Recherche textuelle : algorithme de Boyer-Moore

1. Recherche textuelle simple (naïve)

La recherche naïve d'une sous-chaine substrg dans une chaine strg consiste à parcourir la chaine strg caractère par caractère en recherchant le premier caractère de substrg. Si on le trouve, on teste les caractères suivants un par un…

Exercices

Appliquer l'algorithme naïf pour rechercher "papas" dans
"un papou papa à poux a des poux papas et des poux pas papas".
Compter le nombre de comparaisons de caractères dans cette recherche.

Même question en recherchant le motif 001 dans 00000001.

Même question en recherchant le motif "avis" dans "Réfléchir est un bon moyen de progresser".

Complexité

On cherche la sous-chaine substrg dans strg (le texte). On note n = len(strg), k = len(substrg).

La complexité est :

- en O($n.k$) dans le pire des cas
- en O(n) dans le meilleur des cas (si les premières lettres de substrg n'apparaissent pas dans strg en dehors d'une occurrence de substrg).

Il est difficile de parler de complexité en moyenne sans connaitre la nature des données, mais on peut penser que dans la plupart des cas, 1 à 3 lettres suffisent par vérification et la complexité est alors en O(n).

Deux techniques permettent d'améliorer l'efficacité d'un algorithme de recherche textuelle :

- Prétraiter le motif substrg : algorithme de Boyer-Moore
- Prétraiter le texte strg : construction d'index ou d'arbre de préfixes

2. Algorithme de Boyer-Moore en version simplifiée de Horspool

Principe

L'algorithme de Boyer-Moore effectue la vérification à l'envers : s'il cherche "TARTEMPION" dans un texte, il teste d'abord le 10^{e} caractère de ce texte. S'il trouve un N, il regarde si le 9^{e} caractère est un O, puis le 8^{e} un I, … jusqu'au 1er.
L'intérêt de cet algorithme est que si on trouve en 10^{e} position une lettre n'apparaissant pas dans le motif TARTEMPION, il sait que ce motif commence au mieux en 11^{e} position et saute directement à la 20^{e} position pour rechercher un N.
Ainsi, il vérifie un seul caractère au lieu de dix.

Que faire s'il trouve par exemple un E au lieu d'un N en dixième position ?
Il se peut que ce soit le E de TARTEMPION, auquel cas le N serait en 15e position. L'algorithme saute alors à la 15e position pour rechercher le N (puis éventuellement le O, le I…).
Le saut dépend donc du caractère lu : dans le cas de TARTEMPION, +10 si le caractère n'est pas dans le motif, +5 pour un E, +2 pour un I, +6 pour un T.

Première table de saut (bad character rule)
Plutôt que de calculer la longueur du saut à chaque fois, on construit une table de sauts avant de lancer la recherche.

Exemple 1
Première table de sauts de tartempion

a	r	t	e	m	p	i	o	autre
+8	+7	+6	+5	+4	+3	+2	+1	+10

À chaque lecture d'une lettre, si la lettre correspond à la lettre recherchée, on compare les lettres précédentes pour vérifier s'il s'agit du motif cherché. Sinon, on utilise la table de sauts pour décaler la fenêtre de recherche.

Rechercher "tartempion" dans "son nom e**s**t artemis**,** ne l'app**e**lle **p**as tartempion."

Exemple 2
Première table de sauts de bbedb

d	e	b	autre
+1	+2	+3	+5

Rechercher "bbedb" dans "acebbedba"

Cas de la dernière lettre : si on lit un **b** en cherchant la dernière lettre du motif, c'est peut-être le dernier **b** de bbedb, et on va vérifier les lettres précédentes, mais si on trouve un autre **b** lors de cette vérification, cela ne peut pas être le dernier **b** de bbedb puisqu'on a déjà testé les caractères à gauche des 5 dernières positions. On peut donc effectuer un saut de 3 positions.

Exemple 3
Première table de sauts de tata

t	a	Autre
+1	+2	+4

Rechercher "tata" dans "ma **tat**ie e**s**t u**n**e b**atta**nte".
Vérifier que l'algorithme décrit ci-dessus boucle.

Lors de la vérification, on repart en arrière et le saut effectué peut nous ramener sur une lettre déjà vérifiée. L'algorithme boucle. Pour éviter cette situation, on avance à la lettre suivant la dernière lettre testée lorsque le saut n'est pas suffisamment long.

Une description de l'algorithme de Boyer-Moore simplifié (version à une table de sauts)
On construit la table de saut comme ci-dessus.
La fenêtre de recherche se déplace dans le texte strg de gauche à droite.
La comparaison avec le motif au sein de la fenêtre de recherche s'effectue de droite à gauche.
On note :

- i l'indice du premier caractère de la fenêtre de recherche dans la chaine strg,
- j l'indice du caractère dans le motif substrg lors de la comparaison
- k la longueur du motif
- m le nombre m = k – 1 – j (voir tableau ci-dessous)

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
strg	t	e	x	t	e	d	a	n	s	L	e	q	u	e	l	o	n	e	f	f	e	c	t	u	e	l	a	r	e	c	h	e	r	c	h	e
substrg															m	o	t	i	f																		
j															0	1	2	3	4																		
m=k-1-j															4	3	2	1	0																		

La fenêtre de recherche est la zone grisée

En cas de correspondance, on renvoie la position i du premier caractère de la fenêtre de recherche.
Dans le cas général (pas de correspondance), on déplace la fenêtre de recherche de (saut – (k – 1 – j)), soit (saut – m) positions, la valeur de saut étant donnée par la table de recherche, sauf si cette différence est négative ou nul (ce qui ferait boucler l'algorithme), auquel cas on déplace la fenêtre de recherche d'une position.

En pratique, la table de sauts est souvent un tableau dont la longueur est égale au nombre de lettres de l'alphabet, mais dans notre implémentation en Python, on utilisera un dictionnaire contenant uniquement les lettres du motif (sauf la dernière éventuellement).

Exercices
Appliquer l'algorithme de Horspool pour rechercher "papas" dans
"un papou papa à poux a des poux papas et des poux pas papas".
Compter le nombre de comparaisons de caractères dans cette recherche.

Même question en recherchant le motif 001 dans 00000001.

Même question en recherchant le motif "avis" dans "Réfléchir est un bon moyen de progresser".

Un exemple en bioinformatique
https://www.canal-u.tv/video/inria/3_6_l_algorithme_de_boyer_moore.24590
Ce n'est pas tout-à-fait la version que celle décrite dans ce cours.

3. Algorithme de Boyer-Moore pour les plus motivés

Le véritable algorithme de Boyer-Moore utilise en plus une deuxième table de sauts (the good suffix rule). La première table s'appuie sur la lecture d'un seul caractère, la deuxième sur la comparaison des suffixes (dernières lettres du motif). Par exemple, les suffixes de CONDA sont A, DA, NDA, ONDA.

Explication par l'exemple
Examinons différentes situations.

Dans amim**a**popadame
on recherche cond**a**
On compare le **a** de conda avec le **a** situé au-dessus, puis le **d** avec le **m**. Comme le suffixe **a** n'apparait plus dans conda, on décale la fenêtre de recherche de 5 positions (longueur du motif).

Dans oromom**a**popadame
on recherche odacond**a**
On compare le dernier **a** de odacond**a** avec le **a** au-dessus, puis le **d** avec le **m**. Le suffixe **a** réapparaît dans od**a**cond**a**, mais il est encore précédé d'un **d**. Or, la comparaison avec le **d** vient d'échouer, donc on sait qu'à cette position, le texte ne contient pas **da**. Du coup, on décale la fenêtre de recherche de 8 positions (longueur de odaconda).

Dans oromom**a**popadame
on recherche anacond**a**
Même démarche, mais le suffixe **a** sans **d** devant réapparait 5 positions vers la gauche du motif. On décale donc le motif de 5 positions pour aligner les deux **a** :

oromom**a**popadame
an**a**conda

Dans obo**bab**aboba
on recherche oba**bab**
Dans cette situation, on compare les lettres **b**, **a**, **b** puis **a** ≠ **o**. Le suffixe **bab** pas précédé d'un **a** est présent deux positions vers la gauche. On applique donc un décalage de 2 positions :

obo**bab**aboba
o**bab**ab

Dans anausaona**na**raunaunasaunarana
on recherche anaunasau**na**
On compare les lettres **a**, puis **n**, puis **u** ≠ **a**. Si la comparaison s'arrête à la lettre **u** (m = 2), alors on sait que le suffixe **na** est présent précédé d'une autre lettre que **u** dans la fenêtre le recherche. On décale donc la fenêtre de recherche de huit positions car le suffixe **na** précédé de **u** ne convient pas :

anausaona**na**raunaunasaunarana
a**na**unasauna

Dans nausaonanar**auna**onasaunaranasa
on recherche naonas**auna**
Dans cette situation, on compare les lettres **a**, **n**, **u**, **a** puis **s** ≠ **r**. Le suffixe **auna** n'est pas présent dans le reste du motif, mais on doit aligner le **na** du début du motif avec le **na** de la fin de la fenêtre de recherche. On applique donc un décalage de 8 positions (longueur du motif – la longueur de na) :

nausaonanarau**na**onasaunaranasa
naonasauna

Seconde table de sauts (good suffix rule)
On construit une seconde table de sauts indiquant le décalage à appliquer pour chaque suffixe du motif. Chaque suffixe est identifié par la valeur de m, qui indexe cette table.
Si m = 3 lors de l'échec d'une comparaison, c'est qu'on a déjà comparé avec succès les 3 dernières lettres du motif (una), et que la comparaison avec la lettre précédente a renvoyé un échec, donc m = 3 fait référence au suffixe composé des trois dernières lettres du motif.

Ainsi, pour construire la seconde table de sauts, on doit rechercher les différents suffixes dans le motif. Les motifs n'étant généralement pas longs, on peut effectuer une recherche naïve, mais de droite à gauche pour trouver la précédente occurrence du suffixe dans le motif. On peut effectuer cette recherche en appliquant Boyer-Moore au motif de manière récursive, mais comme Boyer-Moore va renvoyer la première occurrence du suffixe et qu'on veut la dernière, on applique d'abord un effet miroir au motif et à son suffixe avant de lancer la recherche.

En pratique, l'algorithme de Boyer-Moore utilise les deux tables de sauts. À chaque étape, il détermine les décalages fournis par les deux tables et prend le meilleur.

Complexité

Dans le cas le plus favorable, la complexité est en O(*n* / *k*) : dans ce meilleur cas, seul un caractère sur substrg doit être vérifié. Ainsi, plus la sous-chaîne est longue, et plus l'algorithme est efficace pour la trouver.

Pour étudier la complexité en moyenne, il faudrait connaitre la nature des données, mais on peut penser que le plus souvent, il n'y a pas ou peu de longues chaines de caractères ressemblant à la clé cherchée et on est très proche du meilleur des cas. La complexité est donc généralement en O(*n* / *k*).

Applications à des alphabets très petits ou très grands

- Cas d'un alphabet très petit (binaire par exemple) : on travaille sur des groupes de caractères (pour simuler un alphabet plus grand)
- Cas d'un alphabet très grand (Unicode par exemple) : on réduit l'alphabet à des classes de caractères (inverse du cas précédent)