Cours : Texte

Table des matières

1 Définitions

Définition 1.1: Alphabet, mot

- Un *alphabet* Σ est un ensemble fini non vide, dont les éléments sont appelés *lettres* ou *symboles* ou *caractères*.
- Un *mot* ou *chaîne de caractères* m sur Σ est :
  - soit le *mot vide* noté ε.
  - soit une suite finie $m_1 \dots m_n$ d'éléments de Σ, avec $n \geqslant 1$.
- L'ensemble des mots sur Σ est noté Σ^*.

Remarques

⇨ On confondra souvent la lettre a et le mot dont la seule lettre est a. Attention cependant, ce sont en réalité deux objets de nature différente, et ils sont typiquement représentés par des objets de type différents dans un langage informatique (`char` et `string` en OCaml, par exemple).

⇨ On peut considérer le mot vide comme étant de la forme $m_1 \dots m_n$ avec $n = 0$.

⇨ On peut aussi donner une définition inductive d'un mot : un mot est
- soit le mot vide.
- soit de la forme au, où $a \in \Sigma$ et u est un mot.

Exemple

⇨ Quelques alphabets que l'on rencontre couramment :
—$\Sigma \coloneqq \{a, b\}$ qu'on utilisera souvent comme exemple.
—$\Sigma \coloneqq \{0, 1\}$ quand on s'intéresse à un flux de bits.
—l'alphabet « usuel » à 26 lettres.
—l'ensemble des 256 (ou plutôt 128) caractères ASCII standard.
—$\Sigma \coloneqq \{A, T, G, C\}$ si l'on traite des séquences d'ADN.

Définition 1.2: Longueur

- La *longueur* d'un mot m est notée $|m|$. Elle est définie par :
  - $|\varepsilon| = 0$
  - $|m_1 \dots m_n| = n$ (où $m_1, \dots, m_n \in \Sigma$).
- L'ensemble des mots de longueur n sur un alphabet Σ est noté Σ^n.
- Le *nombre d'occurrences* d'une lettre x dans m, ou *longueur en* x de m, est noté $|m|_x$. Il est défini par :

$$|m|_x \coloneqq \text{Card}\{i \in [\![1, n]\!] \mid m_i = x\}$$

Définition 1.3: Concaténation

La *concaténation* de deux mots u et v sur un même alphabet Σ est notée $u \cdot v$. Elle est définie comme suit :
- $\varepsilon \cdot v = v$
- $u \cdot \varepsilon = u$

— $(u_1 \dots u_n) \cdot (v_1 \dots v_p) = u_1 \dots u_n v_1 \dots v_p$.

Pour un mot u et un entier $n \geqslant 0$, on définit u^n par :

— $u^0 = \varepsilon$

— $u^{n+1} = u \cdot u^n$.

Remarques

⇨ L'opération de concaténation est associative : on notera donc $u \cdot v \cdot w$ pour $u \cdot (v \cdot w) = (u \cdot v) \cdot w$.

⇨ On vérifie facilement que les règles de calcul usuelles sur les puissances s'appliquent : $u^m \cdot u^n = u^{m+n}$ et $(u^m)^n = u^{mn}$.

⇨ On écrit parfois (souvent) uv à la place de $u \cdot v$. Attention cependant, on peut arriver à des situations ambiguës : si l'on écrit $u = u_1 \dots u_n$, veut-on dire que $u_1, \dots, u_n$ sont les lettres qui constituent u ou que u est la concaténation des mots $u_1, \dots, u_n$?

Définition 1.4

Préfixe, suffixe, facteur Soit u et v deux mots sur un même alphabet Σ.

— u est un *préfixe* de v s'il existe un mot w tel que $v = u \cdot w$.

— u est un *suffixe* de v s'il existe un mot w tel que $v = w \cdot u$.

— u est un *facteur* de v s'il existe deux mots w et z tels que $v = w \cdot u \cdot z$.

Remarques

⇨ v est un préfixe, un suffixe et un facteur de v (en prenant $w = z = \varepsilon$) ;

⇨ ε est un préfixe, un suffixe et un facteur de v pour tout mot v.

⇨ Un préfixe (ou un suffixe) de v est un facteur de v.

Exercice 1

⇨ Soit $u = abab$. Donner les préfixes, les suffixes et les facteurs de u.

Proposition 1.5

Soient $u, v, u', v' \in \Sigma^*$ tels que $u \cdot v = u' \cdot v'$. Il existe un unique mot t tel que :

— soit $u = u' \cdot t$ et $v' = t \cdot v$

— soit $u' = u \cdot t$ et $v = t \cdot v'$.

Proposition 1.6

Si u et u' sont deux préfixes d'un mot v, alors u est un préfixe de u' ou u' est un préfixe de u.

Remarques

⇨ Bien évidemment, si u est un préfixe de u' et u' est un préfixe de u, alors $u = u'$.

⇨ On a une propriété analogue sur les suffixes.

Exercice 2

⇨ Démontrer la proposition sur les prefixes.

Définition 1.7: Sous-mot

Soient $u, v \in \Sigma^*$, $u = u_1 \dots u_n$ ($n = 0$ si u est vide). On dit que u est un *sous-mot* de v s'il existe des mots $t_0, \dots, t_n$ tels que $v = t_0 \cdot u_1 \cdot t_1 \cdot u_2 \cdots t_{n-1} \cdot u_n \cdot t_n$.

Remarques

⇨ Autrement dit, u est un sous-mot de v s'il existe une fonction d'extraction $\varphi : [1 \dots |u|] \to [1 \dots |v|]$ strictement croissante telle que $u_i = v_{\varphi(i)}$ pour tout $i \in [1 \dots |u|]$.

⇨ Dit encore autrement, u est un sous-mot de v si u est une suite extraite de v.

⇨ v est un sous-mot de v, avec φ l'identité (ou, ce qui revient au même, avec tous les t_i égaux à ε).

⇨ ε est un sous-mot de v, avec φ l'unique application de l'ensemble vide dans $[1 \dots |v|]$ ($n = 0$, $t_0 = v$).

⇨ Plus généralement, si u est un facteur de v, u est un sous-mot de v. La réciproque est bien sûr fausse.

Exemple

⇨ Les sous-mots de $abcb$ sont $\varepsilon, a, b, c, ab, ac, bc, bb, cb, abc, abb, acb, bcb$ et $abcb$.

Exercice 3

⇨ Soit u un mot de longueur n dont toutes les lettres sont distinctes.

1. Combien u a-t-il de préfixes, de suffixes, de facteurs, de sous-mots ?
2. Que peut-on dire dans le cas où les lettres ne sont plus supposées distinctes ?

2 Compression

2.1 Considérations générales

2.1.1 Compression sans perte

Un processus de compression sans perte d'un texte (au sens large), étant donné un alphabet Σ, est la donnée d'un couple de fonctions $comp, decomp : \Sigma^* \to \Sigma^*$ telles que :

— $decomp \circ comp = \mathrm{Id}_{\Sigma^*}$;

— pour la plupart des mots m qui nous intéressent, $|comp(m)| < |m|$.

Remarques

⇨ Cette définition impose que *comp* soit injective.

⇨ Elle n'impose en revanche pas que *comp* soit surjective (et ce ne sera le plus souvent pas le cas) ; par conséquent, on n'a généralement pas $comp \circ decomp = \mathrm{Id}_{\Sigma^*}$.

⇨ En réalité, *decomp* ne sera généralement pas défini sur Σ^* en entier, mais seulement sur $comp(\Sigma^*)$. Le comportement de *decomp* sur un mot qui n'est pas le résultat d'une opération de compression pourra être quelconque.

⇨ Un point très important est qu'il est impossible de compresser efficacement toutes les entrées. Plus précisément, pour un entier n quelconque, il n'existe pas d'injection de Σ^n dans Σ^p avec $p < n$. Par conséquent, il y aura toujours des mots m tels que $|comp(m)| \geqslant |m|$; en réalité, si $|comp(m)| < |m|$ pour certains m, on aura même nécessairement $|comp(m)| > |m|$ pour d'autres mots m.

⇨ Ce qui nous intéresse dans un algorithme de compression, ce n'est pas d'arriver à compresser efficacement une entrée quelconque (c'est impossible), mais d'arriver à compresser efficacement une entrée *typique* : par exemple, un texte écrit dans une langue, ou du code exécutable sur une machine, ou un ensemble de triangles décrivant la géométrie d'une scène… Dans tous ces cas, il y a des régularités dans l'entrée que l'on peut exploiter pour parvenir à compresser.

Exemple

⇨ Les deux commandes suivantes créent chacune un fichier de 10 mégaoctets : `random.bin` constitué d'octets (pseudo)-aléatoires et `zero.bin` constitué intégralement de zéros.

```
$ dd if=/dev/urandom of=random.bin bs=10MB count=1
1+0 records in
1+0 records out
10000000 bytes (10 MB, 9,5 MiB) copied, 0,36532 s, 27,4 MB/s
$ dd if=/dev/zero of=zero.bin bs=10MB count=1
1+0 records in
1+0 records out
10000000 bytes (10 MB, 9,5 MiB) copied, 0,00967416 s, 1,0 GB/s
```

On compresse indépendamment chacun de ces fichiers avec l'outil `zip` :

```
$ zip -r random.zip random.bin
  adding: random.bin (deflated 0%)
$ zip -r zero.zip zero.bin
  adding: zero.bin (deflated 100%)
```

`zip` nous indique qu'il a réussi à économiser 0% de place dans le cas de `random.bin` et 100% dans le cas de `zero.bin`. Bien évidemment, il s'agit de valeurs approchées. On peut récupérer les tailles exactes (en octets) :

```
$ du -b *.zip
10001700  random.zip
9885      zero.zip
```

On voit donc que la taille du fichier constitué de zéros a été divisée par plus de mille, alors que celle du fichier aléatoire a légèrement augmenté après compression.

2.1.2 Compression avec pertes

Pour une méthode de compression avec pertes, on n'impose plus que $decomp(comp(m)) = m$, mais seulement que $decomp(comp(m))$ « ressemble » à m. On peut donner quelques exemples de formats de compression avec pertes :

— MP3 pour le son ;
— `jpg` pour les images ;
— la quasi-totalité des *codecs* pour la vidéo.

On peut définir la « ressemblance » mathématiquement (comme une distance dans un certain espace), mais fondamentalement ce qui nous intéresse dans ce cas est qu'un humain ne voie pas, ou pas trop, la différence entre le fichier compressé et le fichier initial. Nous aurons l'occasion d'expérimenter sur la compression avec pertes d'images, mais dans ce chapitre nous n'étudierons que des méthodes de compression sans perte.

2.2 Code de Huffman

Le principe de l'algorithme a été vu en TP : on suppose ici que l'arbre de Huffman associé à l'alphabet de départ a été construit, et l'on suppose pour simplifier que l'alphabet de départ est $[0 \dots 255]$ (autrement dit, on compresse des octets).

On utilise le type suivant pour représenter un arbre de Huffman :

```
type arbre =
  | Feuille of int (* on suppose que l'étiquette est entre 0 et 255 *)
  | Noeud of arbre * arbre
```

La construction de l'arbre à partir d'un fichier est faite par les fonctions ci-dessous :

```
let table_occurrences (nomdefichier : string) : int array =
    let fichier = open_in_bin nomdefichier in
    let occurrences = Array.make 256 0 in
    begin
     try
        while true do
            let c = input_byte fichier in
            occurrences.(c) <- occurrences.(c) + 1;
        done
      with End_of_file -> ()
    end;
    close_in fichier;
    occurrences

let foret (occs : int array) : arbre list =
  let rec aux k =
    if k >= Array.length occs then []
    else if occs.(k) > 0 then (Feuille k, occs.(k)) :: aux (k + 1)
    else aux (k + 1) in
  aux 0

let construit_arbre (table_occs : int array) : arbre =
  let file = PrioQ.of_list (foret table_occs) 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 (Noeud (c, c'), fr + fr')
  done;
  let arbre, _ = PrioQ.extract_min file in
  arbre

let table_code (arbre : arbre) : bool list array =
  let t = Array.make 256 [] in
  let rec remplit_tab noeud pref =
    match noeud with
    | Feuille c ->
       t.(c) <- List.rev pref
    | Noeud (ga, dr) ->
       remplit_tab ga (false :: pref);
       remplit_tab dr (true :: pref) in
```

```
remplit_tab arbre [];
t
```

Dans l'optique d'une utilisation réelle, il reste deux problèmes à résoudre :
— stocker l'arbre « à plat » dans un fichier – on parle de *sérialisation* ;
— écrire (et lire) des codes de longueur variable dans un fichier. Cela nécessite de voir le fichier comme un flux de bits et non un flux d'octets, ce qui demande un certain travail.

2.2.1 Sérialisation de l'arbre

La manière la plus simple de sérialiser l'arbre est de stocker son parcours en profondeur dans l'ordre préfixe. On indique la présence d'un nœud interne par un octet à zéro, la présence d'une feuille par un octet à un, et dans le cas d'une feuille on stocke son étiquette immédiatement après cet octet à un. Autrement dit :
— *serialise*(`Noeud(g, d)`) = `0` *serialise*(`g`) *serialise*(`d`) ;
— *serialise*(`Feuille c`) = `1 c`

Exercices 4

⇨ 1. Donner la sérialisation de l'arbre suivant :
`Noeud(Noeud(Feuille 12, Noeud(Feuille 7, Feuille 8)), Feuille 40)`.
2. Donner l'arbre dont la sérialisation est :
`0 0 1 17 1 18 0 1 1 0 1 30 1 20`.
3. Combien d'octets la version sérialisée de l'arbre occupera-t-elle dans le fichier compressé, en supposant que toutes les valeurs possibles d'un octet apparaissent dans le fichier d'entrée ?

⇨ On rappelle l'existence des deux fonctions suivantes :
— `input_byte : in_channel -> int`
— `output_byte : out_channel -> int -> unit` (l'entier est considéré modulo 256).
Écrire les deux fonctions suivantes, réalisant respectivement la sérialisation et la dé-sérialisation d'un arbre :

```
output_arbre : out_channel -> arbre -> unit
input_arbre : in_channel -> arbre
```

2.2.2 Écriture bit à bit dans un fichier

Le propre de l'algorithme de Huffman est d'associer à chaque caractère un codage binaire de longueur variable. Afin de pouvoir écrire ce codage dans un fichier, il est nécessaire de grouper les bits par paquet de huit (octet en français, *byte* en anglais) : que ce soit en mémoire ou dans un fichier, l'octet est l'unité de base d'information pour un ordinateur.

Ainsi, par exemple, si on a le codage suivant :

c	`a`	`b`	`c`
Otoprule $code(c)$	0	100	101

et qu'on doit encoder `"abbaca"`, on obtient le mot binaire 010010001010 que l'on complète avec des 0 à la fin et que l'on sépare en octets : 01001000 10100000. On obtient donc les deux octets, convertis en décimal, 72 et 160. Ce sont eux qu'on va écrire dans un fichier.

Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit ainsi que le nombre de bits qui ont été accumulés. Dès qu'on accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis réinitialiser ces variables.

Si l'accumulateur vaut $acc = b_0 \dots b_{i-1}$ et qu'on lui ajoute un bit b, on veut obtenir $acc = b_0 \dots b_{i-1} b$. Cela revient à faire l'opération $acc \leftarrow acc + b \cdot 2^i$.

Remarque

⇨ On pourrait tout-à-fait accumuler les bits dans l'autre sens, c'est-à-dire passer de acc à $2 \cdot acc + b$.

Exercice 5

⇨ On considère le type suivant :

```
type out_channel_bits = {
    o_fichier : out_channel;
    mutable o_accumulateur : int;
    mutable o_bits_accumules : int
}
```

Le champ `o_accumulateur` correspond au acc ci-dessus, et le champ `o_bits_accumules` au nombre de bits contenus dans l'accumulateur (autrement dit, au i ci-dessus). La fonction suivante correspond à l'ouverture d'un fichier :

```
let open_out_bits fn =
    {o_fichier = open_out_bin fn;
     o_accumulateur = 0;
     o_bits_accumules = 0}
```

Écrire une fonction `output_bit : out_channel_bits -> bool -> unit` qui traite l'écriture d'un bit en mettant à jour l'accumulateur et le nombre de bits accumulés, et en écrivant un octet dans le canal de sortie le cas échéant.

Il faut encore s'occuper du dernier octet potentiellement incomplet. Si l'on dispose de k bits à la fin de l'écriture, il faut les compléter avec $8 - k$ zéros pour écrire un octet. Avec notre ordre d'écriture des bits, cela ne change en fait pas la valeur de l'accumulateur. Cependant, ces zéros ne sont pas significatifs et il faudra tenir compte de ce fait à la lecture. Pour cela, on ajoute un dernier octet indiquant le nombre de zéros de *padding* présents dans l'avant-dernier octet du flux.

```
let close_out_bits f =
    if f.o_bits_accumules = 0 then
      output_byte f.o_fichier 0
    else begin
        let padding = 8 - f.o_bits_accumules in
        output_byte f.o_fichier f.o_accumulateur;
        output_byte f.o_fichier padding
    end;
    close_out f.o_fichier
```

Pour la lecture, on maintient de même un accumulateur et un nombre de bits accumulés. Quand on nous demande un bit :

— si l'on ne dispose d'aucun bit accumulé, on lit un octet du fichier (et l'on dispose maintenant de 8 bits, sauf cas particulier traité plus bas) ;

— ensuite, on récupère le bit de poids faible de l'accumulateur, et l'on décale ensuite les autres bits. Autrement dit, si $acc = b_0 \dots b_{i-1}$, on veut renvoyer b_0 et remplacer acc par $b_1 \dots b_{i-1}$.

Il faut traiter correctement les zéros de *padding* ajoutés en fin de fichier. Pour cela, il est nécessaire de savoir si l'on est en train de lire l'avant-dernier octet (celui qui a été « rembourré ») : on calcule la taille du fichier à l'ouverture et l'on vérifie à chaque lecture d'octet. Si l'on est en train de lire cet octet rembourré, il suffit de lire l'entier suivant pour savoir combien de bits sont à ignorer.

On donne le type et les fonctions d'ouverture et de fermeture :

```
type in_channel_bits = {
    i_fichier : in_channel;
    mutable i_accumulateur : int;
    mutable i_bits_accumules : int;
    i_taille : int
}

let open_in_bits fn =
    let fichier = open_in_bin fn in
    {i_fichier = fichier;
     i_accumulateur = 0;
     i_bits_accumules = 0;
     i_taille = in_channel_length fichier}

let close_in_bits f = close_in f.i_fichier
```

Exercice 6

⇨ Écrire la fonction `input_bit : in_channel_bits -> bool` qui permet de lire un bit du flux.

2.2.3 Compression et décompression d'un octet

Nous pouvons maintenant écrire les deux fonctions permettant respectivement :

— de compresser un octet du flux d'entrée, c'est-à-dire d'écrire sur le flux de sortie les bits correspondant au code de Huffman de l'octet en question ;

— de décompresser un octet, c'est-à-dire de lire des bits du flux compressé jusqu'à avoir obtenu le code de Huffman d'un octet, et d'écrire alors cet octet sur le flux correspondant au fichier décompressé.

Exercice 7

⇨ 1. Écrire une fonction

```
compresse_byte : out_channel_bits -> bool list array -> int -> unit
```

qui prend en entrée le canal de sortie, la table de codes (telle que produite par la fonction `table_code`) et un entier entre 0 et 255 et écrit la série de bits correspondant au code de l'entier dans le canal de sortie.

2. Écrire une fonction

```
decompresse_byte : in_channel_bits -> arbre -> int
```

qui lit décode un octet depuis le canal d'entrée fourni (en utilisant la représentation arborescente fournie du code de Huffman).

On dispose maintenant de tous les outils nécessaires pour écrire un programme complet. La fonction principale est donnée :

```
let main () =
  let error_and_exit () =
    let invocation = Sys.argv.(0) in
    Printf.eprintf "Usage : \n";
    Printf.eprintf "%s compress <input-file> <compressed-output-file>\n"
       invocation;
    Printf.eprintf "%s decompress <compressed-file> <output-file>\n" invocation;
    exit 1 in
  if Array.length Sys.argv < 4 then error_and_exit ();
  let commande = Sys.argv.(1) in
  let nom_in = Sys.argv.(2) in
  let nom_out = Sys.argv.(3) in
  if commande = "compress" then compresse_fichier nom_in nom_out
  else if commande = "decompress" then decompresse_fichier nom_in nom_out
  else error_and_exit ()

let () = main ()
```

Exercice 8

⇨ Écrire les deux fonctions manquantes :

```
compresse_fichier : string -> string -> unit
decompresse_fichier : string -> string -> unit
```

2.3 Algorithme de Lempel-Ziv-Welch

L'idée de l'algorithme LZW (Lempel-Ziv-Welch) est de construire au fur et à mesure de la lecture du texte un tableau contenant des motifs déjà rencontrés (des facteurs du mot à compresser). Quand on rencontre un motif déjà vu, on le code avec une référence vers ce tableau (l'indice de la case, tout simplement) ; quand on rencontre un motif pour la première fois, on l'ajoute au tableau. On peut tout de suite faire deux remarques :

— cet algorithme permet de compresser un *flux*, il ne nécessite pas de commencer par lire l'intégralité du fichier avant le début de la compression ;

— le dictionnaire que l'on se constitue au fur et à mesure de la compression (le tableau de motifs) n'est pas transmis : comme les règles pour l'ajout et le référencement de motifs sont non ambiguës, il pourra être reconstitué lors de la décompression.

On se fixe une longueur de code d ($d = 12$ est assez typique) : on peut alors avoir une table t de longueur 2^d, et les codes feront tous d bits (un code est simplement un indice dans la table, comme dit plus haut).

2.3.1 Compression

L'algorithme de compression fonctionne alors comme suit :

— on initialise les 256 premières valeurs de la table en posant $t[x] = x$ pour $0 \leqslant x < 256$;

— on maintient une variable m correspondant au motif que l'on est en train de lire ;

— m est initialisée avec un motif vide ;

— on lit ensuite le texte à partir du premier octet ; quand on lit un octet x :

  — soit mx est présent dans la table, et alors on fait $m \leftarrow mx$;

  — soit mx n'est pas présent, dans ce cas on émet le code $t[m]$, on fait $m \leftarrow x$ et l'on ajoute une entrée pour mx dans la table (sauf si elle est pleine) ;

— quand il n'y a plus d'octet à lire, on émet le code $t[m]$ (avec le m que l'on a à cet instant).

Exemple

⇨ Considérons la séquence `0 1 0 1 2 0 1 1 0 1` d'octets en entrée. Le déroulé de l'algorithme est le suivant :

Étape	Octet	$m \mid x$	Présent	Nouvelle entrée	Code émis
Otoprule 1	0	| 0	Oui		
2	1	0 | 1	Non	t[256] = 0 1	(0) = 0
3	0	1 | 0	Non	t[257] = 1 0	(1) = 1
4	1	0 | 1	Oui		
5	2	0 1 | 2	Non	t[258] = 0 1 2	(0 1) = 256
6	0	2 | 0	Non	t[259] = 2 0	(2) = 2
7	1	0 | 1	Oui		
8	1	0 1 | 1	Non	t[260] = 0 1 1	(0 1) = 256
9	0	1 | 0	Oui		
10	1	1 0 | 1	Non	t[261] = 1 0 1	(1 0) = 257
11	EOF	1 |			(1) = 1

La séquence « compressée » sera donc `0 1 256 2 256 257 1`, avec chacun des 7 blocs codé sur d bits. Pour une valeur typique de d, on utilisera sans doute plus de bits que la séquence initiale : c'est normal, la séquence est clairement trop petite pour que la méthode ait un réel intérêt.

On remarque que l'on a besoin à chaque étape de vérifier si un certain motif est présent dans la table (et de trouver son code si c'est le cas). Une solution simple est de maintenir un dictionnaire ayant comme clé les motifs (ce seront des séquences d'octets, donc typiquement des chaînes de caractères) et comme valeurs les codes. Ce n'est pas le plus efficace cependant : nous verrons (peut-être) une autre manière de procéder en TP.

2.3.2 Décompression

Pour décompresser, on ne dispose pas de la table : il faut donc la reconstituer au fur et à mesure du décodage, en déterminant les opérations de compression qui ont nécessairement conduit à l'émission des codes lus. On initialise la table avec les valeurs des 256 octets, puis l'on traite le premier code lu (qui est nécessairement inférieur à 256) en émettant cet octet. À partir du second code lu :

— on lit un code n où $n < |t|$ et $t[n] = xm'$ (avec x un octet et m' une séquence) ;
— on émet xm' ;
— on ajoute à la table mx, où c est le précédent code lu et $t[c] = m$.

Exercice 9

⇨ Simuler le processus de décompression sur le flux `0 1 256 2 256 257 1`.

Il reste cependant un cas à traiter : celui où on lit un code n avec $n = |t|$ – autrement dit, on lit un code que l'on n'a pas encore ajouté à la table. Regardons ce qui ce passe sur un exemple.

Exercice 10

⇨ Simuler la compression puis la décompression de la séquence suivante : `ABABABA`.

Plus précisément, on lit un code c qui correspond à une séquence u, puis un code c' inconnu pour l'instant, qui code une certaine séquence u'. Réfléchissons à comment cela peut se produire (du côté de la compression) :

— on sait que c' vient d'être ajouté après émission du code pour u, donc c' code $u' = ux$ pour un certain octet x ;
— d'un autre côté, x suit immédiatement u dans l'entrée, donc u' commence par x ;
— ainsi, $u' = ux$ avec x le premier octet de u qui est connu.

On en déduit la procédure complète suivante :

— on initialise la table avec une entrée pour chaque octet ;
— on maintient une variable c contenant le dernier code lu ;
— pour chaque code n que l'on lit :
  — si $n < |t|$ et $t[n] = xm$ (avec x un octet), on écrit xm en sortie ;
  — si $n = |t|$, on écrit $t[c]x$ en sortie, où x est le premier octet de $t[c]$;
  — dans les deux cas, on ajoute $t[c]x$ à la table, et l'on fait $c \leftarrow n$.

Exercice 11

⇨ Compresser puis décompresser la séquence suivante : `ABABCABCDABCDABCDA`.

3 Recherche de chaîne

Une des opérations les plus courantes sur le texte est de rechercher les occurrences d'une chaîne de caractères dans une autre chaîne de caractères (plus longue). C'est ce qu'on fait avec un simple Ctrl-F, mais c'est également le

problème central de l'alignement de séquences en génétique, par exemple. La version la plus simple du problème se formalise ainsi :

Définition 3.1: Occurrences d'un mot dans un autre

Soient $u = u_0 \dots u_{k-1}$ et $v = v_0 \dots v_{n-1}$ deux mots sur un même alphabet Σ. Une *occurrence* de u dans v est un entier i tel que $v_i \dots v_{i+k-1} = u$.
Le problème de la recherche d'occurrences consiste, étant donnés u et v, a déterminer toutes les occurrences i de u dans v.

Remarques

⇨ Deux occurrences peuvent tout à fait se chevaucher : par exemple, *abbab* possède deux occurrences dans *abbabbab* (une à $i = 0$ et une à $i = 3$).

⇨ L'extension la plus classique de ce problème est de chercher les occurrences d'un *motif* plutôt que d'une chaîne. On peut par exemple chercher toutes les occurrences de `a_a`, où `_` est un caractère quelconque, ou les motifs constitués d'une suite de chiffres quelconques. Nous reviendrons largement sur cette question en deuxième année quand nous traiterons les expressions régulières et les automates finis.

⇨ Une autre extension envisageable est de chercher simultanément les occurrences d'un ensemble fini de mots (plus efficacement qu'en faisant successivement une recherche pour chacun des mots).

Exercices 12

⇨ *Algorithme naïf en C* : On rappelle la représentation d'une chaîne de caractères en C : un bloc contigu de `char` non nuls, terminé par un `char` nul (la longueur de la chaîne étant le nombre de `char` non nuls).

1. Écrire une fonction `is_prefix` qui détermine si la chaîne `substring` est un préfixe de la chaîne `text`.

```
bool is_prefix(char *text, char *substring);
```

2. Écrire une fonction `first_occurrence` qui détermine la première occurrence d'une sous-chaîne dans un texte. On renverra -1 s'il n'existe aucune occurrence.

```
int first_occurrence(char *text, char *substring);
```

3. Quelle est la complexité temporelle de `first_occurrence` en fonction de la longueur n de `text` et k de `substring` ?
4. Écrire une fonction `occurrences` qui renvoie toutes les occurrences d'une sous-chaîne dans un texte. On commencera par déterminer un prototype possible pour cette fonction (plusieurs choix sont bien sûr possibles).

⇨ *Algorithme naïf en OCaml* : On pourrait bien sûr adapter facilement les fonctions C de l'exercice précédent au type `string` de OCaml (et ce serait en général la bonne manière de procéder). À titre d'entraînement, on peut cependant utiliser des listes de caractères pour représenter les chaînes.

1. Écrire une fonction `is_prefix` telle que l'appel `is_prefix text substring` renvoie `true` si et seulement si `substring` est un préfixe de `text`, et donner sa complexité.

```
is_prefix : 'a list -> 'a list -> bool
```

2. Écrire une fonction `occurrences` telle que l'appel `occurrences text substring` renvoie la liste des occurrences de `substring` dans `text`, dans l'ordre, et donner sa complexité.

```
occurrences : 'a list -> 'a list -> int list
```

3.1 Algorithme de Rabin-Karp

Comme nous venons de le voir, l'algorithme naïf a une complexité temporelle en $O(kn)$ (et une complexité spatiale constante). Le problème est que l'on effectue (environ) n comparaisons de chaînes de caractères sur des chaînes de longueur k. Le principe de l'algorithme de Rabin-Karp est de limiter le nombre de comparaisons coûteuses à effectuer, en commençant à chaque fois par comparer une *empreinte* (un *hash*) du facteur `text[i : i + k]` avec la chaîne cherchée. Une empreinte étant un entier, on peut faire une telle comparaison en temps $O(1)$, puis :

— si les empreintes diffèrent, on est sûr qu'il n'y a pas d'occurrence en i ;
— si les empreintes coïncident, on peut être en présence d'une occurrence ou d'une collision – il faut comparer les chaînes pour trancher.

Pour que l'algorithme apporte un gain réel, deux conditions doivent être réunies :

— le pourcentage de collisions doit être faible ;
— le calcul de l'empreinte doit être rapide.

Il y a clairement une tension entre ces deux objectifs : pour limiter les collisions, il est indispensable que l'empreinte dépende des k caractères du facteur considéré, mais d'un autre côté le calcul de l'empreinte doit être en O(1). Si l'on écrit la version la plus « naturelle » de l'algorithme, on échouera nécessairement :

Algorithme 1 Algorithme de Rabin-Karp, **tentative infructeuse**

fonction RabinKarp(t, s)
  $n \leftarrow |t|$
  $k \leftarrow |s|$
  $target \leftarrow hash(s)$
  $occurences \leftarrow ()$
  pour $i \in [0 \dots n-k]$ **faire**
    si $hash(t[i : i+k]) = target$ et $t[i : i+k] = s$ **alors**
      $occurrences \leftarrow occurrences, i$
    fin si
  fin pour
  renvoyer *occurrences*
fin fonction

Remarques

- ⇨ On utilise ici le caractère séquentiel du « et » : le test $t[i : i+k] = s$ n'est effectué que si $hash(t[i : i+k]) = target$.
- ⇨ Malgré cela, cette version ne **peut pas convenir** : pour que $hash(t[i : i+k])$ dépende de tous les caractères de $t[i : i+k]$, il est indispensable de parcourir ce facteur en entier, ce qui impose une complexité au moins proportionnelle à k pour ce calcul. On a donc au minimum une complexité proportionnelle à nk, et donc pas meilleure que celle de l'algorithme naïf. D'un autre côté, si l'empreinte ne dépend que d'un nombre constant de caractères, on ne peut pas espérer limiter les collisions dans le cas général.

Pour résoudre ce problème, il faut utiliser un *rolling hash* (une « empreinte tournante ») : l'idée est de choisir une fonction de hachage permettant de calculer en temps constant l'empreinte de $x_1 \dots x_k$ si l'on connaît celle de $x_0 \dots x_{k-1}$. Le plus simple est de choisir une base b :

$$h(x_0 \dots x_{k-1}) = x_0 b^{k-1} + x_1 b^{k-2} + \cdots + x_{k-1} b^0$$

On a alors :

$$\begin{aligned} h(x_1 \dots x_k) &= x_1 b^{k-1} + x_2 b^{k-2} + \cdots + x_{k-1} b^1 + x_k b^0 \\ &= b \cdot \left(h(x_0 \dots x_{k-1}) - b^{k-1} x_0 \right) + x_k \end{aligned}$$

Remarques

- ⇨ Autrement dit, on considère $x_0 \dots x_{k-1}$ comme un nombre écrit en base b, chiffre le plus significatif en premier.
- ⇨ Pour la base, on prendra typiquement $b = 256$ si l'on travaille octet par octet (b doit être supérieur à la plus grande valeur possible pour un x_i).

Avec cette version, il ne peut pas y avoir de collision. Cependant, le passage d'une empreinte à une autre n'est en fait pas en temps constant : en effet, la taille de $h(x_0 \dots x_{k-1})$ n'est pas bornée. En pratique, avec $b = 2^8$, on dépasse la capacité d'un entier 64 bits dès que la longueur du motif recherché dépasse 7… On travaille donc en fait modulo p, où p est un entier que l'on choisira :

- premier (cela limite les collisions) ;
- grand (*idem*) ;
- mais pas trop (il faut que l'empreinte rentre dans un entier machine, et on ne veut pas de dépassement de capacité dans les calculs intermédiaires).

On obtient alors l'algorithme suivant :

Remarques

- ⇨ Si l'on veut être sûr de ne pas avoir de dépassement de capacité, il suffit que pb rentre dans le type entier utilisé.
- ⇨ En moyenne, on aura environ n/p collisions si les données sont aléatoires.
- ⇨ En revanche, un adversaire connaissant b et p peut sans grande difficulté choisir un texte t et un motif s donnant de très nombreuses collisions, dans l'optique d'une attaque par déni de service par exemple. Pour contrer cela, la solution classique est de choisir aléatoirement la fonction de hachage dans une famille ayant les propriétés souhaitées.

Proposition 3.2: Complexité de l'algorithme de Rabin-Karp

En notant c la proportion de collisions, la complexité temporelle de l'algorithme de Rabin-Karp est en $O(n + cnk)$. Sauf cas pathologique, c est de l'ordre de $1/p$, et comme p peut être choisi de l'ordre de 2^{60}, on a $cn \ll 1$.

Algorithme 2 Algorithme de Rabin-Karp

On fixe *a priori* une base b et un nombre premier p, légèrement inférieur au plus grand entier machine.
fonction RabinKarp(t, s)
  $n \leftarrow |t|$
  $k \leftarrow |s|$
  si $k > n$ **alors**
    renvoyer ()
  fin si
  $d \leftarrow b^{k-1} \mod p$
  $target \leftarrow 0$
  $h \leftarrow 0$
  pour $i \in [0 \dots k-1]$ **faire**
    $target \leftarrow b \cdot target + s[i] \mod p$
    $h \leftarrow b \cdot h + t[i] \mod p$
  fin pour
  $occurences \leftarrow ()$
  pour $i \in [0 \dots n-k]$ **faire**
    si $h = target$ et $t[i : i+k] = s$ **alors**
      $occurrences \leftarrow occurrences, i$
    fin si
    si $i + k < n$ **alors**
      $h \leftarrow b \cdot (h - d \cdot t[i]) + t[i+k] \mod p$
    fin si
  fin pour
  renvoyer *occurrences*
fin fonction

Remarques

⇨ Comme nous le verrons, on dispose d'algorithmes plus efficaces que Rabin-Karp pour rechercher une chaîne dans un texte.

⇨ Rabin-Karp a cependant deux avantages :
— il est simple à implémenter ;
— il s'étend facilement à la recherche d'un *ensemble* de motifs dans un texte.
Pour le deuxième point, il suffit de disposer d'une structure d'ensemble raisonnablement efficace (table de hachage typiquement, ou ABR équilibré) : on calcule alors l'ensemble S des empreintes des différents motifs au moment de l'initialisation, et l'on remplace le test $h = target$ par $h \in S$.

3.2 Algorithme de Boyer-Moore

Pour rechercher les occurrences d'un motif m de taille k dans un texte t de taille n, l'algorithme naïf teste tous les alignements possibles : il y en a $n - k + 1$. L'algorithme de Boyer-Moore permet de sauter certains de ces tests au prix d'un pré-traitement du motif m, et en comparant $t[i : i + k]$ et m *de droite à gauche*. Il existe deux versions de cet algorithme :
— la version originale de Boyer et Moore (1977), qui utilise deux tables de décalage (et deux règles permettant d'éliminer des alignements) ;
— une version simplifiée, connue sous le nom de Boyer-Moore-Horspool (1980), qui ne conserve qu'une règle d'élimination, et donc qu'une table de décalage. Cette version est nettement plus simple à comprendre et est la seule à être explicitement au programme : nous commencerons donc par elle.

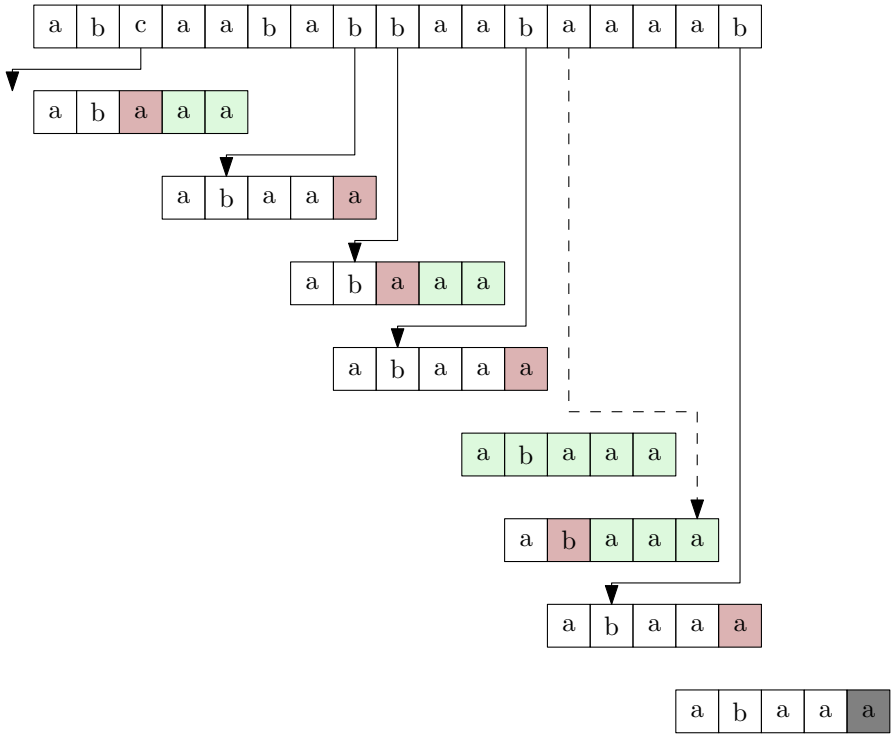
3.2.1 Algorithme de Boyer-Moore-Horspool

Principe On cherche la première occurrence du motif m dans le texte t. On commence par tester s'il y a une occurrence en 0 (c'est-à-dire si $m = t[0 : k]$), et pour faire ce test on commence à comparer les caractères *à partir du dernier caractère du motif.*
— Si l'on ne trouve aucun caractère différent entre motif et facteur du texte, on a une occurrence. Si l'on ne veut que la première, on a fini, sinon on recommence en cherchant une occurrence en 1.
— Sinon, soit i le plus grand indice tel que $m_i \neq t_i$, soit $x = t_i$ et $droite(x)$ le plus grand indice tel que $m_j = x$ (-1 si aucun indice ne convient). On peut être sûr qu'il n'y a pas d'occurrence du motif commençant avant $i - droite(x)$. En effet, en commençant à $s < i - droite(x)$, il faudrait que $m_{i-s} = t_i = x$, or $i - s > i + (droite(x) - i) = droite(x)$, ce qui contredirait la maximalité de $droite(x)$. Notons que $i - droite(x)$ peut être négatif si jamais il y a une occurrence de x à droite de i dans m : il faut donc se décaler de $\max(1, i - droite(x))$.

Exercice 13

⇨ On a dit qu'on posait $droite(x) = -1$ si x n'avait aucune occurrence dans m. La formule pour le décalage reste-t-elle valable dans ce cas ?

Le principe est beaucoup plus facile à comprendre sur une figure :



Algorithme de Boyer-Moore-Horspool

Par rapport à l'algorithme naïf, on voit bien qu'on économise parfois des comparaisons. Ce n'est toutefois pas systématique : essentiellement, l'algorithme sera d'autant plus rapide que l'on effectuera souvent des sauts importants.

Exercice 14

⇨ Donner un exemple de texte et de motif pour lequel le nombre de comparaisons de caractères effectués sera en $Omega(kn)$.

Mise en œuvre Pour que l'algorithme soit efficace, il est indispensable de pouvoir déterminer $droite(x)$ (avec les notations ci-dessus) sans parcourir tout le motif. Pour cela, on précalcule les $droite(x)$ pour tous les caractères x de l'alphabet, ce qui peut se faire sans problème en temps $O(|m|)$.

Exercices 15

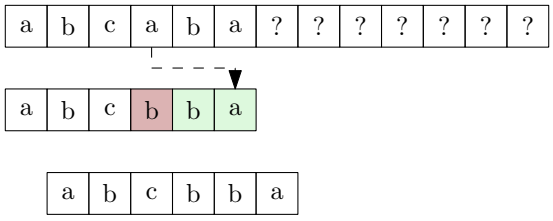
⇨ Écrire une fonction OCaml `make_bct` (pour *bad character table*) prenant en entrée une chaîne `s` et renvoyant un tableau `bct` de longueur 256 contenant tel que, pour tout caractère `x`, on ait `bct.(int_of_char x)` égal au $droite(x)$ défini ci-dessus.

```
1 make_bct : string -> int array
```

⇨ Écrire une fonction `boyer_moore_horspool` prenant en entrée un texte et un motif à chercher, et renvoyant la liste des occurrences du motif dans le texte, dans l'ordre (sous forme d'une liste d'entiers).

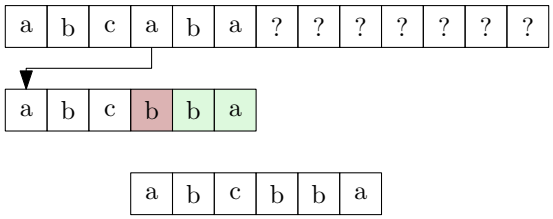
```
1 boyer_moore_horspool : string -> string -> int list
```

Amélioration possible Considérons la situation suivante :



En appliquant l'algorithme décrit jusqu'à présent, on constate en lisant a dans le texte au lieu du b présent en position 3 du motif que $droite(a) = 5 > 3$. On ne peut donc pas effectuer de saut, et l'on se déplace seulement de 1.

Cependant, il est clair que l'on aurait pu effectuer un saut plus important : en effet, l'occurrence de a la plus à droite *parmi celle situées avant 3* est en 0. En alignant cette occurrence avec le a lu dans le texte, on ne risque pas de rater une occurrence du motif : on peut donc effectuer un saut de $3 - 0 = 3$.



Le problème est que l'on ne peut alors plus parler de $droite(x)$, mais de $droite(x, i)$:

$$S(x, i) = \{l \in [0 \dots i-1] \mid m_l = x\}$$

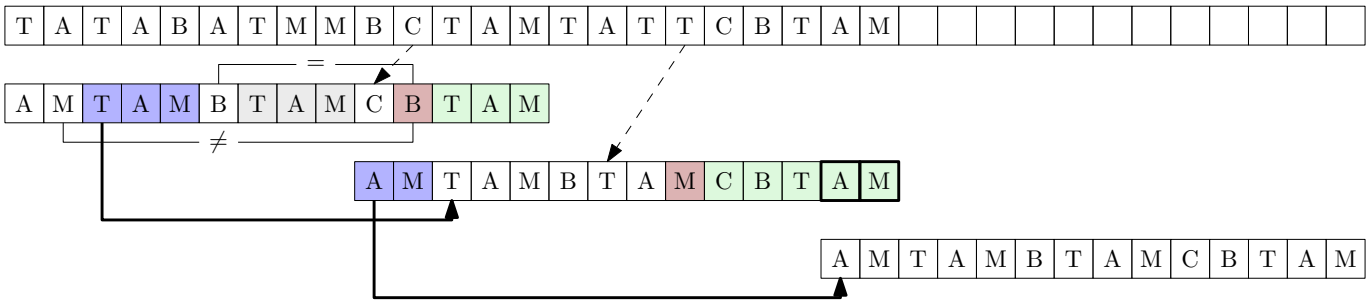
$$droite(x, i) = \begin{cases} -1 & \text{si } S(x, i) = \varnothing \\ \max S(x, i) & \text{sinon} \end{cases}$$

Exercice 16

⇨ Proposer une ou plusieurs solutions pour le pré-calcul et le stockage des $droite(x, i)$. Discuter de la complexité spatiale et temporelle de ces solutions, en distinguant pour la complexité temporelle le pré-calcul de l'accès ultérieur à $droite(x, i)$.

3.2.2 Algorithme de Boyer-Moore complet

Dans l'algorithme de Boyer-Moore à proprement parler, on ne s'intéresse pas seulement au caractère qui a fait échouer la comparaison, mais aussi au suffixe du motif qui est déjà correctement aligné :



Algorithme de Boyer-Moore

— À la première étape, la comparaison échoue pour le B en rouge sur la figure. On a lu un C à la place, et la règle du mauvais caractère nous amènerait à un saut de 1 (puisqu'il y a un C juste à gauche du B dans le motif). Cependant, on sait que l'on a réussi à lire TAM, puis échoué à lire un B. On cherche donc dans le motif l'occurrence de TAM qui n'est pas immédiatement précédée par un B la plus à droite, et on l'aligne avec le TAM lu dans le texte.

— À la deuxième étape, la règle du mauvais caractère nous donne un saut de 2. Mais on a lu avec succès CBTAM dans le texte, avant d'échouer à lire M. On fait donc la même recherche qu'à l'étape précédente : une occurrence de CBTAM qui ne soit pas précédée d'un M. Il n'y en a aucune : on cherche alors le plus grand préfixe du motif qui soit un suffixe de CBTAM. En l'occurrence, c'est AM : on décale en conséquence.

L'idée n'est pas très compliquée, mais est-il possible de faire cela de manière efficace ? La réponse est oui, avec un pré-traitement en temps $O(|m|)$ et un stockage également en $O(|m|)$, mais c'est assez loin d'être évident. Nous y reviendrons.