Recherche de motifs
Algorithme de BOYER MOORE

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** (sous-chaine) dans un texte (chaine). Voici un exemple :

Soit le texte suivant :
Texte = "Je suis des cours de NSI et je trouve ça vraiment super chouette. »
Question : le motif "super" 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 "super», il est en position 50 (Texte [50 :55] vaut « super »).

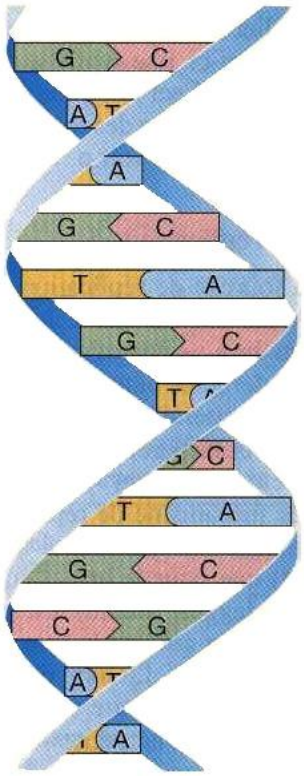
On a tous utilisé une fois dans notre vie la fonction « rechercher » pour trouver un mot dans une page web, ou dans un document texte (le plus souvent, en faisant *ctrl F*). Des algorithmes se cachent derrière, c'est l'objet de notre cours.

Les algorithmes de recherche textuelle sont aussi utilisés en bioinformatique, pour le séquençage des molécules d'ADN :

Petit rappel de bio : une molécule d'ADN contient le code génétique d'un individu. Ce code a un vocabulaire restreint : 4 « lettres », qu'on appelle des bases ou des nucléotides: l'adénine (A), la thymine (T), la guanine (G) et la cytosine (C). L'ADN humain contient plus de 3 milliards de ces bases associées. Notre code génétique s'écrit donc comme une chaine de caractères de 3 milliards de caractères.

….ATTGCACGGTTAACACATGGAGCTATTCCTTAAGG….

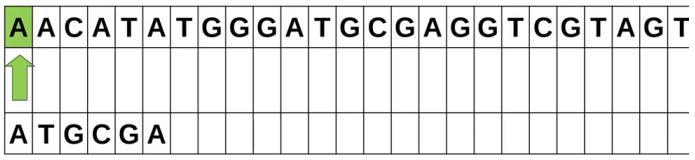
On comprend que les biologistes soient heureux de pouvoir compter sur l'outil informatique, mais vu le nombre de caractères, la complexité des algorithmes à utiliser est un peu essentielle…



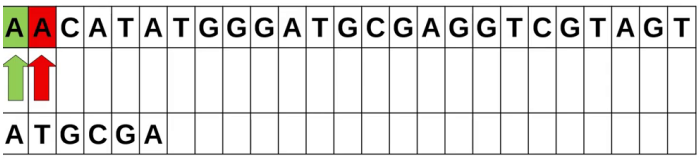
1 – Algorithme « naïf » (appelé aussi algorithme de « force brute ») :

C'est le premier algorithme qui nous vient à l'esprit. Il consiste à parcourir toute la chaine en décalant de 1 à chaque comparaison. Exemple, on cherche le **motif ATGCGA** dans la chaine AACATATGGGATGCGAGGTCGTAGT.

On place le motif au début afin de comparer les 6 caractères du motif aux 6 premiers caractères de la chaine.



On trouve un « match » sur le premier caractère. Cool. Comparons ensuite le second.



Mince, ça ne match pas sur le second, ce n'est pas bon, on décale de 1 le motif et c'est reparti.

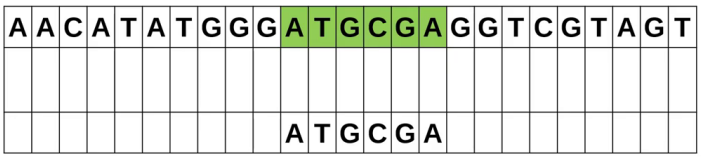
Le premier caractère match, le second mismatch. Ce n'est pas bon. Décalons de 1

Ça dismatch dès le premier, ce n'est pas bon, on décale de un.

Ça match sur les deux premier mais pas sur le troisième.. on décale de 1

...

Et ainsi de suite….



… jusqu'à arriver à un match complet. On retournera alors l'info « on a trouvé le motif au rang 10 »

Cet algorithme naïf peut, selon les situations, demander un très grand nombre de comparaisons (imaginez si à chaque fois on match sur les 5 premiers caractères et dismatch sur le 6ème !), ce qui peut entraîner un très long temps de "calcul" avec des chaines très longues et un motif long.

Sauf si le motif qu'on recherche n'a qu'un seul caractère, **il y a moyen de faire mieux**.

Exemples d'algorithmes naïfs :

Rappelons tout d'abord qu'une chaîne de caractères est un objet de type *str* composé de caractères. Chaque caractère d'une chaîne est repéré par son index dans la chaîne. Les index commencent à 0.
Prenons l'exemple de la chaîne s='abracadabra'. Alors, s[0] a pour valeur le caractère 'a', s[3] a pour valeur le même
caractère et s[4] le caractère 'c'.
La fonction *len* renvoie le nombre de caractères d'une chaîne. Ici, *len(s)* a pour valeur 11.
Considérons deux chaînes de caractères, l'une appelée texte, l'autre appelée motif, on cherche s'il existe une occurrence du motif dans le texte, c'est-à-dire un index i tel que *texte[i:i + len(motif)] == motif*.

Un algorithme nous renseignant sur le nombre de fois qu'on trouve un motif dans une chaine peut s'écrire ainsi :

VAR texte, motif en chaine de caractères
VAR long_txt, long_motif, i, nb_trouve en entier

nb_trouve = 0
long_txt = longueur de la chaine texte
long_motif = longueur du motif

POUR i allant de 0 à (long_txt – long_motif)
    Si motif == texte[i : i+long_motif] :
        nb_trouve = nb_trouve + 1
    FIN SI
FIN POUR
RENVOYER nb_trouve

Un algorithme nous renvoyant la liste des indices de la chaine où se trouve un motif peut s'écrire ainsi :

VAR texte, motif en chaine de caractères
VAR liste_d_index en liste
VAR long_txt, long_motif, i en entier

POUR i allant de 0 à (long_txt – long_motif)
    Si motif == texte[i : i+long_motif] :
        AJOUTER i à liste_d_index
    FIN SI
FIN POUR
RENVOYER liste_d_index

! Attention pour l'implémentation en Python :

De la même manière que *range(0,3)* ira de 0 à 2, *texte[0:3]* correspondra à *texte[0]+ texte[1]+ texte[2]*

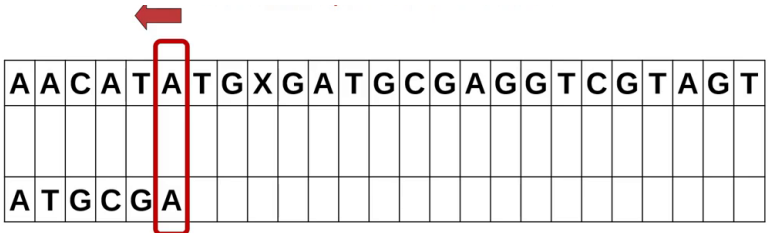
On a dit qu'on pouvait mieux faire, faisons le :

2- Algorithme de Boyer-Moore (version simplifiée de Horspool)

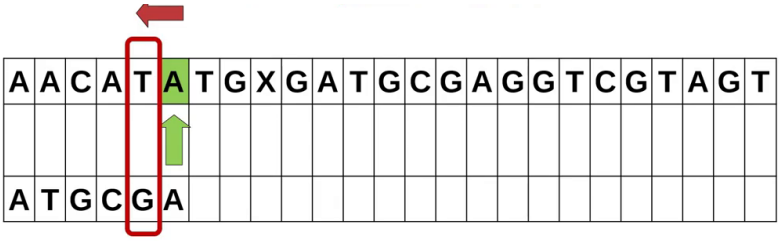
Dans la méthode naïve, les décalages du motif vers la droite se faisaient toujours d'un "cran" à la fois. L'intérêt de l'algorithme de Boyer-Moore, c'est qu'il permet d'effectuer un **décalage de plusieurs crans** en une seule fois lorsque c'est possible.
Pour ce, deux choses sont utiles :

- effectuer un **prétraitement du motif**. Cela signifie que l'algorithme "connait" les caractères qui se trouvent dans le motif
- commencer la comparaison motif-chaine par la droite du motif. On parcourt le motif **de la droite vers la gauche**.

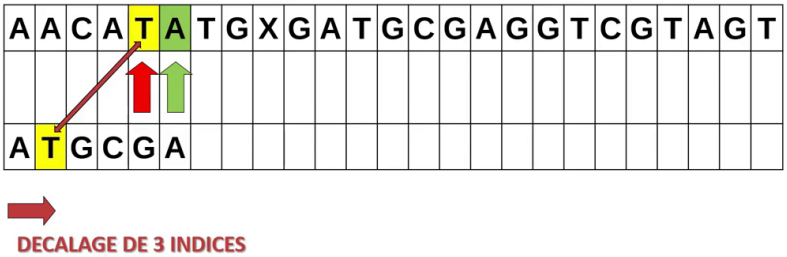
Illustrons le principe :



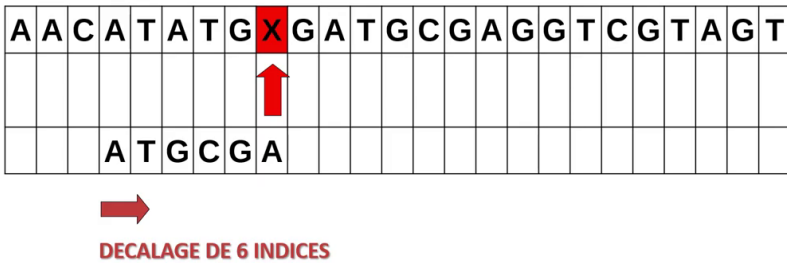
On se place au début. On commence par comparer le dernier caractère. Il match ! cool ! On continu les comparaisons de droite à gauche.



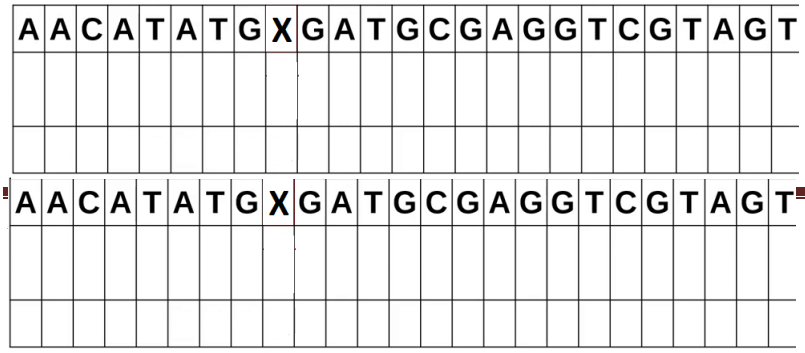
La seconde comparaison mismatch. On va donc décaler mais regardons si ce second caractère appartient au motif.



Oui, il appartient au motif. Je vais décaler d'autant qu'il faut pour que la caractère en commun match avec le motif. On décale donc de 3.



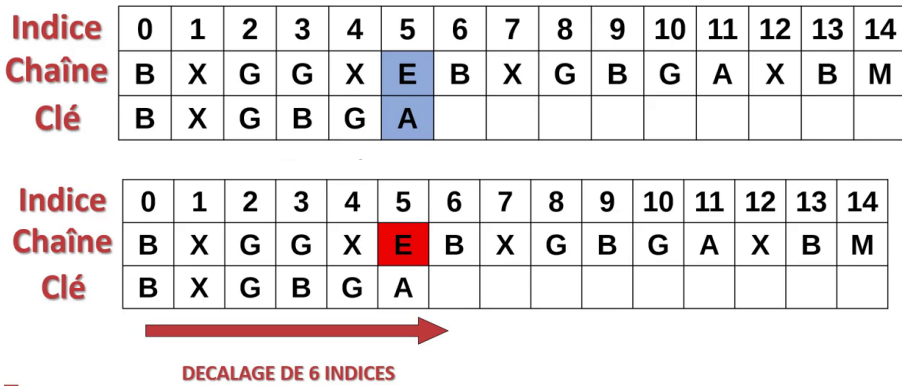
Le premier caractère comparé mismatch. On regarde s'il y a un X dans le motif. Non, il n'y en a pas, on peut décaler de la longueur du motif, soit de 6 !



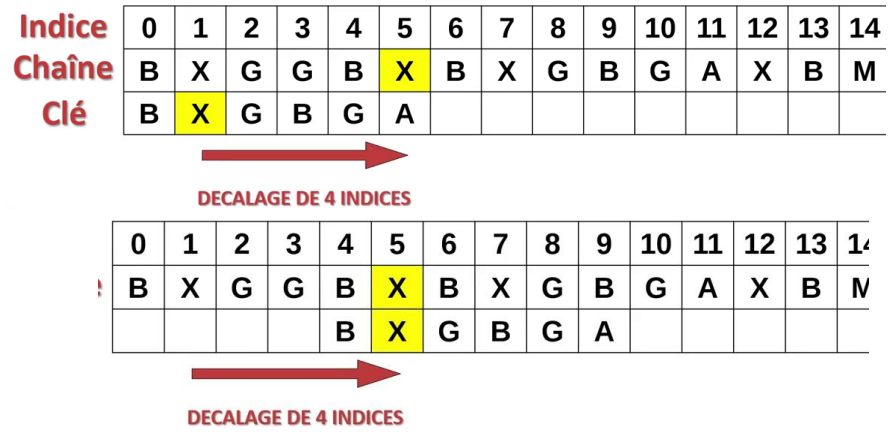
Terminez le travail :

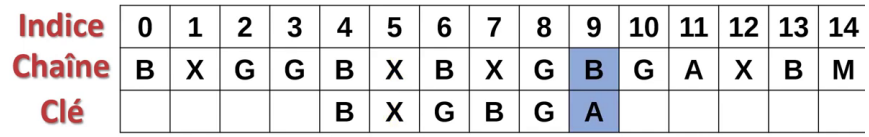
Quelques exemples pour se familiariser avec le principe :

Cas d'une lettre non présente dans la clé :

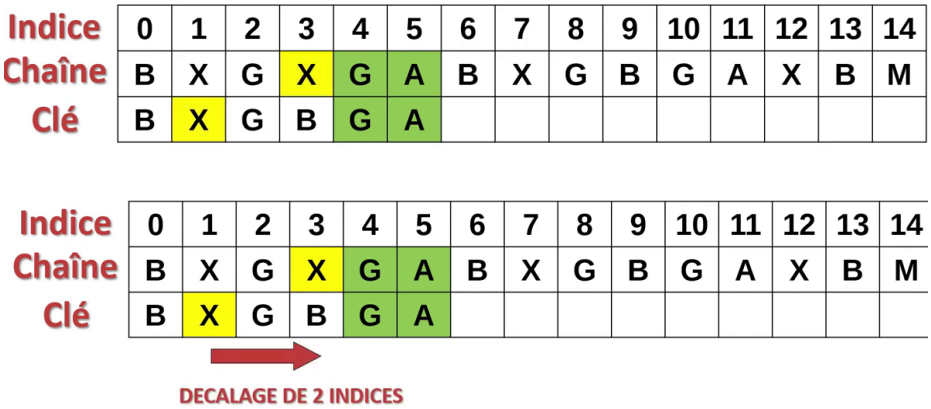


Cas d'une lettre présente dans la clé :





Cas d'une lettre présente dans la clé après quelques coïncidences :



Rq : plus le motif est long, plus cet algorithme sera efficace : les sauts pourront être d'autant plus grand.

Le prétraitement (création de la table de saut):

On a vu qu'on ne se contentait pas de comparer le caractère de la chaine avec juste le caractère en face mais avec l'ensemble des caractères du motif. Le programme doit donc **avoir en mémoire les différents caractères du motif**. Il doit **avoir aussi « en tête » le nombre de décalage à faire** selon le caractère rencontré.

Avant de lancer l'algorithme, il faut donc créer une **table de saut** pour chaque caractère de la clef (sauf la dernière).

Une idée est d'utiliser un dictionnaire des lettres présentes avec la position de la lettre par rapport à la dernière (ce qui correspond au saut à effectuer si la lettre est trouvée)

Réalisation de la table des sauts :

VAR motif en chaine de caractères
VAR table_des_sauts en dictionnaire

POUR i allant de 1 à lenght(motif)-1 // pas besoin de mettre la dernière lettre dans la table
    table_des_sauts[motif[i]] = lenght(motif) – i]
FIN POUR

RETOURNER table_des_sauts

Implantation en Python (aide : souvenez-vous : en Python, une chaine de caractère peut se lire comme une liste (de caractère). Ex : si *texte= « bonjour »* alors *texte[0]* vaut *« b », texte[1]* vaut *« o »*) :

Indiquer le résultat obtenu avec votre fonction pour la clé « NSI » (faites le d'abord « à la main »).

Indiquer le résultat obtenu avec votre fonction pour la clé « EXCELLENT » (faites le d'abord « à la main »).

Voila une implantation en Python d'un algorithme de Boyer Moore, qui utilise la fonction que nous avons défini page précédente :

```
def boyer_moore (texte, motif):
    long_txt = len(texte)
    long_motif = len(motif)
    positions = []

    if long_motif <= long_txt :
        decalage = table_sauts(motif)
        i=0
        trouve = False
        while (i <= long_txt-long_motif):
            for j in range (long_motif -1, -1, -1):
                trouve = True
                if texte[i+j] != motif[j] :
                    if (texte[i+j] in decalage and decalage[texte[i+j]]<=j):
                            i=i+decalage[texte[i+j]]
                    else :
                        i=i+j+1
                    trouve = False
                    break

            if trouve :
                positions.append(i)
                i=i+1
                trouve = False
    return positions
```

➔ Ouvrez ce script dans votre éditeur préféré (vous avez le fichier ***boyer moore eleve.py*** de disponible). Faites le fonctionner puis ajoutez l'ensemble des commentaires expliquant les différentes étapes de l'algorithme. *(rappel : n'hésitez pas à chercher des infos sur le web si le sens de quelques instructions vous échappent. Par exemple ici : https://docs.python.org)*

➔ Utilisons notre programme pour effectuer une recherche dans un livre entier. Vous avez à dispo le fichier *LeRougeEtLeNoir.txt* , qui contient l'ensemble du roman de Stendhal.
Ecrivez un programme qui permette de répondre aux questions suivantes :

- Combien de lettres constituent ce livre.
- Combien de fois le mot « Sorel » est-il présent ?
- Combien de mots constituent ce roman (les mots sont souvent séparés par un espace…) ?
- Combien de phrases constituent ce roman (les phrases se terminent souvent par un point) ?

On a réinventé la roue…

Comme vous vous en doutez peut être, les algorithmes « de bases » sont déjà disponibles dans la plupart des distributions des langages de programmation.

➔ Essayez la méthode *find* disponible pour les chaines de caractères en Python : *stendhal.find("Sorel")* . Cette méthode vous renvoie l'index de la première occurrence trouvée.

Crédits : Documents EDUSCOL, David Roche, Cédric Gerland