Cette *vingt-deuxième* colle vous fera écrire des fonctions pour la recherche de sous-chaîne dans un texte (méthode naïve et méthode de Boyer-Moore-Horspool) en OCaml, ainsi qu'appliquer à la main la compression et décompression de Lempel-Ziv-Welch (LZW).

On pourra travailler depuis Windows avec `https://BetterOCaml.ml/`.

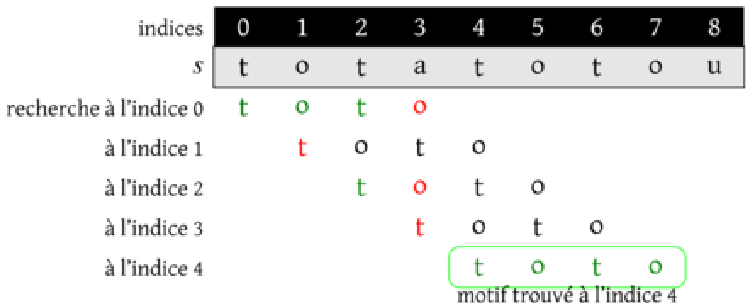
Ex.1 Recherche de sous-mot dans un texte - OCaml (45 minutes)

On s'intéresse dans cette colle à répondre au problème de recherche de sous-mots dans un texte, qui s'exprime comme cela :

- *Entrée* : un texte `t` et un motif `m` écrits sur le même alphabet Σ (typiquement, l'alphabet ASCII à 256 symboles différents), tels que `|m|` $\leqslant$ `|t|`.
- *Sortie* : toutes les positions i entre 0 et `|t| - |m|` telles que `m` $=$ `t`$[i : i +$ `|m|`$[$, ou la liste vide `[]` si aucune position valide n'est trouvée.

Méthode naïve

1. Rappeler comment obtenir la longueur d'une chaîne de caractère en OCaml (une `string`), en écrivant une fonction `longueur (m:string) : int` utilisant une fonction de la bibliothèque standard `String`.
2. Écrire une fonction `occurrence (t:string) (m:string) (i:int) : bool` qui vérifie si une occurrence du mot `m` se trouve en position `i` du texte `t`, c'est-à-dire si `t[i + j] = m[j]` pour tout `j = 0 ... |m|-1`. On pensera à tester si `i+j` est un indice valide pour `t`.



3. Avec `occurrence`, écrire une fonction `recherche_naive (t:string) (m:string) : int list` qui renvoie la liste des `i` tels qu'une position `i = 0 .. |t|-|m|` est trouvée à laquelle `m` apparaît comme sous-mot de `t`, ou qui renvoie `[]` si `m` n'apparaît comme sous-mot de `t` à aucune position.
4. Quelle est la complexité temporelle de cette fonction `recherche_naive` en fonction de `|t|` taille du texte et `|m|` taille du mot ?
5. Et quelle est sa complexité mémoire ?
6. Faire quelques tests de `recherche_naive` à la fin de votre code.

Méthode de Boyer-Moore-Horspool

Le principe de l'algorithme de Boyer-Moore-Horspool est d'effectuer une recherche du motif comme précédemment pour la recherche naïve, mais en partant de la fin du motif.

On part toujours du début du texte `t` (avec une variable `i` croissante), mais on compare les caractères du motif `m` et ceux de la fenêtre du mot `t`$[i : i +$ `|m|`$[$ depuis la fin (avec une variable `j` décroissante).

On va alors tenter de trouver des suffixes de plus en plus grand du motif.

— Si on trouve ainsi le motif, on ajoute la position `i` trouvée à la liste des positions trouvées, et on continue avec `i+1`, sauf si on est à la dernière valeur possible de `i` auquel cas on s'arrête.

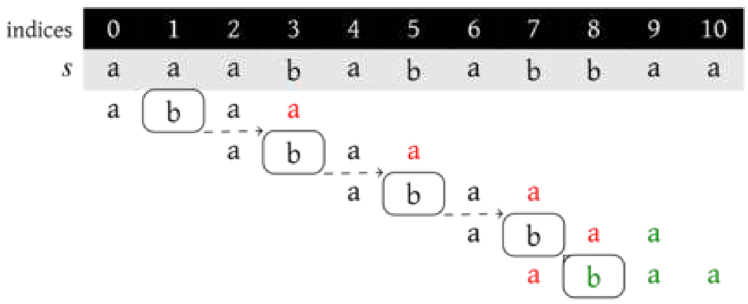
— Sinon, c'est qu'on a lu dans le texte `t` un mot de la forme xm' où m' est un suffixe strict du mot recherché m, mais xm' n'en est pas un (x est une seule lettre). Si la lettre x n'est pas présent dans m, alors on peut relancer la recherche juste après x dans t. Si x est présent dans m, on peut relancer la recherche en alignant ce caractère avec sa position la plus à droite dans m.

Remarque :

Il faut tenir compte différemment du dernier caractère du motif, car il n'est pas utile de le réaligner. On considère alors, quand elle existe, l'occurrence *précédente* de ce caractère.

On obtient ainsi une stratégie de saut, qui en cas d'échec relance la recherche plus loin. La méthode naïve correspond à la stratégie de saut qui relance la recherche toujours à l'indice `i+1`.

Voici un premier exemple où on effectue une recherche de `m` $= abaa$ dans le mot `t` $= aabababbaa$. Cette stratégie a permis d'éviter une recherche inutile à partir de l'indice 1.



Implémentation par table de saut

Pour réaliser ces sauts, on construit une table `droite`, indexée par Σ, et telle que `droite[c]` indique l'indice de l'occurrence la plus à droite dans le motif `m` du caractère `c`, en ignorant le dernier caractère du motif.

Ainsi, dans l'exemple précédent du motif `m` $= abaa$, on obtient la table suivante :

c	'a'	'b'	'c'	...
`droite[c]`	2	1	$\varnothing$	...

On a indiqué ici $\varnothing$ quand un caractère de Σ n'est pas présent dans le motif `m` (par exemple `'c'` est absent de `m = "abaa"`), car il peut être présent dans le texte `t`.

Cette table contient donc $|\Sigma|$ éléments. On peut la réaliser par un tableau direct de taille $|\Sigma|$, étant donné un ordre d'énumération de l'alphabet Σ. Par exemple pour l'alphabet ASCII, cet ordre est l'ordre naturel qui identifie un caractère à son code ASCII entier, comme par exemple `'a'` qui est l'entier 97.

7. Définir une variable globale entière `taille_alphabet` valant 256.
8. Écrire une fonction de signature `calculer_droite (m:string) : int array` qui crée (avec `Array.make taille_alphabet (-1)`) un tableau `droite` de `taille_alphabet` entiers, d'abord initialisés tous à `-1` (pour représenter le $\varnothing$ de la table). Ensuite il faut remplir la table avec l'algorithme suivant :

```
Pour i = 0 à |m| - 2 Faire
    j = |m| - 2 - i
    c = Char.code m[j]      // conversion d'un char en un int en OCaml
    Si droite[c] < 0 Alors
        droite[c] = j
    Fin Si
Fin Pour
```

   Attention à penser ne calculer la longueur du motif, `|m|`, qu'une seule fois, et pas à chaque passage de boucle.
9. Quelle est la complexité mémoire de cette fonction `calculer_droite`, et sa complexité temporelle, en fonction de `|m|` et $|\Sigma|$?

Algorithme de Boyer-Moore-Horspool

Afin d'implémenter l'algorithme de Boyer-Moore-Horspool lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer. Si à la position `i` on a un échec après avoir lu le caractère `c`, où `droite[c]` contient la valeur entière `k`, alors on a trois cas :

a. Si `k = -1`, cela signifie que le motif `m` ne pourra jamais être trouvé tant que ce caractère `c` sera présent (vu que `c` n'est *pas* dans le motif `m`). On relance donc la recherche juste après l'indice `i + j + 1`. (Cf. Figure 1.)

0	1	2	3	4	5	6	7	8
a	b	b	a	a	d	a	c	a
d	a	c						
			d	a	c			

Figure 1 – Cas a. quand on peut décaler la recherche au maximum à $i + j + 1$.

b. Si `k >= j`, cela signifie que `c` est présent plus à droite dans le motif, donc aligner cette occurrence ne permettrait pas d'avancer la recherche. En effet, rien ne nous permet de savoir si `c` est présent ou non ailleurs dans le motif `m`, on relance alors (prudemment) la recherche à l'indice `i+1`. (Cf. Figure 2.)

0	1	2	3	4	5	6	7	8
a	c	a	c	a	d	a	c	a
c	a	c						
	c	a	c					

Figure 2 – Cas b. quand on doit décaler la recherche prudemment à $i + 1$.

c. Sinon, on veut aligner ce symbole `c` avec le caractère correspondant du motif `m`, si on relance à l'indice `i'` on souhaite ainsi avoir `i' + k = i + j`, donc on relance à l'indice `i' = i + j - k`.

Dans les deux cas a. et c., on a relancé la recherche à un indice plus grand que `i + 1`, donc on a économisé quelques comparaisons du motifs avec une fenêtre `t[i : i + |m|[`, en comparaison à la méthode naïve.

10. Écrire une fonction `recherche_Boyer_Moore_Horspool (t:string) (m:string) : int list` qui implémente cette stratégie. On commencera par calculer une fois la table `droite` avant d'effectuer les recherches.
11. Quelle sont les complexités mémoire et temporelle de cette fonction `recherche_Boyer_Moore_Horspool`, en fonction de `|m|`, `|t|` et $|\Sigma|$?
12. Faire quelques tests de `recherche_Boyer_Moore_Horspool` à la fin de votre code.

Ex.2 Application de l'algorithme de Lempel-Ziv-Welch (LZW) à la main (10 minutes)

Comme en cours vendredi dernier, on s'intéresse à la méthode de compression (et décompression) de Lempel-Ziv-Welch (LZW). On considère l'alphabet ASCII Σ de taille 256, et on considère un mot $m \in \Sigma^*$.

13. Dérouler à la main la compression de LZW sur le mot `m = "ABABBAAC"`. Combien d'entiers sont nécessaires pour encoder ce mot `m`, et de combien de symboles ASCII est-il constitué ? La méthode de compression est-elle utile ici ?
14. Trouver un exemple de mot m, de longueur au moins $|m| \geqslant 26$, telle que sa compression par la méthode LZW n'apporte aucun gain. (sans preuve)
15. Dérouler à la main la décompression de LZW sur le code `c = [65, 32, 66, 65, 83, 257, 65, 258, 82]`. Vous aurez besoin de savoir les correspondances suivantes entre lettres ASCII et leur valeur entière : `'A' = 65`, `'B' = 66`, `' ' = 32` (espace), `'R' = 82`, `'S' = 83`. Quelle est la taille du mot `m` obtenu en décodant ? La méthode de compression avait-elle été utile ici ?

Note : vous pourrez aller tester vos résultats sur `https://fr.planetcalc.com/9069/` une fois chez vous.