Chapitre 3

Recherche textuelle

La recherche d'une sous-chaîne au sein d'une chaîne de caractères est un problème intervenant dans de nombreux logiciels comme :

— les navigateurs, les lecteurs de documents, les applications de messagerie avec les fonctions « recherche » ;

— les éditeurs de texte ou de code qui comportent eux aussi des fonctions « rechercher », souvent couplées à des fonctions « remplacer » ;

— les langages de programmation avec en Python le mot clé `in` par exemple.

On peut également citer des applications en recherche scientifique comme la recherche de séquences de nucléotides ou de gênes au sein du génome d'un individu. Ce dernier cas pointe à lui seul la nécessité de développer des algorithmes performants puisqu'un gêne peut contenir plusieurs milliers, voire millions de nucléotides parmi les milliards qui composent un génome.

Il existe de nombreux algorithmes de recherche d'une sous-chaîne au sein d'une chaîne de caractères, plus ou moins efficients en fonctions des chaînes et sous-chaînes, mais aussi de l'encodage utilisé pour le texte.

Dans ce chapitre, nous allons voir une approche « naïve » ou par force brute qui constitue le pire des cas en terme de complexité, puis l'algorithme de Boyer-Moore qui constitue une approche extrêmement efficace, notamment lorsque la sous-chaîne recherchée est longue.

3.1 Recherche par force brute

L'approche par force brute consiste à faire glisser la sous-chaîne recherchée le long dans la chaîne en commençant par le début puis en effectuant des comparaisons pour chaque lettre composant la sous-chaîne jusqu'à rencontrer une différence ou à constater la présence de celle-ci dans la chaîne. Dans les deux cas, on avance ensuite d'un caractère et on réitère pour détecter la sous-chaîne une première fois si une différence a été constatée à l'étape précédente ou une nouvelle fois si la sous-chaîne a été trouvée.

Exemple : recherchons la chaîne de caractères « feu » dans « dracaufeu » selon une approche par force brute. On commence par comparer « feu » avec les trois premiers caractères de « dracaufeu » comme dans le tableau ci-dessous

d	r	a	c	a	u	f	e	u
f	e	u						

La comparaison des deux chaînes grisées échoue dès le premier caractère. On passe donc à la comparaison à partir du deuxième caractère comme dans le tableau suivant.

d	r	a	c	a	u	f	e	u
	f	e	u					

Cette comparaison échoue à nouveau dès le premier caractère. En réitèrant le processus jusqu'à la dernière comparaison possible, on arrive à la situation ci-dessous, laquelle atteste de la présence de « feu » dans « dracaufeu ».

d	r	a	c	a	u	f	e	u
						f	e	u

3.2 Algorithme de Boyer-Moore

3.2.1 Principe

L'algorithme de Boyer-Moore est une amélioration extrêmement efficace de l'approche par force brute reposant sur deux idées principales :

— une comparaison des deux sous-chaînes de caractères non plus par le début mais par la fin de celles-ci ;

— une avancée non plus caractère par caractère mais par sauts en fonction de la présence ou de l'absence des caractères de la chaîne dans la sous-chaîne recherchée.

3.2.2 Exemples

Exemple 1 : reprenons la recherche de « feu » dans « dracaufeu ». On commence non plus par le début mais par la fin des deux sous-chaînes en comparaison, donc en position 2 (caractères grisés).

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
f	e	u						
0	1	2						

Les deux caractères ne correspondent pas et « a » n'apparaît pas dans « feu », il ne sert donc à rien de tester les deux positions suivantes, i.e. avec « f » en positions 1 et 2 :

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
	f	e	u					
	0	1	2					

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
		f	e	u				
		0	1	2				

Avec l'algorithme de Boyer-Moore, on effectue dans ce cas un saut équivalent à la longueur de la sous-chaîne recherchée, ici 3. On est donc dans la position ci-dessous.

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
			f	e	u			
			0	1	2			

Cette fois-ci, la première comparaison, celle des « u » en position 5 s'avère fructueuse. On effectue donc la comparaison suivante en position 4 avec « e » et « a » qui, elle, s'avère infructueuse. Comme « a » n'appartient pas au reste de la sous-chaîne recherchée, il ne sert à nouveau à rien de tester les deux positions suivantes, i.e. avec « f » en positions 4 et 5. On peut donc effectuer un saut équivalent à la longueur de la sous-chaîne recherchée moins le nombre caractères ayant eu une comparaison positive – ici, $3 - 1 = 2$ – et directement passer à la position 5.

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
					f	e	u	
					0	1	2	

Ici, la comparaison échoue ; on avance d'un indice, ce qui nous amène à la dernière position.

0	1	2	3	4	5	6	7	8
d	r	a	c	a	u	f	e	u
						f	e	u
						0	1	2

Dans ce dernier cas, on commence par comparer les « u » ; cette comparaison s'avérant positive, on compare les « e » et enfin les « f ». La sous-chaîne « feu » a été trouvée à l'aide sept comparaisons avec l'algorithme de Boyer-Moore.

Exemple 2 : recherchons « psykokwak » dans la phrase « akwakwak et psykokwak ».

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	k	w	a	k	w	a	k		e	t		p	s	y	k	o	k	w	a	k
p	s	y	k	o	k	w	a	k												
0	1	2	3	4	5	6	7	8												

La première comparaison échoue et comme l'espace « » n'appartient à « psykokwak », on peut effectuer un saut de longueur à la taille de « psykokwak », i.e. 9.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	k	w	a	k	w	a	k		e	t		p	s	y	k	o	k	w	a	k
									p	s	y	k	o	k	w	a	k			
									0	1	2	3	4	5	6	7	8			

La première comparaison – celle des « k » – est positive, mais pas la seconde – celle de « a » et « o ». Toutefois, on remarque que « psykokwak » contient un « o » dont la dernière occurrence est en position 4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	k	w	a	k	w	a	k		e	t		p	s	y	k	o	k	w	a	k
									p	s	y	k	o	k	w	a	k			
									0	1	2	3	4	5	6	7	8			

On peut donc effectuer un saut longueur $7 - 4 = 3$ afin de les mettre en correspondance.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a	k	w	a	k	w	a	k		e	t		p	s	y	k	o	k	w	a	k
												p	s	y	k	o	k	w	a	k
												0	1	2	3	4	5	6	7	8

L'algorithme de Boyer-Moore a trouvé « psykokwak » en douze comparaisons contre vingt et une pour la recherche par force brute.

Exemple 3 : recherchons « akwakwak » dans la phrase « psykokwak et akwakwak ».

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p	s	y	k	o	k	w	a	k		e	t		a	k	w	a	k	w	a	k
a	k	w	a	k	w	a	k													
0	1	2	3	4	5	6	7													

La première comparaison échoue, toutefois, on observe que « a » appartient à la sous-chaîne recherchée et se trouve à une distance de 1, on effectue donc un saut de 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p	s	y	k	o	k	w	a	k		e	t		a	k	w	a	k	w	a	k
	a	k	w	a	k	w	a	k												
	0	1	2	3	4	5	6	7												

On a à présent quatre comparaisons positives et une négative. Le « o » n'appartient pas au reste de la clé, on peut donc effectuer un saut de façon à ce que la clé se retrouve immédiatement après le « o ». C'est un saut de longueur égale au nombre de caractères de la clé qui n'ont pas été comparés plus le dernier comparé ; ou encore la longueur de la chaîne moins le nombre de caractères comparés avec succès. On fait donc un saut de 4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p	s	y	k	o	k	w	a	k		e	t		a	k	w	a	k	w	a	k
					a	k	w	a	k	w	a	k								
					0	1	2	3	4	5	6	7								

Comme il n'y a pas d'espace dans « akwakwak », on peut effectuer un saut sur toute la longueur de la sous-chaîne, ce qui nous permet de déterminer sa présence dans la chaîne.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p	s	y	k	o	k	w	a	k		e	t		a	k	w	a	k	w	a	k
													a	k	w	a	k	w	a	k
													0	1	2	3	4	5	6	7

3.2.3 Prétraitement

L'algorithme de Boyer-Moore nécessite une phase prétraitement de la sous-chaîne recherchée – aussi appelée clé – afin de prendre connaissance des caractères la et composant d'établir une table de sauts permettant d'obtenir les meilleurs sauts possibles lors de son exécution. Ce sont ces sauts qui permettent de réduire le nombre de comparaisons nécessaires à l'algorithme et donc sa complexité temporelle.

La table est composée de la distance entre la dernière occurrence de chacun des caractères de la clé et le dernier de celle-ci, à l'exception du dernier lui-même.

Exemple 1 : la table des sauts pour « feu » donne

caractère	f	e
distance	2	1

Chacune des distances du tableau est calculée par rapport au dernier caractère « u » de « feu » ; puisque l'on fait exception de ce dernier, il n'apparaît pas dans la table.

Exemple 2 : la table des sauts pour « psykokwak » donne

caractère	p	s	y	k	o	w	a
distance	8	7	6	3	4	2	1

Chacune des distances du tableau est calculée par rapport au dernier « k » de « psykokwak ». On remarquera que la distance donnée pour « k » lui-même n'est pas égale à 0 car il s'agit de la dernière occurrence des « k » à l'exception de la dernière lettre, i.e. du troisième « k » ; la dernière occurrence des « k » considérée est donc la deuxième.

Remarque : il est possible de construire une table pour tous les caractères de la table utilisée (ASCII, Unicode, etc). Cependant, l'immense majorité des éléments de ces tables ne seront pas présents dans la clé ; on pourrait les traiter en mettant leur valeur de dernière occurrence égale à la longueur de la clé. Toutefois, il est plus simple de ne construire que pour les caractères de la clé et de vérifier si le caractère comparé y appartient bien.

3.2.4 Algorithme

Dans l'algorithme ci-dessous, on suppose que l'on dispose de la fonction ***construction_table_sauts*** prenant en entrée une clé et renvoyant sa table des sauts sous la forme d'un dictionnaire comme décrit dans la section précédente.

Algorithme 1 : Boyer-Moore

```
1  Fonction boyer-moore(texte : str, clé : str) → liste :
2
3      longueur_texte ← longueur(texte)
4      longueur_clé ← longueur(clé)
5      liste_occurences ← [ ]
6
7      Si longueur_clé ≤ longueur_texte :
8          table_sauts ← construction_table_sauts(clé)
9          i ← longueur_clé −1 # position initiale du dernier caractère de la clé
10
11         Tant que i < longueur_texte :
12             # Comparaison du dernier caractère de la clé
13             j ← 0
14             caractère ← texte[i − j]
15             correspondance ← (caractère == clé[longueur_clé −1 − j])
16
17             # Tant que les caractères correspondent
18             Tant que correspondance et j < longueur_cle −1 :
19                 # on se décale vers la gauche et on vérifie la correspondance suivante
20                 j ← j + 1
21                 caractère ← texte[i − j]
22                 correspondance ← (caractère == clé[longueur_clé −1 − j])
23
24             Si correspondance :
25                 # Cas d'une correspondance sur la totalité de la clé
26                 ajouter i−longueur_clé+1 à liste_occurences
27                 i ← i+longueur_clé
28             Sinon Si caractère ∉ clé :
29                 # Le dernier caractère comparé n'est pas dans la clé
30                 # Saut d'une longueur de clé moins le nombre de caractères
                     précédemment correspondants
31                 i ← i+ longueur_clé −j
32             Sinon :
33                 # Le dernier caractère comparé est présent dans la clé
34                 # Saut donné par la table moins le nombre de caractères précédemment
                     correspondants
35                 i ← i+ table_sauts[caractère]−j
36
       Renvoyer : liste_occurences
```

3.2.5 Complexité temporelle

Notons L la longueur de la chaîne dans laquelle est effectuée la recherche et K la longueur de la sous-chaîne recherchée. L'algorithme de Boyer-Moore a une complexité temporelle de :

— $O(L/K)$ dans le meilleur de cas ;

— $O(L+K)$ dans le pire des cas ;

— à peine supérieure au meilleur des cas la plupart du temps avec un temps moyen en $3K$.

On remarque que l'on est bien en dessous de l'approche par force brute qui a une complexité temporelle de :

— $O(L)$ dans meilleur des cas (cas où l'on ne recherche qu'une seule lettre) ;

— $O(L \times K)$ dans le pire des cas.

Au pire, l'algorithme de Boyer-Moore fournit des résultats comparables aux meilleurs cas de l'approche par force brute (et cela est à relativiser par le fait que ce ne sont probablement pas les mêmes cas).

3.3 Ressources supplémentaires

— Vidéo sur la recherche textuelle.

— Page Wikipédia sur la recherche de sous-chaîne.

— Page Wikipédia sur l'algorithme de Boyer-Moore.

— Visualisation de l'algorithme de Boyer-Moore

3.4 Exercices

Exercice 3.1. [Approche par force brute]

1. Estimer les ordres de grandeurs des nombres de comparaisons à effectuer dans une approche par force brute dans le meilleur et le pire des cas possibles. On pourra compter le nombre de comparaisons à effectuer pour rechercher dans « aaaaaaaa » :
   — « b » ;
   — « aaab ».
2. Programmer une fonction de recherche textuelle par force brute. Elle renverra un booléen pour signifier la présence ou non de la sous-chaîne recherchée dans la chaîne principale et si elle y est présente la liste de ses positions.
3. Si ce n'était pas déjà le cas, faire en sorte que votre fonction ne compare pas plus d'un caractère à la fois.
4. Ajouter un compteur à votre fonction permettant de compter le nombre de comparaisons effectuées par la fonction.

Exercice 3.2. Exécuter l'algorithme de Boyer-Moore sur la chaîne de caractères « Ici Luffy, celui qui deviendra le roi des pirates ! » pour chacune des clés suivantes.

1. « pirate » ;
2. « patate ».

Exercice 3.3. [Le plus pire du plus pire] Trouver un exemple de réalisation du pire des cas pour l'algorithme de Boyer-Moore.

Exercice 3.4. [Table des sauts] Programmer une fonction prenant en entrée une chaîne de caractères et renvoyant en sortie la table des sauts des caractères qui la composent.

Exercice 3.5. [Algorithme de Boyer-Moore]

1. Programmer l'algorithme de Boyer-Moore.
2. Ajouter un compteur à votre fonction permettant de compter le nombre de comparaisons effectuées par la fonction.
3. Vérifier étape par étape avec la visualisation de l'algorithme de Boyer-Moore.
4. Comparer avec l'approche par force brute.