Chapitre 20 - Recherche textuelle

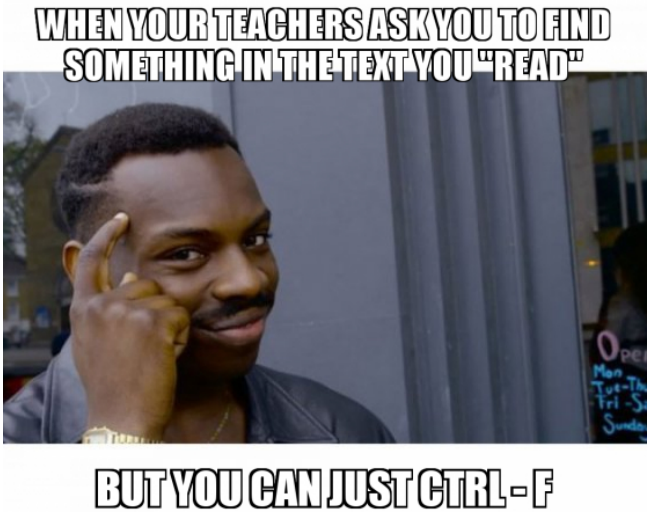
Objectifs :

- ▷ Comprendre le principe de la recherche textuelle.
- ▷ Etudier l'algorithme de *Boyer-Moore* pour la recherche d'un motif dans un texte.

1 Introduction

Tout logiciel de traitement de texte possède une fonction de recherche textuelle, permettant de trouver un mot ou une partie d'un mot (chaine ou sous-chaine). Il en est de même pour les navigateurs Internet.

Ainsi, le raccourci clavier `CTRL+F` permet de rechercher dans la page la chaine `"mot"`.



Les algorithmes qui permettent de trouver une **sous-chaine** de caractères dans une chaine de caractères plus grande sont des "grands classiques" de l'algorithmique. On parle aussi de recherche d'un **motif** dans un texte.

Voici un exemple. Soit le texte suivant :

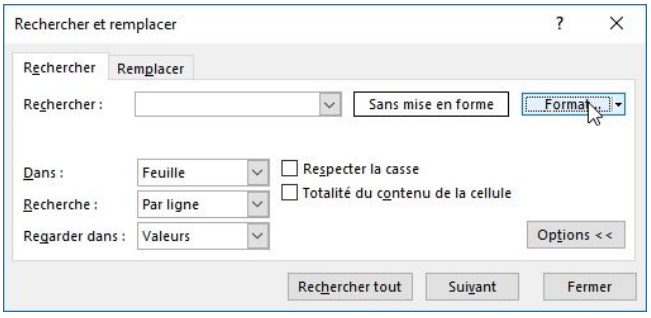
"Les sanglots longs des violons de l'automne blessent mon coeur d'une langueur monotone. Tout suffocant et blême, quand sonne l'heure, je me souviens des jours anciens et je pleure."

Question : le motif `"vio"` est-il présent dans le texte ci-dessus, si oui, en quelle(s) position(s) ? La numérotation d'une chaine de caractères commence à zéro et les espaces sont considérés comme des caractères.

Réponse : on trouve le motif `"vio"` en position `23`

La recherche de sous-chaîne dans une chaîne donnée est un problème important de l'informatique, puisque ses applications sont nombreuses :

- pour les **moteurs de recherches**,
- pour l'utilisation des outils de recherche dans les **traitements de textes** ou **navigateurs web**,
- dans le domaine de **bio-informatique**, pour la recherche de séquences données de nucléotides.



2 La bio-informatique

Comme son nom l'indique, la bio-informatique est issue de la rencontre de l'**informatique** et de la **biologie**. La récolte des données en biologie a connu une très forte augmentation ces 30 dernières années.

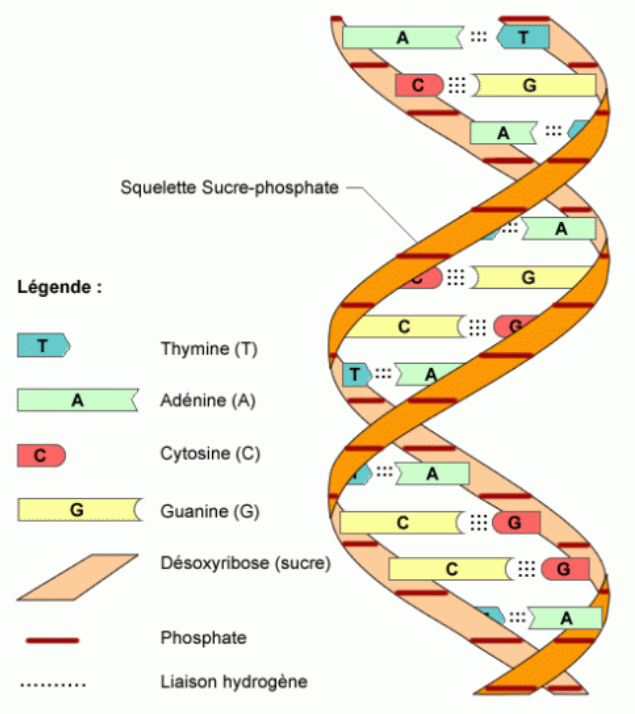
Pour analyser cette grande quantité de données de manière efficace, les scientifiques ont de plus en plus recourt au traitement automatique de l'information, c'est-à-dire à l'informatique.

2.1 Analyse de l'ADN

Comme vous le savez déjà, l'information génétique présente dans nos cellules est portée par les molécules d'ADN.

Les molécules d'ADN sont, entre autres, composées de bases azotées ayant pour noms :

— *Adénine* (représenté par un `A`)
— *Thymine* (représenté par un `T`)
— *Guanine* (représenté par un `G`)
— *Cytosine* (représenté par un `C`)



L'information génétique est donc très souvent représentée par de très longues chaines de caractères, composées des caractères `A`, `T`, `G` et `C`. Exemple : `CTATTCAGCAGTC`...

Il est souvent nécessaire de détecter la présence de certains enchainements de bases azotées.

Par exemple, on peut se poser les questions suivantes :

— Trouve-t-on le triplet `ACG` dans le brin d'ADN suivant ?
— Si c'est le cas, en quelle position ?

```
CAAGCGCACAAGACGCGGCAGACCTTCGTTATAGGCGATGATTTCGAACCTACTAGTGGGTCTCTTAGGCCGAGCGG
TTCCGAGAGATAGTGAAAGATGGCTGGGCTGTGAAGGGAAGGAGTCGTGAAAGCGCGAACACGAGTGTGCGCAAGCG
CAGCGCCTTAGTATGCTCCAGTGTAGAAGCTCCGGCGTCCCGTCTAACCGTACGCTGTCCCCGGTACATGGAGCTAA
TAGGCTTTACTGCCCAATATGACCCCGCGCCGCGACAAAACAATAACAGTTTGCTGTATGTTCCATGGTGGCCAATC
CGTCTCTTTTCGACAGCACGGCCAATTCTCCTAGGAAGCCAGCTCAATTTCAACGAAGTCGGCTGTTGAACAGCGAG
GTATGGCGTCGGTGGCTCTATTAGTGGTGAGCGAATTGAAATTCGGTGGCCTTACTTGTACCACAGCGATCCCTTCC
CACCATTCTTATGCGTCGTCTGTTACCTGGCTTGGCAT
```

Nous allons commencer par le premier algorithme qui nous vient à l'esprit : l'**algorithme naïf**.

3 La recherche naïve

Un premier algorithme dit **"naïf"** qui utilise une **méthode itérative brute** permet de rechercher une sous-chaine dans une chaine de caractères en parcourant le texte de gauche à droite. Il faut avancer dans le texte **caractère par caractère**, puis si le caractère considéré correspond au premier caractère du mot, nous comparerons les caractères suivants à ceux du mot.

3.1 Algorithme

Algorithme 1 : Fonction recherche_naive(texte, motif)

Entrées : Texte et Motif
Sortie : Liste des positions

$m \leftarrow$ longueur du motif `// Longueur du motif`
$n \leftarrow$ longueur du texte $-m+1$ `// Nombre d'itérations`
positions $\leftarrow$ [] `// Liste des positions`

pour *i variant de* 0 *à* n
  si *motif* $=$ *texte*$[i...i+m]$ **alors**
  | ajouter i à la liste *positions*
fin
return *positions*

3.2 Exemple 1

L'une des citations les plus connues du chimiste et philosophe *Antoine Laurent de Lavoisier* est la suivante : `"rien ne se perd, rien ne se crée, tout se transforme"`.

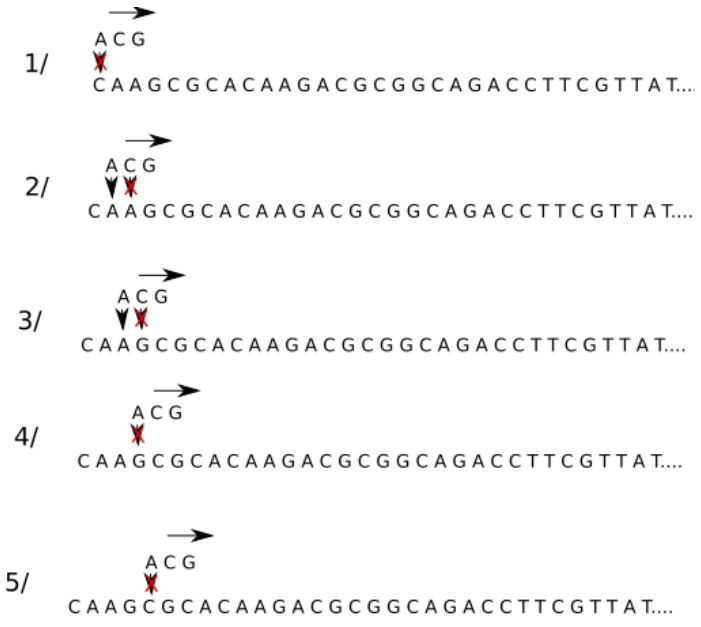
Cherchons la chaine `"rien"` dans le texte de cette citation.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
texte	r	i	e	n		n	e		s	e		p	e	r	d	,		r	i	e	n		n	e		s	e		c	r	é	e	,		t	o	u	t		s	e		t	r	a	n	s	f	o	r	m	e	.
chaine														r	i	e	n																																				

Le motif est trouvé dans le texte à deux reprises, on rajoute les index (`0` et `17`) à la liste des index puis on le décale d'un cran vers la droite.

La recherche se termine lorsqu'on atteint le rang : *longueur du texte - longueur du motif*

3.3 Exemple 2



Par exemple, on cherche la séquence `ACG` dans le brin d'ADN ci-dessus :

Déroulé (étape par étape)

1. On place le motif recherché au même niveau que les 3 premiers caractères de notre chaine, le premier élément du motif **ne correspond pas** au premier élément de la chaine (`A` et `C`), on décale le motif d'un cran vers la droite.
2. Le premier élément du motif **correspond** au premier élément de la chaine (`A` et `A`) mais **pas le second** (`C` et `A`), on décale d'un cran vers la droite.
3. Le premier élément du motif **correspond** au premier élément de la chaine (`A` et `A`) mais **pas le second** (`C` et `G`), on décale d'un cran vers la droite.
4. Le premier élément du motif **ne correspond pas** au premier élément de la chaine (`A` et `G`), on décale d'un cran vers la droite.
5. Le premier élément du motif **ne correspond pas** au premier élément de la chaine (`A` et `C`), on décale d'un cran vers la droite.
6. On continue le processus jusqu'au moment où les 3 éléments du motif correspondent avec les 3 éléments de la chaine situés au même niveau.

Visualiser étape par étape le document suivant présentant l'algorithme naïf sur un autre exemple d'ADN.

3.4 Coût de l'algorithme naïf

Cet algorithme naïf peut, selon les situations demander un **très grand nombre de comparaisons**, ce qui peut entraîner un très long temps de calcul avec des chaines très très longues.

On suppose que l'on a une chaine de longueur `N` et un mot de longueur `n`.

L'exécution est relativement **lente**, la fonction doit tester `N-n` positions dans la chaine et pour chacune effectuer jusqu'à `n` comparaisons.

La complexité de cet algorithme est donc dans le pire des cas $O((N-n) \times n)$, c'est-à-dire une complexité **quadratique**.

Nous allons voir que l'algorithme de *Boyer-Moore* permet de faire mieux en termes de comparaisons à effectuer. Il est beaucoup plus efficace en utilisant la recherche à l'envers à partir de la fin du mot.

4 L'algorithme de Boyer-Moore-Horspool

Robert S. Boyer et *J. Strother Moore* sont connus pour leur contribution majeure à l'élaboration de l'algorithme de recherche *Boyer-Moore*. Diplômés de l'*Université de Berkeley* en Californie, ces chercheurs ont marqué l'histoire avec leur algorithme novateur, publié en 1977 sous le titre *"A Fast String Searching Algorithm"*.

Cette méthode efficace pour trouver toutes les occurrences d'un motif donné dans un texte est devenu un pilier fondamental dans de nombreux domaines de l'informatique.

Il existe une version simplifiée, développée en 1980 par *Nigel Horspool*, informaticien et professeur émérite à l'*Université de Victoria* en Colombie-Britannique (Canada), que nous allons présenter ici.

L'algorithme de *Boyer-Moore-Horspool* (simplification de l'algorithme de *Boyer-Moore*), effectue la **vérification à l'envers**, c'est-à-dire en partant non pas du début mais de la **fin du motif**.

4.1 Le principe

Etapes de l'algorithme

1. On aligne le mot (ou motif, ou *pattern* en anglais) sous la chaine, en partant de la gauche.
2. On compare les lettres du mots avec celles correspondantes de la chaine, mais en partant de la fin du mot (c'est contre-intuitif).

Trois cas peuvent alors se produire :

— Les lettres du mot et de la chaine **correspondent**, on continue alors en remontant le motif de la droite vers la gauche.

— les lettres **ne correspondent pas**, et la lettre de la chaine n'est pas présente dans le motif. On va alors décaler le motif vers la gauche d'un nombre de saut égal à sa longueur.

— les lettres **ne correspondent pas**, et la **lettre de la chaine est présente dans le motif**. On va alors décaler le motif de manière à faire correspondre la lettre de la chaine à celle du motif.

4.2 Exemple

Prenons un exemple :

Si l'algorithme cherche le mot `RECURSION` dans un texte, il ne teste pas le premier, mais d'abord **le dernier caractère du motif** (ici, le $9^{ème}$ caractère du texte).

Plusieurs situations peuvent alors se présenter :

1. si on trouve un `N` (dernier caractère du motif), on regarde si le $8^{ème}$ caractère est un `O`, puis le $7^{ème}$ un `I`,... jusqu'au 1^{er}.
   — Si tous les caractères correspondent, on a trouvé le motif.
   — Si les autres caractères ne correspondent pas, on en déduit que le motif ne peut commencer au mieux qu'en $10^{ème}$ position du texte : l'algorithme décale le motif d'une longueur complète, et "saute" alors directement à la $18^{ème}$ position pour rechercher un `N`.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	.....
texte	**U**	**N**	**D**	**E**	**R**	**S**	**T**	**A**	**N**	D		R	E	C	U	R	S	I	O	N	.....
motif	R	E	C	U	R	S	I	O	N												
									→	R	E	C	U	R	S	I	O	N			

   Le `N` correspond, mais pas les caractères précédents : le motif ne peut donc se trouver que 9 positions (la longueur du motif) plus à droite.
2. si on trouve en $9^{ème}$ position une lettre n'apparaissant pas dans le motif : cette situation est similaire à la précédente, où l'on peut donc aussi décaler le motif d'une longueur complète : ainsi, on ne vérifie qu'un seul caractère au lieu de dix.
3. si on trouve par exemple un `I` au lieu d'un `N` en ^{9ème} position : il se peut que ce soit le `I` de `RECURSION`, auquel cas le `N` serait 2 positions plus à droite : l'algorithme décale alors le motif de 2 positions pour faire "coïncider" le `I` du motif et celui du texte, et recherche alors le `N` (puis éventuellement le `O`, le `I`...).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	.....
texte	U	N	D	E	R	S	T	A	N	**D**		**R**	**E**	**C**	**U**	**R**	**S**	**I**	O	N	.....
motif										R	E	C	U	R	S	I	O	N			
											→	R	E	C	U	R	S	I	O	N	

   Dans cet algorithme, le décalage appliqué au motif n'est donc pas de 1 position, mais dépend du caractère lu : dans le cas de `RECURSION`, `+9` si le caractère est un `N` ou n'est pas dans le motif, `+7` pour un `E`, `+2` pour un `I`,..., *etc.*

   Si une même lettre apparaît plusieurs fois dans le motif, on retiendra logiquement le plus petit décalage (donc celui correspond au caractère le plus à droite du motif).

L'algorithme **décale donc beaucoup plus "vite"** le motif vers la droite, et ce d'autant plus que le motif est long : on arrive donc plus rapidement à la fin du texte, la complexité en temps est bien meilleure que celle de la recherche naïve.

4. Visualiser sur la démo interactive suivante le fonctionnement de l'algorithme de *Boyer-Moore*.

5 Exercices

Exercice 1 : *Fonctions natives de Python*

1. Consulter la documentation de Python sur la méthode `find` des chaînes de caractères.
2. Quel est le rôle de cette méthode ?
3. Tester cette méthode sur les exemples suivants :
   - `'numérique et sciences informatiques'.find('que')`
   - `'numérique et sciences informatiques'.find('nsi')`

Note : Une autre méthode presque identique (`str.index`) lève une erreur lorsque le motif cherché ne se trouve pas dans la chaîne.

Exercice 2 : *Algorithme naïf*

1. Regarder la capsule vidéo suivante (jusqu'à `5'44"`)
2. Résumer le principe de l'algorithme de recherche textuelle naïve.
3. Un outil en ligne permet de visualiser un algorithme dit de "recherche naïve". Utiliser cet outil (en changeant éventuellement le motif et la chaîne).
4. Sans utiliser `find` ou `index`, et à partir de l'*algorithme 1* du cours, écrire une fonction `recherche(motif,chaine)` qui renvoie `True` ou `False` suivant que la chaîne de caractères `motif` se trouve ou non dans `chaine`.
5. Même question mais en renvoyant l'indice de la première position de `motif` dans `chaine` (si elle s'y trouve) et `-1` sinon.
6. Même question mais en renvoyant le nombre d'occurrences de `motif` trouvées dans `chaine`.
7. Même question mais en renvoyant la liste des indices des occurrences de `motif` trouvées dans `chaine`.

Exercice 3 : *Mesure du nombre d'étapes*

Pour répondre aux questions ci-dessous, vous utiliserez les outils interactifs suivants.

On veut rechercher le motif `string` dans la chaîne `stupid_spring_string`.

1. Combien d'étapes sont nécessaires avec l'algorithme naïf ?
2. Appliquer l'algorithme de *Boyer-Moore-Horspool* sur cette recherche. Combien d'étapes sont nécessaires ?
3. Faire varier le motif et la chaîne (très longue) et comparer à nouveau le nombre d'étapes pour chacun des algorithmes. Que peut-on conclure ?

Exercice 4 : *Mesure du temps de recherche*

Le Projet Gutenberg permet de télécharger légalement des ouvrages libres de droits dans différents formats.

Nous allons travailler avec le Tome 1 du roman *Les Misérables* de *Victor Hugo*, à télécharger ici au format `.txt`.

1. Récupérere le texte dans une seule chaîne de caractères en utilisant le code suivant :

```
with open('Les_Miserables.txt') as f:
    roman = f.read().replace('\n', ' ')
```

2. À l'aide du module `time`, mesurer le temps de recherche dans *Les Misérables* d'un mot court, d'une longue phrase (présente dans le texte), d'un mot qui n'existe pas. Que remarquez-vous ?

Exercice 5 : *Algorithme naïf inversé*

Re-écrire l'algorithme de recherche naïve mais en démarrant de la fin du motif et non du début.

Exercice 6 : *Fonction préparatoire*

On va d'abord coder une fonction `dico_lettres` qui prend en paramètre un mot `mot` et qui renvoie un dictionnaire associant à chaque lettre de mot son dernier rang dans `mot`. On exclut la dernière lettre, qui poserait un problème lors du décalage (on décalerait de 0...)

Écrire la fonction `dico_lettres`.

Exemple d'utilisation :

```
>>> dico_lettres("MARINA BAY SANDS")
{'M': 0, 'A': 12, 'R': 2, 'I': 3, 'N': 13, ' ': 10, 'B': 7, 'Y': 9, 'S':
   11, 'D': 14}
```

Exercice 7 : *Algorithme de Boyer-Moore*

Toujours à partir de la vidéo précédente, répondre aux questions ci-dessous :

1. Dans quelle(s) condition(s) de recherche textuelle peut-on décaler la recherche de plus d'un caractère ?
2. En déduire pourquoi la recherche d'un mot dans un texte est un cas particulier permettant d'accélérer notablement le traitement.
3. Décrire le fonctionnement de l'algorithme de *Boyer-Moore*.
4. Expliquer l'avantage du prétraitement dans cet algorithme.
5. Écrire la fonction décrite dans la vidéo permettant de retourner la table des sauts pour une clé donnée. Cette fonction `table_saut(cle)` retournera un dictionnaire dont les clés seront les lettres de la clé, et les valeurs le saut associé.
6. Indiquer le résultat obtenu avec votre fonction pour la clé « `EXCELLENT` ».

Exercice 8 : *Algorithme de Boyer-Moore-Horspool*

En utilisant la fonction `dico_lettres` créée précédemment, compléter le code ci-dessous permettant d'implémenter l'agolrithme de *Boyer-Moore-Horspool*.

```
def dico_lettres(mot):
    ...
    return d

def BMH(texte, motif):
    dico = dico_lettres(motif)
    indices = []
    i = 0
    while i <= ... - ...:
        k = ...
        while k > 0 and texte[...] == motif[...]:
            k = ...
        if k == ...:
            indices.append(...)
            i = ...
        else:
            if ... in dico:
                i += ... - ... - 1
            else:
                i += ...
    return ...
```

Exemple d'utilisation :

```
>>> BMH("une magnifique maison bleue", "maison")
[15]
>>> BMH("une magnifique maison bleue", "nsi")
[]
>>> BMH("une magnifique maison bleue", "ma")
[4, 15]
```

Exercice 9 : *Temps de recherche BMH*

Reprendre les mesures effectuées sur *Les Misérables*, mais cette fois avec l'algorithme *Boyer-Moore-Horspool*. Que remarquez-vous ?

Exercice 10 : *Epreuve pratique 2022 - Sujet n°6 - Exercice 2*

La fonction recherche prend en paramètres deux chaines de caractères `gene` et `seq_adn` et renvoie `True` si on retrouve gene dans `seq_adn` et `False` sinon.

Compléter le code Python ci-dessous pour qu'il implémente la fonction `recherche`.

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = ...
    trouve = False
    while i < ... and trouve == ... :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            ...
        if j == g:
            trouve = True
        ...
    return trouve
```

Exemples :

```
>>> recherche("AATC", "GTACAAATCTTGCC")
True
>>> recherche("AGTC", "GTACAAATCTTGCC")
False
```