I Chaînes de caractères

Une chaîne de caractères est un tableau dont les éléments sont des `char` (caractères) et se termine par le caractère spécial `'\0'`. En OCaml, on n'utilise pas `'\0'`.

```c
// C
char s[] = "mpi"; // \0 est ajouté automatiquement
printf("%c", s[0]); // affiche le 1er caractère
s[2] = '2'; // modifie un caractère
strlen(s); // nombre de caractères
for(int i = 0; s[i] != '\0'; i++)
    // parcourir s
```

```ocaml
(* OCaml *)
let s = "mpi"
s.[0] (* 1er caractère *)
s.[2] <- '2' (* ERREUR : String est immuable en OCaml *)
let n = String.length s (* nombre de caractères *)
if s = "mp2i" then ... (* comparaison en O(n) *)
for i = 0 to n - 1 do
```

II Recherche d'un mot dans un texte

Si $m = m_1 ... m_k$ est un mot, on note $m[i : j]$ le mot $m_i \dots m_{j-1}$. On dit que m est un facteur de t s'il existe i, j tels que $t[i : j] = m$. Les lettres de m doivent donc apparaître consécutivement dans t.

Recherche d'un mot

Entrée : Deux mots `m` et `t`.
Sortie : `m` est-il un facteur de `t` ?

Complexité des algorithmes classiques de recherche de mot, où $k = |m|$ et $n = |t|$:

Algorithme	Complexité	Prétraitement	Mémoire	Méthode
Naïf	$O(nk)$	0	0	Fenêtre glissante
Rabin-Karp	$O(n)$	0	0	Fonction de hachage
Boyer-Moore	$O(nk)$	$O(k)$	$O(k)$	Décalage
KMP	$O(n)$	$O(k)$	$O(k)$	Automate

La complexité de Rabin-Karp est donnée en moyenne (à cause de la complexité moyenne O(1) des tables de hachage). Boyer-Moore est efficace en pratique.

II.1 Algorithme naïf par fenêtre glissante

```c
bool is_substring(char* m, char* t) {
    int k = strlen(m), n = strlen(t);
    for(int i = 0; i < n - k + 1; i++)
        for(int j = 0; t[i + j] == m[j]; j++)
            if(j == k - 1)
                return true;
    return false;
}
```

Complexité :

II.2 Algorithme de Rabin-Karp ♡

L'algorithme de Rabin-Karp utilise une fonction de hachage h rapide à calculer (si possible O(1)) et, pour chaque indice i de t :

- Compare $h(m)$ et $h(t[i : i + k])$.
- Si ces deux valeurs sont égales, on compare m et $t[i : i + k]$ lettre par lettre, en $O(k)$.

Ainsi, si $h(m) \neq h(t[i : i + k])$, on sait que $m \neq t[i : i + k]$ et on gagne du temps d'exécution.

Remarque : Comme h n'est *a priori* pas injective, il peut y avoir des collisions, c'est-à-dire $h(m) = h(t[i : i+k])$ et $m \neq t[i : i+k]$.

Étant donné un mot $u = u_0...u_{k-1}$, un entier b (le nombre de caractères possibles) et un entier q, on peut utiliser $h(u) \in \{0, ..., q-1\}$ défini par :

$$h(u) = \sum_{i=0}^{k-1} \text{code}(u_i) b^{k-1-i} \mod q$$

où $\text{code}(u_i)$ est le code de la lettre u_i (par exemple, le code ASCII).

On peut calculer chaque $h(t[i : i+k])$ par « rolling hash », en calculant $h(t[i+1 : i+1+k])$ à partir de $h(t[i : i+k])$ en O(1). Par exemple, $h(u_1...u_k) = b \times (h(u_0...u_{k-1}) - \text{code}(u_0)b^{k-1}) + \text{code}(u_k) \mod q$.

```
let hash b q s =
    let rec aux i p =
        if i = -1 then 0
        else ((Char.code s.[i])*p + aux (i-1) ((p*b) mod q)) mod q in
    aux (String.length s - 1) 1

let rabin_karp m t =
    let k, n = String.length m, String.length t in
    let q = 3719 in (* prime number for modulo *)
    let b = 256 in (* number of characters (basis) *)
    let p = pow b (k - 1) q in (* maximum power of b *)
    let h_w = hash b q m in
    let rec search i h =
    if h = h_w && m = String.sub t i k then i
    else if i >= n - k then -1
    else
        let h_ = (b*(h - p*Char.code t.[i]) + Char.code t.[i + k]) mod q in
        search (i + 1) (if h_ >= 0 then h_ else h_ + q) in
    search 0 (hash b q (String.sub t 0 k))
```

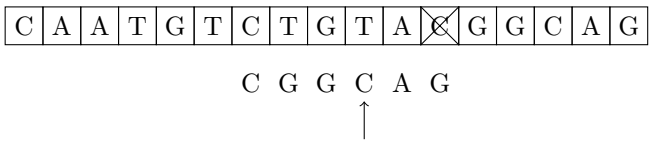
où `pow a n q` renvoie $a^n \mod q$.

III Algorithme de Boyer-Moore-Horspool ♡

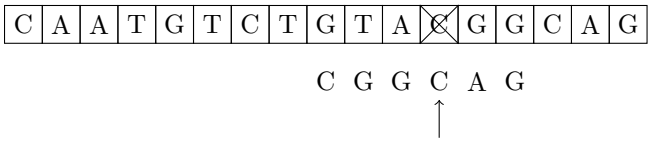
L'algorithme de Boyer-Moore-Horspool utilise un dictionnaire `d` tel que `d[c]` soit l'indice de la première apparition de `c` dans `m`. Exemple : si `w = "CGGCAG"` alors `d` a les associations `('A', 1)`, `('C', 2)`, `('G', 3)` et `'T'` n'est pas une clé de `d`.

L'algorithme considère alors les lettres de t et les compare aux lettres de m, de droite à gauche. Dès qu'il y a une différence, on décale d'un nombre de caractères donné par `d` et on reprend la comparaison du début.

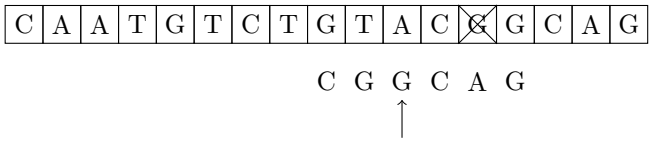
T n'apparaît pas dans le mot : on décale de k



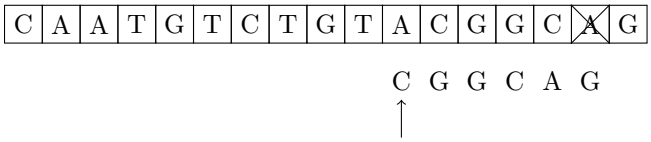
Le C ne correspond pas



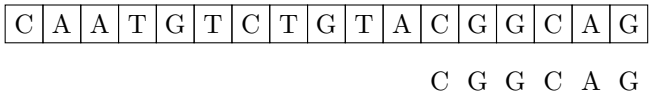
On décale de 2 pour avoir un C



L'avant-dernière lettre ne correspond pas : on décale de 2 pour avoir un G



La dernière lettre ne correspond pas : on décale de 1 pour avoir un A



On a trouvé m !

```
let boyer_moore_horspool t w =
    let k, n = String.length w, String.length t in
    let module M = Map.Make(Char) in
    let rec make_d i =
        if i = k then M.empty
        else make_d (i+1) |> M.add w.[k - i - 1] i in
    let d = make_d 1 in

    let rec search i j = (* teste si w[:k-j] = t[i-k:i-j] *)
        if i >= n then -1
        else if j = k then i - k + 1
        else if w.[k - j - 1] = t.[i - j] then search i (j + 1)
        else match M.find_opt t.[i - j] d with
            | Some s -> search (s + i) 0
            | None -> search (i + k - j) 0 in
    search (k - 1) 0
```

où on a utilisé le module `Map` qui implémente un dictionnaire persistant (par arbre binaire de recherche) :

```
(* exemple d'utilisation du module Map *)
module M = Map.Make(Char) (* dictionnaire dont les clés sont des caractères *)
let d = M.empty (* dictionnaire vide *)
let d2 = M.add 'a' 1 d (* ajoute une clé 'a' avec valeur 1 *)
M.find_opt 'a' d2 (* renvoie Some 1 *)
M.find_opt 'b' d2 (* renvoie None *)
```

Complexité :

Exercice 1.

1. Quelle est la complexité de Boyer-Moore-Horspool dans le meilleur cas ?
2. Montrer que l'algorithme de Boyer-Moore-Horspool a une complexité en $\Theta(nk)$ dans le pire des cas.

IV Compression de texte

Définition : Algorithme de compression sans perte

Un algorithme de compression sans perte consiste à définir deux fonctions $f : \Sigma^* \longrightarrow \Sigma^*$ (codage) et $g : \Sigma^* \longrightarrow \Sigma^*$ (décodage) telles que $g \circ f = id$ et en espérant que $|f(m)|$ soit petit par rapport à $|m|$.

Cette définition implique que f est bijective (il existe un unique décodage).

Théorème

Il n'existe pas de fonction injective $f : \Sigma^* \longrightarrow \Sigma^*$ telle que $|f(m)| < |m|$ pour tout $m \in \Sigma^*$.

Il n'existe donc pas d'algorithme de compression sans perte qui diminue toujours la taille. Par contre, on peut faire en sorte que ce soit le cas pour un ensemble de mots particuliers (textes en français où il y a des répétitions, par exemple).

IV.1 Compression de Huffman ♡

Idée du codage de Huffman : utiliser moins de bits pour les lettres qui apparaissent souvent.

1. Prétraitement : On construit un arbre binaire T, appelé arbre de Huffman, dont les feuilles sont les lettres.

2. Compression : On code chaque lettre c par une suite de 0 et 1 correspondant au chemin (0 : gauche, 1 : droite) de la racine à la feuille contenant c dans T.
3. Décompression : On décode une suite de 0 et 1 en parcourant le chemin correspondant dans T.

L'algorithme a besoin d'estimer la fréquence d'apparition $f(c)$ de chaque lettre c.
Pour cela, on peut d'abord lire une première fois le texte à coder et compter le nombre d'occurrences de chaque caractère.

On construit un arbre dont les étiquettes (lettres) sont aux feuilles et le chemin de la racine à une lettre donne son codage.

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

```
type 'a tree = F of 'a | N of 'a tree * 'a tree
```

Construction de l'arbre de Huffman

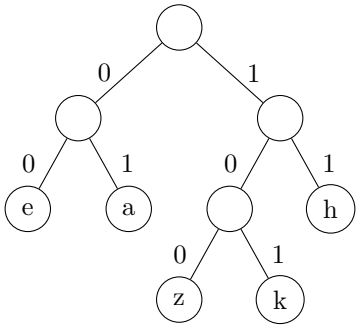
`q` ← file de priorité contenant `F c` avec la priorité $f(c)$, pour chaque lettre c
Tant que `q` contient ≥ 2 éléments **:**
  Extraire de `q` les 2 arbres `t1` et `t2` de priorités min `f1` et `f2`
  Ajouter à `q` l'arbre `N(t1, t2)` avec la priorité `f1 + f2`
Renvoyer Le seul arbre de `q`

Exercice 2.
Quel arbre de Huffman obtient-on avec les lettres suivantes ?

Lettre	a	b	c	d	e
Fréquence	20	15	7	14	44

Exercice 3.
Quelle est la complexité de la construction de l'arbre de Huffman, en fonction du nombre n de lettres ? On précisera le type de file de priorité utilisé.

Une fois l'arbre construit, on construit un tableau (ou dictionnaire) associant à chaque lettre une liste de bits. On peut alors compresser un texte en remplaçant chaque lettre par sa liste de bits.



Dans cet arbre de Huffman, a est codé par `[0; 1]`, z par `[1; 0; 0]`...

Théorème

Le codage de Huffman est un codage préfixe, ce qui signifie qu'aucun code d'une lettre n'est préfixe d'un autre code.

Preuve : On ne peut pas prolonger un chemin menant à une feuille pour obtenir le chemin d'une autre feuille.

Théorème : (admis)

Le codage de Huffman est optimal dans le sens où il minimise la longueur moyenne de codage.

Pour décoder une suite de bits, il suffit de parcourir l'arbre suivant le bit lu (0 = gauche, 1 = droite) et, dès qu'on trouve une lettre (une feuille), de l'afficher puis de revenir à la racine pour le bit suivant.

Pour décoder un fichier compressé par Huffman, il faut stocker l'arbre de Huffman. Pour cela, on peut le sérialiser (le convertir en chaîne/liste de caractères).
Une façon possible est de construire son parcours préfixe en ajoutant un caractère spécial pour reconstruire les feuilles ($*$) et les noeuds (`#`) :

```
let rec serialize_tree = function
  | F c -> ['*'; c]
  | N (g, d) -> '#'::(serialize_tree g)@serialize_tree d
```

Exercice 4.
Écrire une fonction pour désérialiser un arbre de Huffman.

V Lempel-Ziv-Welch (LZW) ♡

Problème : le codage de Huffman demande de calculer la fréquence des lettres du texte entier pour être optimal. Parfois, on a besoin de compresser du texte « en ligne », c'est-à-dire sans attendre d'avoir reçu la totalité.
LZW détermine un codage (dans un dictionnaire d) au fur et à mesure de la lecture du texte. On va coder certains motifs (groupements de lettres consécutives). Il fonctionne bien quand il y a des motifs qui se répètent.

Algorithme LZW

Entrée : Texte s à compresser
Sortie : Liste d'entiers correspondant au codage de s

$d \leftarrow$ dictionnaire initialisé avec un code pour chaque lettre de l'alphabet
Tant que $s \neq \emptyset$ **:**
  Retirer le plus long préfixe w de s qui appartienne à d
  Ajouter $d[w]$ à la liste de retour
  $w' \leftarrow w$ concaténé avec la prochaine lettre de s
  Ajouter un nouveau code pour w' dans d

Exercice 5.
Appliquer cet algorithme sur *barbapapapabap*.