle approche :

- On compresse au fur et à mesure de la lecture sans faire de pré-traitement du texte
- On ne compresse pas lettre à lettre mais on va associer un code à des facteurs du texte

Principe de la compression Initialement, on dispose d'une table qui associe à chaque lettre de l'alphabet un code et au cours de la lecture on va ajouter des mots dans la table.

0	…	65	66	…	255	256
		A	B	…	ÿ	case libre

Algorithme - LZW

$m \leftarrow$ 1ère lettre du texte
while lecture du texte **do**
  if on lit la lettre x **then**
    if mx est dans la table **then**
      $m \leftarrow mx$
    else
      On incrit dans le fichier compressé le code de m
      On insère mx dans la table
      $m \leftarrow x$
On inscrit dans le fichier compressé le code de m le mot restant.

Exemple Prenons le mot `ABBBABBAABBA`, on effectue alors les instructions suivantes :

- **Boucle 1 :**

A	B	B	B	A	B	B	A	A	B	B	A
	↑										

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(AB,256)} \\ \texttt{code}(m) = 65 \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 2 :**

A	B	B	B	A	B	B	A	A	B	B	A
		↑									

$\begin{cases} m = \texttt{B} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{BB}$ n'apparait pas dans notre table!

$$\Rightarrow \begin{cases} \texttt{add(BB,257)} \\ \texttt{code}(m) = 66 \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 3 :**

A B | B | B A B B A A B B A

↑

$\begin{cases} m = \texttt{B} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{BB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{BB}$$

- **Boucle 4 :**

A B | B B | A B B A A B B A

↑

$\begin{cases} m = \texttt{BB} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{BBA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(BBA,258)} \\ \texttt{code(}m\texttt{) = 257} \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 5 :**

A B B B | A | B B A A B B A

↑

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{AB}$$

- **Boucle 6 :**

A B B B | A B | B A A B B A

↑

$\begin{cases} m = \texttt{AB} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{ABB}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(ABB,259)} \\ \texttt{code(}m\texttt{) = 256} \\ m \leftarrow \texttt{B} \end{cases}$$

- **Boucle 7 :**

A B B B A B | B | A A B B A

↑

$\begin{cases} m = \texttt{B} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{BA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(BA,260)} \\ \texttt{code(}m\texttt{) = 66} \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 8 :**

A B B B A B B | A | A B B A

↑

$\begin{cases} m = \texttt{A} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{AA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(AA,261)} \\ \texttt{code}(m) = 65 \\ m \leftarrow \texttt{A} \end{cases}$$

- **Boucle 9 :**

A	B	B	B	A	B	B	A	A	B	B	A
									↑		

$\begin{cases} m = \texttt{A} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{AB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{AB}$$

- **Boucle 10 :**

A	B	B	B	A	B	B	A	A	B	B	A
										↑	

$\begin{cases} m = \texttt{AB} \\ x = \texttt{B} \end{cases}$ et $mx = \texttt{ABB}$ apparait dans notre table !

$$\Rightarrow \{ m \leftarrow \texttt{ABB}$$

- **Boucle 11 :**

A	B	B	B	A	B	B	A	A	B	B	A
											↑

$\begin{cases} m = \texttt{ABB} \\ x = \texttt{A} \end{cases}$ et $mx = \texttt{ABBA}$ n'apparait pas dans notre table !

$$\Rightarrow \begin{cases} \texttt{add(ABBA,262)} \\ \texttt{code}(m) = 259 \\ m \leftarrow \texttt{A} \end{cases}$$

- **Fin de la boucle `while`**

$$\{ \texttt{code}(m) = 65$$

Donc, en reprenant les étapes balisées en rouge, on obtient le code suivant :

65 66 257 256 66 65 259 65

La conversion en binaire est laissée au lecteur...

Bon, on va pas se mentir, c'est fastidieux... À l'oral on présenterait plus comme ceci :

m	**code obtenu**	code ajouté dans la table
A	**65**	256 : AB
B	**66**	257 : BB
B		258 : BBA
BB	**257**	259 : ABB
	256	260 : BA
A		261 : AA
B	**66**	262 : ABBA
...	**65**	
...	**259**	
...	**65**	

et à l'écrit ? Il y a des questions qu'il ne vaut mieux pas poser...

Algorithme LZW - Décompression

On a le pseudo-code suivant, en notant C le code obtenu et `t` notre table de base, notre "dictionnaire" (donc qui va de 0 à 255) :

```
n ← |C|
v ← lire C
m ← t[v]
Afficher m
for i ← 1 to n − 1 do
    v ← lire C
    if v est déjà une clé t then
        mot_tmp ← t[v]
    else
        mot_tmp ← Concaténer(m,m[0])
    Afficher mot_tmp
    Ajouter(Concaténer(m,mot_tmp[0]))
    m ← mot_tmp
```

Le mot d'ordre est donc le suivant :

On simule la compression

Reprenons notre code C = 65 66 257 256 66 65 259 65

- $n = 8$
- v ← 65
- m ← A
- mot_vide ← ε

$$\Rightarrow \{\texttt{Afficher(m) = A}$$

- **Entrée dans la boucle :**
  1. ∘ v ← 66
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[66] = B} \\ \texttt{Afficher(mot\_tmp) = B} \\ \texttt{Add(Concaténer(A,B),256) = Add(AB,256)} \\ \texttt{m} \leftarrow \texttt{B} \end{cases}$$

  2. ∘ v ← 257
     ∘ v n'est pas une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \text{Concaténer(B,A)} = \text{BB} \\ \texttt{Afficher(mot\_tmp) = BB} \\ \texttt{Add(Concaténer(B,B),257) = Add(BB,257)} \\ \texttt{m} \leftarrow \text{BB} \end{cases}$$

  3. ∘ v ← 256
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[256] = AB} \\ \texttt{Afficher(mot\_tmp) = AB} \\ \texttt{Add(Concaténer(BB,A),258) = Add(BBA,258)} \\ \texttt{m} \leftarrow \text{AB} \end{cases}$$

  4. ∘ v ← 66
     ∘ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[66] = B} \\ \texttt{Afficher(mot\_tmp) = B} \\ \texttt{Add(Concaténer(AB,B),259) = Add(ABB,259)} \\ \texttt{m} \leftarrow \text{B} \end{cases}$$

5. ◦ v ← 65
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[65] = A} \\ \texttt{Afficher(mot\_tmp) = A} \\ \texttt{Add(Concaténer(B,A),260) = Add(BA,260)} \\ \texttt{m} \leftarrow \texttt{A} \end{cases}$$

6. ◦ v ← 259
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[259] = ABB} \\ \texttt{Afficher(mot\_tmp) = ABB} \\ \texttt{Add(Concaténer(A,A),261) = Add(AA,261)} \\ \texttt{m} \leftarrow \texttt{ABB} \end{cases}$$

7. ◦ v ← 65
   ◦ v est déjà une clé de t :

$$\Rightarrow \begin{cases} \texttt{mot\_tmp} \leftarrow \texttt{t[65] = A} \\ \texttt{Afficher(mot\_tmp) = A} \\ \texttt{Add(Concaténer(ABB,A),262) = Add(ABBA,262)} \\ \texttt{m} \leftarrow \texttt{A} \end{cases}$$

- **Fin de la boucle** - on a donc trouvé le mot (en mettant un espace à chaque différence d'affichage)

`A B BB AB B A ABB A`

qui est bien le mot qu'on avait encodé au début de ce paragraphe :)

2 Recherche dans un texte

On veut résoudre le problème suivant :

Recherche dans un texte

- **Entrée :** Un texte `t` codé dans un tableau et un motif `m` codé dans un tableau.
- **Sortie : 1** si `m` apparait dans `t`, **0** sinon.

Un premier algorithme naïf pour résoudre ce problème consisterait à parcourir le texte et pour chaque lettre, vérifier si à partir de cette dernière on peut lire `m` :

Pseudo-code d'une approche naïve

```
for i← 0 to |t|-|m|-1 do
    check ← true
    for j← 0 to |m|-1 do
        if t[i+j] != m[j] then
            check ← false
    if check then
        return true
return false
```

EXEMPLE. Prenons ce passage de L'Étranger d'Albert Camus :

« Aujourd'hui, maman est morte. Ou peut-être hier, je ne sais pas. »

et recherchons le motif « maman » :

A	u	j	o	u	r	d	'	h	u	i	,		m	a	m	a	n	...
m	a	m	a	n														

Ici, `A` ne coincide pas avec `m`, mais on est obligé de parcourir jusqu'au `u` pour pouvoir décaler `maman` d'un indice vers la droite :

A	u	j	o	u	r	d	'	h	u	i	,		m	a	m	a	n	...
	m	a	m	a	n													

Etc..., jusqu'à tomber sur le motif recherché :

A	u	j	o	u	r	d	'	h	u	i	,		m	a	m	a	n	...
													m	a	m	a	n	

⇒ On a alors une complexité en $\mathcal{O}(|t| \times |m|)$

En termes de notations, notre "boucle `j`" s'appellera le **scan** tandis que notre "boucle `i`" s'appellera la **lecture**.

Nous allons donc voir maintenant deux algorithmes au programme qui permettent de répondre à ce problème avec une bien meilleure complexité.

2.1 Boyer-Moore-Horspool

Intuitivement, au lieu de procéder par un scan de gauche à droite, **on va scanner de droite à gauche tout en lisant de gauche à droite** : lors d'un mauvais scan (i.e qui change la valeur de `check` à `false`), **on saisit la lettre dans `t` alignée avec la dernière lettre de `m` et on regarde dans `m` la première occurrence de cette dernière en partant de la droite** :

- Dans le cas où cette occurrence existe : on déplace notre mot de sorte que cette occurrence se retrouve au bon endroit pour être scannée en même temps que la lettre qui nous avait posé problème auparavant.
- Si `m` ne comporte pas cette lettre : alors on déplace `m` afin que la première lettre de notre motif se retrouve un indice plus tard que cette lettre.

EXEMPLE.

Prenons une autre citation d'Albert Camus :

« Il faut imaginer Sisyphe heureux »

et cherchons le motif `m = 'sisyphe'` :

1.

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
s	i	s	y	p	h	e																										

Ici, notre premier scan est mis en défaut par une différence entre la lettre `t` et la lettre `e`, on cherche donc la première occurence de `t` dans le motif `m` : elle n'existe pas. On peut donc faire un "gros" saut vers la droite pour mettre la première lettre de `m` à droite de la lettre `t` qui nous a posé problème.

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
							s	i	s	y	p	h	e																			

2.

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
							s	i	s	y	p	h	e																			

De même, `n` et `e` ne collent pas ensemble, on fait encore un "gros" saut car `n` n'apparait pas dans `m` :

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
														s	i	s	y	p	h	e											

3.

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
														s	i	s	y	p	h	e											

Mais ici, `y` apparaît bien dans `m`, on déplace donc `m` vers la droite de sorte que la première occurence de `y` (depuis la droite je le rappelle) soit bien placée :

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
																	s	i	s	y	p	h	e								

4. Et on a donc bien trouvé notre motif :

i	l		f	a	u	t		i	m	a	g	i	n	e	r		s	i	s	y	p	h	e		h	e	u	r	e	u	x
																	s	i	s	y	p	h	e								

Et tout ça avec un nombre record de comparaisons par rapport à notre version naïve! Nous sommes donc heureux comme Sisyphe :)

Bon maintenant, ça reste une version "intuitive", j'ai évoqué un décalage sans formalisme, attaquons-nous donc à ce dernier!

De manière assez immédiate, on décalait à l'œil nu, mais comment le faire dans un programme? On va construire une **table de décalage** pour savoir où se trouve la première occurence de telle lettre dans `m` (en partant de la droite!) et surtout pour répondre à la question : « De combien de cases dois-je décaler `m`? »

Ici, on va utiliser une table de hachage, on va notamment la construire en lisant le motif :

Construire la table de décalage (version simple)

```
res ← table de hachage vide
for i ← 0 to |m|-2 do
    res[m[i]] ← |m|-1-i
return res
```

EXEMPLE.

Ici, `decalage('sisyphe')` nous donnerait

```
H : 1
P : 2
Y : 3
S : 4
I : 5
```

```
let decalage (m : string) : (char,int) Hashtbl.t =
  let n = String.length m in
  let res = Hashtbl.create n in
  for i = 0 to n-2 do
    Hashtbl.replace res (m.[i]) (n-1-i)
  done;
  res
```

Avec cette fonction `decalage`, on en déduit la version avec optimisation première de Boyer-Moore, plus communément appelée Boyer-Moore-Horspool :

Algorithme BMH opti1

```
D ← decalage(m)
i ← |m|-1
while i < |t| do
    check ← true
    for j ← 0 to |m|-1 do
        if t[i-j] != m[|m|-1-j] then
            check ← false
    if check then
        return true
    else
        if t[i] est une clé de D then
            i ← i + D[t[i]]
        else
            i ← i + |m|
return false
```

Remarques (à connaître) :

- Cette méthode nécessite au préalable le calcul d'une table de hachage :
  - Pour les petits alphabets, on se contentera d'un tableau
  - Sinon, une table de hachage ou un arbre seront notre option.
- Ne fonctionne pas bien pour les alphabets très petits (typiquement le binaire) ou les motifs de très petite tailles, car les décalages vont avoir tendance à être minimes.
- Maintenant, on pourrait se demander s'il n'existe pas une meilleure fonction de décalage qui prend encore plus en compte le scan que nous effectuons, surtout si il y a des redondances etc... ⇒ cf. Boyer-Moore 2ème opti (HP).

➢ Voilà, c'est tout ce qu'il y a à savoir selon le programme officiel sur cet algorithme, donc il faut surtout bien connaître le principe et savoir le dérouler et tout se passera bien normalement!

```
exception Motif_trouve

let bmh (t : string) (m : string) : bool =
  try
    let len_m = String.length m in
    let len_t = String.length t in
    let d = decalage m in
    let i = ref (len_m-1) in
    while (!i < len_t) do
      let check = ref true in
      for j = 0 to len_m -1 do
        if t.[!i - j] <> m.[len_m - 1 -j] then check := false
      done;
      if !check then
        raise Motif_trouve
      else if Hashtbl.mem d (t.[!i]) then i := !i + Hashtbl.find d (t.[!i])
      else i := !i + len_m
    done;
    false
  with Motif_trouve -> true
```

Proposition - Complexité de Boyer-Moore-Horspool

Alors comme dit dans les remarques précédentes, on améliore pas vraiment la complexité "pire cas" car on reste toujours (notamment dans le cas d'un alphabet petit ou d'un motif petit) en

$$\mathcal{O}(|\texttt{m}| \times |\texttt{t}|)$$

Néanmoins, notre complexité moyenne est nettement améliorée (on l'admettra) en

$$\mathcal{O}(|\texttt{m}| + |\texttt{t}|)$$

2.2 Rabin-Karp

Le plus coûteux dans une recherche textuelle, c'est la comparaison du motif avec un facteur du texte. Pour éviter ces comparaisons, l'algorithme de Rabin-Karp consiste à calculer une **empreinte** (un *hash*) du motif et à la comparer avec l'empreinte du facteur (de même taille que le motif) à la position actuelle.

Une empreinte est le résultat d'une **fonction de hachage**. C'est donc un entier : on peut comparer deux empreintes en temps $\mathcal{O}(1)$.

- Si les deux empreintes sont différentes : on est certain que le facteur étudié ne correspond pas au motif, on passe donc à l'indice suivant.
- Sinon : il faut comparer manuellement les caractères du motif et du facteur de texte et si ils sont tous égaux, on a alors trouvé une occurence du motif et sinon, c'est qu'il s'agissait d'une **collision** avec le hash du motif, on passe à l'indice suivant.

EXEMPLE. Prenons l'empreinte f suivante :

$$f : \texttt{t[k,...,k+|m|-1]} \mapsto \text{nombre de } \texttt{i} \text{ dans } \texttt{t[k,...,k+|m|-1]}$$

et considérons cet extrait de « À une passante » de Charles Baudelaire :

« Un éclair … puis la nuit! Fugitive beauté »

et recherchons le motif `m = nuit` qui possède un seul `i` :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t

Ici, rien ne sert de lancer le scan vu que la partie rouge ne dispose pas du bon nombre de `i`, on passe donc à l'indice suivant :

U	n		é	c	l	a	i	r	…	p	u	i	s		l	a		n	u	i	t

De même ici, rien ne sert de lancer le scan vu que la partie rouge ne dispose pas du bon nombre de `i`, on passe donc à l'indice suivant etc, jusqu'à arriver à la situation suivante :

U	n		é	c	l	a	i	r	...	p	u	i	s		l	a		n	u	i	t

Ici, on a le bon nombre de `i` mais le `t` de `m` ne colle pas au `i` :

U	n		é	c	l	a	i	r	...	p	u	i	s		l	a		n	u	i	t
				n	u	i	t														

On peut donc décaler etc, jusqu'à arriver à :

U	n		é	c	l	a	i	r	...	p	u	i	s		l	a		n	u	i	t
																	n	u	i	t	

On décale donc d'un, on garde le bon nombre de `i` donc on peut lancer le scan et on trouve :

U	n		é	c	l	a	i	r	...	p	u	i	s		l	a		n	u	i	t
																		n	u	i	t

Remarques importantes :

➢ On remarque déjà **l'importance d'avoir un bon critère de différenciation** pour l'empreinte, choisir le nombre `i` (alors que c'est une des voyelles les plus utilisées, 6.59% du corpus de Wikipedia quand même) n'est peut-être pas le choix le plus optimal!

➢ En plus de cela, **il faudrait que le pré-traitement se fasse en** $\mathcal{O}(1)$ pour éviter d'avoir une complexité catastrophique en calculant à chaque fois l'empreinte et en scannant par la suite (on serait en $\mathcal{O}(|t| \times (|m| + f_{|m|}))$ en fait). Il faut donc essayer de trouver un moyen de **calculer l'empreinte avec l'empreinte précédente** (surtout qu'on ne fait que des sauts de 1 donc on voit très bien une formule de récurrence arriver).

Pour récapituler :

Ce que doit respecter notre empreinte

Une empreinte utilisée par notre algorithme doit à tout prix :

- Avoir peu de collisions (donc avoir un bon critère de différenciation).
- Le calcul de l'empreinte du motif doit se faire en $\mathcal{O}(|m|)$.
- On doit pouvoir calculer l'empreinte du facteur en position `i+1` **en temps constant** à partir de l'empreinte du facteur en position `i`, on parle alors d'**empreinte tournant ou glissante** (*rolling hash* en anglais).

Reprenons

$$f : t[k,\dots,k+|m|-1] \mapsto \text{nombre de } i \text{ dans } t[k,\dots,k+|m|-1]$$

Pour calculer $f(t[k+1,\dots,k+|m|])$, on a juste besoin de savoir si `t[k]` était un `i`, de même pour `t[k + |m|]`, et de l'empreinte $f(t[k,\dots,k+|m|-1])$. On revient à la formule de récurrence suivante :

$$\begin{cases} f(t[0,\dots,|m|-1]) = \text{nombre de } i \text{ dans } t[0,\dots,|m|-1] \\ \forall k \text{ tq. } 0 < k < |t|-|m|,\ f(t[k+1,\dots,k+|m|]) = f(t[k,\dots,k+|m|-1]) + \delta_{(t[k]==i)} + \delta_{(t[k+|m|]==i)} \end{cases}$$

Encore une fois, compter le nombre de `i` n'est pas la meilleure des techniques, on va donc utiliser une empreinte spécifique très connue : **l'empreinte de Rabin**.

Définition - Empreinte de Rabin

Soit p un nombre premier (on le prend premier pour justement limiter les collisions).
On considère que le texte et le motif sont passés en paramètres dans le code ASCII, qui va nous permettre d'utiliser directement des opérations algébriques.
On définit alors l'empreinte de Rabin f de la manière suivante :

$$\begin{cases} f(x_0 x_1 \dots x_{|m|}) = x_0 \times p^{|m|-1} + x_1 \times p^{|m|-2} + \dots x_{|m|-1} \\ \forall k \in [\![0; |t|-|m|]\!],\ f(x_{k+1} \dots x_{k+|m|}) = \left(f(x_k \dots x_{k+|m|}) - x_k \times p^{|m|-1}\right) \times p + x_{k+|m|} \end{cases}$$

➢ L'algorithme de Horner pourra effectivement s'avérer utile pour le calcul du cas initial. On remarque qu'on a besoin de garder en mémoire uniquement :

- $p^{|m|-1}$ et p.
- La valeur de l'empreinte précédente.
- La valeur de l'empreinte du motif.