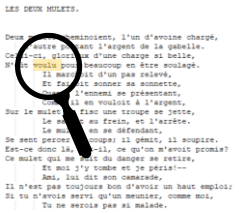
Cours - Recherche textuelle


LES DEUX MULETS.

Deux [illegible] cheminoient, l'un d'avoine chargé,
[illegible]estre portant l'argent de la gabelle.
Celui-ci, glorieux d'une charge si belle,
Fait [illegible] beaucoup en être soulagé.
Il marche, il [illegible] d'un pas relevé,
Et fait sonner sa sonnette,
[illegible] l'ennemi se présentant,
[illegible] en vouloit à l'argent,
Sur le mulet du fisc une troupe se jette,
Le saisit au frein, et l'arrête.
Le mulet en se défendant,
Se sent percer de coups; il gémit, il soupire.
Est-ce donc là, dit-il, ce qu'on m'avoit promis?
Ce mulet qui me sert du danger se retire,
Et moi j'y tombe et je péris!--
Ami, lui dit son camarade,
Il n'est pas toujours bon d'avoir un haut emploi;
Si tu n'avois servi qu'un meunier, comme moi,
Tu ne serois pas si malade.

Objectifs :

- ⇨ Essayer différentes approches de la recherche textuelle
- ⇨ Découvrir et mettre en œuvre l'algorithme de Boyer-Moore

Dans tout ce chapitre on considère la recherche d'un motif (chaîne de caractère) dans un texte (chaîne de caractère de longueur supérieure ou égale à celle du motif).

I - Travail préliminaire

Application n°1 :

1) Ecrire une fonction `def comparaison_algos(algos,args,nb_fois=10)` qui affiche les temps d'exécution des algorithmes dont on fournit un tableau contenant des `str` avec le nom des fonctions. Chaque algorithme est exécuté `nb_fois` avec les arguments `args`. On utilisera la fonction `timeit` du module `timeit`.

Exemple :

```
t = [1,2,3,4,5,6,7,8,9,10,11,12]
algos = ["t.pop", "t.append"]
comparaison_algos(algos, "0")
```

affichera :

```
t.pop : 2.600019797682762e-06 pour 10 répétitions
t.append : 1.400010660290718e-06 pour 10 répétitions
```

Cette fonction de comparaison nous permettra par la suite de comparer le temps d'exécution des différentes variantes des algorithmes de recherche programmés.

II - Recherche simple

On commence par une recherche simple, consistant à essayer de faire correspondre le motif avec chaque sous-chaîne du texte de même longueur.

Application n°2 :

1) Ecrire une fonction `recherche_simple(motif, texte)` qui renvoie un tableau des index de toutes les occurrences de `motif` dans `texte` en utilisant un algorithme simple qui passe en revue toutes les lettres de `texte` successivement et vérifie à chaque position la correspondance avec le motif.

`recherche_simple("ra","abracadabra")` renverra `[2, 9]`.

`recherche_simple("bri","abracadabra")` renverra `[]`.

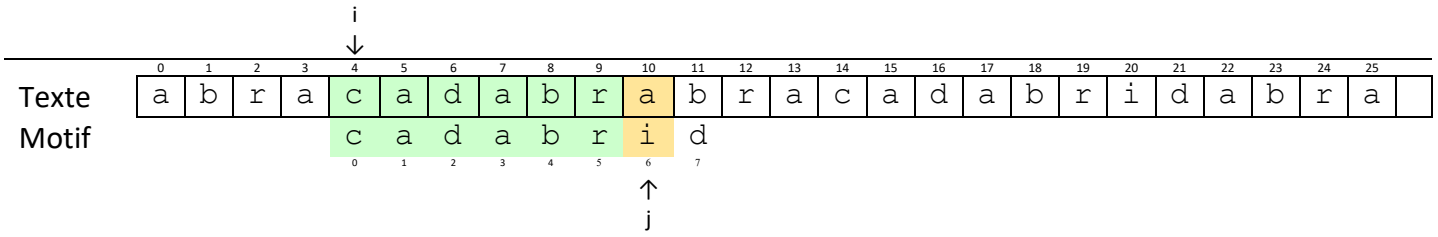
2) Quelle est la complexité de cet algorithme ?

III - Améliorations

1) Règle du bon suffixe

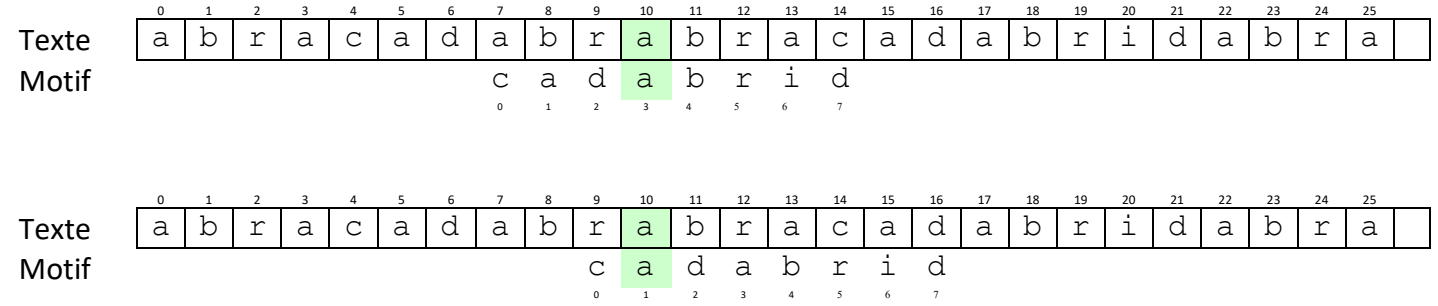
La recherche précédente peut être améliorée en utilisant la connaissance qu'on obtient lorsque des caractères du motif correspondent au texte mais pas tous (correspondance partielle ou bon suffixe).

Par exemple si on considère la recherche de « `cadabrib` » dans « `abracadabrabracadabridabra` », lorsque la recherche arrive au caractère $i = 4$, les caractères correspondent jusqu'au caractère $i + j = 10$ où un « `a` » est trouvé dans le texte alors que le motif contient un « `i` » à cette position.

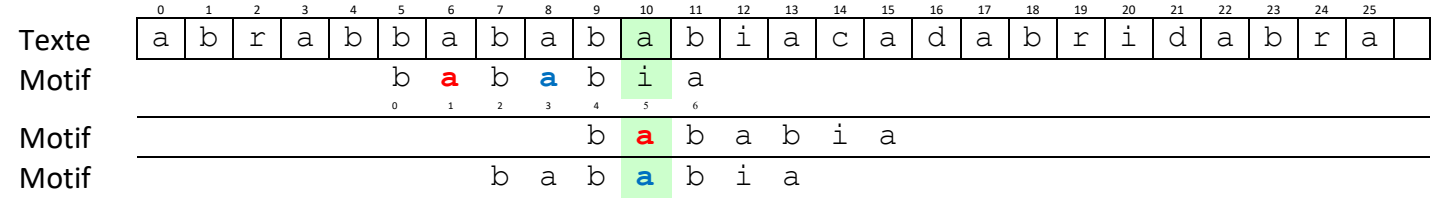


On pourrait repartir de la position $i = 5$, mais étant donné qu'on sait qu'il y a un caractère « `a` » en position 10 et qu'il y en a d'autres dans le motif, on peut décaler le motif (avancer i) de manière à faire correspondre le « `a` » en position 10 du texte (qui a causé l'échec de la comparaison) avec un caractère « `a` » du motif. De cette manière on peut progresser davantage.

On peut faire avancer à $i = 7$ ou à $i = 9$.



Le choix du premier « `a` » (le plus loin du caractère ayant fait échouer, celui qui nous place en position $i = 9$) serait plus intéressant car il fait sauter plus de comparaisons, mais il risque de nous faire passer à côté de correspondances particulières.



On voit ici qu'en sautant au premier « `a` » du motif, on a manqué une correspondance à l'indice $i = 9$.

Enfin, si le caractère manqué dans le texte (celui à l'indice $i + j$) **n'est pas présent dans le motif**, cela veut dire qu'on peut avancer la recherche jusqu'au caractère suivant celui-ci car le motif ne correspondra jamais à cet endroit.

Finalement, l'amélioration proposée revient à comparer à partir de l'indice i le texte avec le motif en utilisant un index j sur le motif (on compare à chaque fois le caractère $i+j$ du texte avec le caractère j du motif) :

- Si la sous-chaîne du texte correspond intégralement au motif, on a trouvé une occurrence à l'indice i. On la reporte et on continu la recherche à l'indice $i + 1$.
- Si on a une différence au caractère $i + j$ du texte (avec le caractère j du motif), note k la position de la différence ($k = i + j$) on incrémente i jusqu'à ce que :
  - i dépasse k ;

  ou
  - le caractère à l'indice $k - i$ du motif soit égal au caractère à l'indice k du texte.

Application n°3 :

1) Ecrire une fonction `recherche_amelioree(motif, texte)` qui renvoie un tableau des index de toutes les occurrences de `motif` dans `texte` en implémentant l'amélioration vue plus haut.
`recherche_simple("cadabrid","abracadabrabracadabridabra")` renverra [14].
`recherche_simple("bababia","abrabbabababiacadabridabra")` renverra [7].
2) Cet algorithme est-il plus rapide ?

2) Précalculer pour accélérer

Le problème de rapidité vu précédemment vient du fait qu'on doit finalement passer en revue les caractères du motif à chaque fois pour savoir combien de caractères on peut sauter, ce qui revient finalement à faire autant de comparaisons.
On peut remarquer que ces comparaisons se font à chaque fois sur le motif et que celui-ci est généralement de petite taille par rapport au texte. Il peut donc être judicieux de calculer les valeurs de sauts (de combien de caractères peut-on avancer) dans chaque cas à l'avance.
On effectue un pré-traitement en calculant les valeurs de sauts en fonction du nombre de lettres qui ont correspondu dans le motif et de la lettre manquée. On va stocker ces calculs dans une structure de donnée adaptée puis lancer la recherche dans le texte en utilisant cette structure plutôt que de refaire les calculs à chaque fois.

Application n°4 :

1) Quelle structure de donnée peut-on utiliser ici pour stocker les valeurs de sauts ?

On choisit d'utiliser un dictionnaire dont les clés sont les différents caractères du motif et les valeurs sont des listes, indexées sur le nombre de caractères du motif ayant été testés avant un échec de la comparaison (variable *j*) et indiquant le nombre de caractères à sauter pour la recherche (à ajouter à l'index *i*).

Ainsi pour le motif « `textuelle` », la table serait celle-ci-contre :

									?								
0									t	e	x	t	u	e	l	l	e
1								t	e	x	t	u	e	l	l	e	
2							t	e	x	t	u	e	l	l	e		
3						t	e	x	t	u	e	l	l	e			
4					t	e	x	t	u	e	l	l	e				
5				t	e	x	t	u	e	l	l	e					
6			t	e	x	t	u	e	l	l	e						
7		t	e	x	t	u	e	l	l	e							
8	t	e	x	t	u	e	l	l	e								

Lettre	Nombre de caractère corrects								
	0	1	2	3	4	5	6	7	8
t	-	1	2	-	1	2	3	4	5
e	1	-	1	2	3	-	1	2	-
x	1	1	-	1	2	3	4	5	6
u	1	1	1	1	-	1	2	3	4
l	1	1	1	1	1	1	-	-	1

2) Calculer « à la main » la table de saut pour le motif « `cadabrid` ».

Vérifier que pour le motif « `cadabrid` » on a bien la table :

```
{'c': [0, 1, 2, 3, 4, 5, 6, 7], 'a': [1, 0, 1, 0, 1, 2, 3, 4], 'd': [1, 1, 0, 1, 2, 3, 4, 0], 'b': [1, 1, 1, 1, 0, 1, 2, 3], 'r': [1, 1, 1, 1, 1, 0, 1, 2], 'i': [1, 1, 1, 1, 1, 1, 0, 1]}
```

3) Ecrire maintenant une fonction `recherche_amelioree_avec_pretraitement(motif, texte, table_sauts)` qui renvoie un tableau des index de toutes les occurrences de `motif` dans `texte` en utilisant la table de sauts `table_sauts` pour accélérer la recherche.
4) Cet algorithme est-il plus rapide ?
5) *(pour les plus rapides)* Ecrire une fonction `creation_table_sauts(motif)` qui prend en argument le motif à rechercher et renvoie une table des sauts. Tester votre fonction avec quelques valeurs de motifs.

Les recherches des exemples proposés ne sont pas réalistes et ne montrent pas l'intérêt de ce type d'optimisation qui sera visible surtout sur des textes ou des motifs suffisamment grands pour que le temps de calcul du pré-traitement soit amorti.

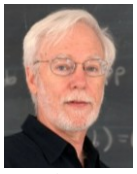
Pour obtenir des comparaisons réalistes, on peut utiliser la fonction ci-dessous et les fichiers proposés.

```
def contenu_fichier(nom_fichier:str, retire_sauts_ligne=True) -> str:
    """Renvoi la chaîne contenue dans le fichier donné en argument, en retirant
    les sauts de ligne (sauf si retire_sauts_ligne est à False)."""
    texte = ""
    with open(nom_fichier, "r", encoding="utf-8") as fichier:
        texte = fichier.read()
    return texte if not retire_sauts_ligne else texte.replace('\n','')
```

5) Reprendre la comparaison des algorithmes, mais en recherchant `"Maître corbeau, sur un arbre perché,"` dans les fables de La Fontaine ou `"CACAAAGCACCCAACTTACACTTAGGAGATTTCAACTTAACT"` dans l'ADN mitochondrial humain. Conclure.

IV - L'algorithme de Boyer-Moore

1) Historique


J Strother Moore


Robert S. Boyer

Cet algorithme de recherche textuelle a été développé par Robert S. Boyer et J Strother Moore, deux informaticiens américains en 1977.

C'est depuis un standard et une référence dans la recherche de texte usuelle. La plupart des logiciels qui utilisent une recherche de texte (comme les navigateurs, les éditeurs de texte, …) le font avec cet algorithme.

2) Fonctionnement de l'algorithme

a. Les principes de base

L'algorithme de Boyer-Moore repose sur 3 principes pour améliorer la rapidité de la recherche en utilisant la connaissance obtenue en comparant un caractère pour en sauter le plus possible :

1. si le caractère ne correspond pas, utiliser sa valeur pour sauter autant d'alignements que possible ;
2. si certains caractères ont correspondu, utiliser la connaissance de la partie correspondante pour sauter les alignements ne pouvant pas correspondre ;
3. faire la recherche des alignements ..... et la comparaison de caractère .....

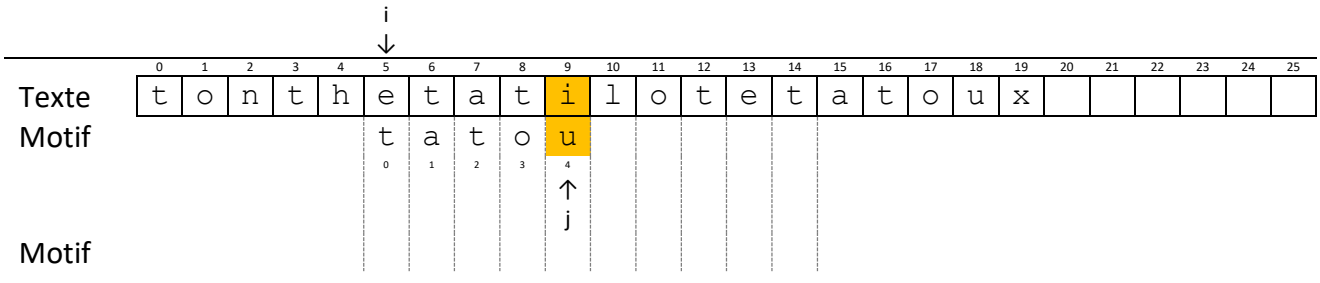
Règle des « ..... »

Règle des « ..... »

Pour des sauts ..... en moyenne

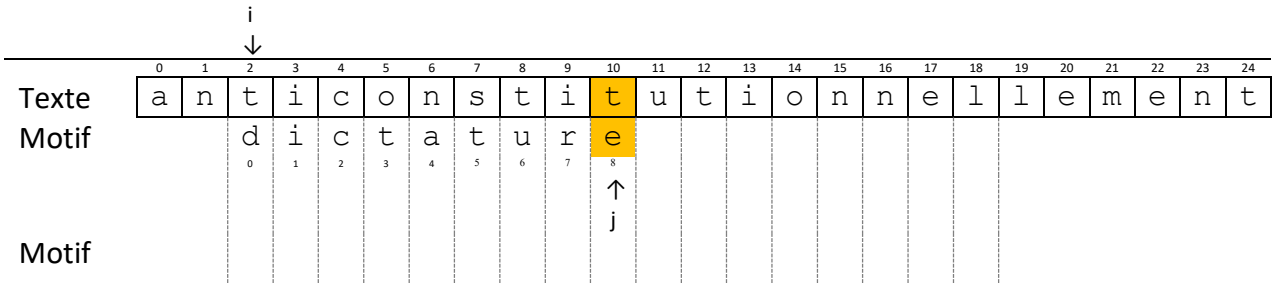
b. La règle des mauvais caractères

Cette règle s'applique lorsque le premier caractère testé n'est pas correct :
Si le caractère incorrect du texte n'apparaît pas dans le motif, alors on saute un nombre de caractère égal à .....



Si le caractère incorrect du texte apparaît dans le motif, alors on saute .....

pour aligner le caractère incorrect .....



Dans la pratique, comme on l'a vu précédemment, on préfèrera pré-calculer une table de saut pour les lettres qui sont dans le motif.

Par exemple pour le motif « `dictature` », la table de saut serait : {`d` : 8, `i` : 7, `c` : 6, `t` : 3, `a` : 4, `u` : 2, `r` : 1}.
On remarque que le caractère « e » n'apparaît pas car il ne peut pas être un mauvais caractère étant donné qu'il se trouve à la fin du motif (là où a lieu la première comparaison).
De même le caractère « `t` » apparaît deux fois dans le motif et on prend donc comme valeur de saut la position du « `t` » le plus à droite du motif (ou plutôt son décalage par rapport à la fin).

c. Règle des bons suffixes

Lorsqu'on a plusieurs caractères qui ont correspondu au motif avant d'arriver sur un caractère différent on peut se servir de la connaissance qu'on a du texte et du motif (qui ont donc leur suffixe (fin du mot) identique) pour sauter un certain nombre de comparaisons.
Si on appelle *s* la sous-chaîne du texte qui correspond avec un suffixe du motif (*s* est donc l'ensemble des caractères qui ont correspondu), alors on avance dans la comparaison (donc on saute des caractères) jusqu'à ce que :

① *s* corresponde à nouveau avec des caractères du motif,

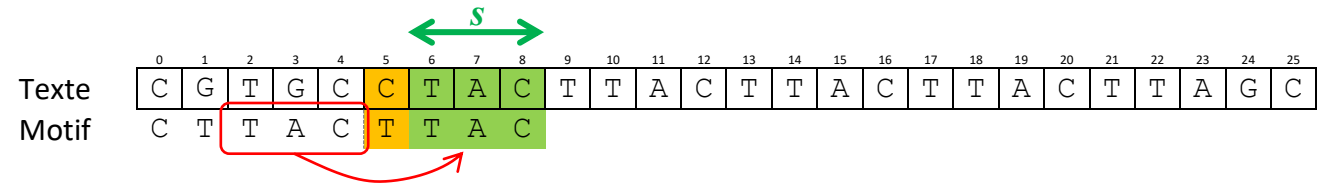
ou

② un préfixe du motif corresponde à un suffixe de *s*,

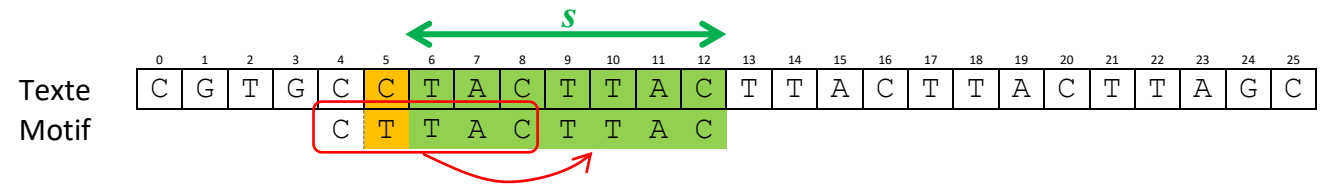
ou

③ le motif dépasse *s*.

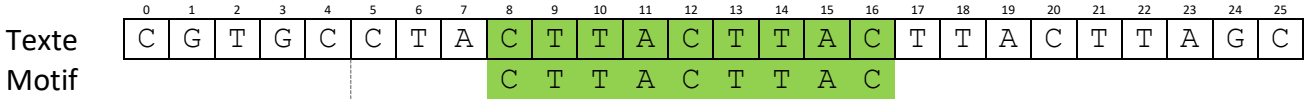
Prenons l'exemple suivant :



Ici on peut avancer de manière à aligner la prochaine occurrence du suffixe « `TAC` » dans le motif avec le suffixe « `TAC` » du texte (cas ① de la règle précédente) :



Maintenant, c'est la partie ② qui s'applique et nous permet de sauter 4 caractères.



De la même manière que pour la règle précédente, on peut pré-calculer une table de saut pour la règle des bons suffixes.

Cette règle peut aussi s'affiner car par exemple dans le cas précédent le premier saut n'était pas utile car on était sûr de ne pas aller au bout de la comparaison car le caractère en position 5 n'est forcément pas un « `T` » et donc avancer à i = 4 mènerai forcément à un échec de comparaison sur le même caractère. On aurait ainsi pu sauter directement 8 caractères pour faire correspondre le « `C` » à fin du motif avec le « `C` » du texte en position 8.

3) Implémentation en python

On cherche maintenant à implémenter l'algorithme de Boyer-Moore en python. Dans un premier temps nous n'appliquerons que la règle des mauvais caractères (C'est la forme appelée « algorithme de Horspool »).

Application n°5 :

1) Déterminer la table de saut des mauvais caractères pour le motif « `cypres` ».
2) Ecrire une fonction `creation_table_sauts_MC(motif)` qui prend en argument une chaîne de caractère représentant le motif à chercher et renvoi la table de sauts correspondante sous la forme d'un dictionnaire python contenant comme clé les lettres du motif et comme valeurs le nombre de comparaisons à sauter.

`creation_table_sauts_MC("cypres")` doit renvoyer `{'s':1,'c':5,'y':4,'p':3,'r':2,'e':1}`.

3) Ecrire une fonction `recherche_boyer_moore_simplifiee(motif, texte)` qui renvoie un tableau des index de toutes les occurrences de `motif` dans `texte` en implémentant algorithme de Boyer-Moore sans la règle des bons suffixes.

Si `texte = "Si six scies scient six cypres, six cent six scies scierons six cent six cypres."`
`recherche_boyer_moore_simplifiee("cypres", texte)` renverra `[24, 73]`.
`recherche_boyer_moore_simplifiee("six", texte)` renverra `[3, 20, 32, 41, 60, 69]`.

4) Comparer la vitesse d'exécution de cet algorithme avec les précédents.
5) Si on note *n* le nombre de caractères dans le texte et *m* le nombre de caractères dans le motif, quelle est la complexité de l'algorithme dans le meilleur et dans le pire des cas ?

Application n°6 :

(Pour les plus rapides)

1) Quelle est la table de saut de « bon suffixe » pour le motif « `CTTACTTAC` » ?
2) Ecrire une fonction `recherche_boyer_moore(motif, texte)` qui renvoie un tableau des index de toutes les occurrences de `motif` dans `texte` en implémentant algorithme de Boyer-Moore complet.

`recherche_boyer_moore("CTTACTTAC", "CGTGCCTACTTACTTACTTACTTACGCGAA")` renverra `[8, 12, 16]`.

3) Comparer sa vitesse d'exécution avec les autres algorithmes.

V - Autres algorithmes de recherche

L'algorithme de Boyer-Moore est le plus utilisé car il est très efficace sur la plupart des cas de recherche simples (où le motif est plutôt court et l'alphabet utilisé pas trop important).
Dans des cas particuliers (par exemple de très grands motifs, d'alphabet réduit ou très grand, s'il faut une tolérance à quelques caractères faux, …) d'autres algorithme peuvent être plus efficaces.
De même certaine optimisation de l'algorithme de Boyer-Moore peuvent être faites pour l'accélérer légèrement (comme le Turbo-BM).

Certains algorithmes, comme celui de Rabin-Karp, utilisent des fonctions de hachage qui permettent une recherche simultanée efficace de plusieurs motifs dans un même texte.

L'essentiel :

.....

.....

.....

.....

.....

.....

.....

.....

.....

Références :

Algorithme Turbo-BM (et autres) : https://www-igm.univ-mlv.fr/~lecroq/string/node15.html#SECTION00150

Algorithme de Rabin-Karp : https://www.baeldung.com/cs/rabin-karp-algorithm

Exercices – **Recherche textuelle**

Exercice 1 : Nombre de comparaisons

1) Combien fait-on de comparaison lorsqu'on recherche le motif « `braca` » dans le texte « `abracadabracadabricadabra` » avec un algorithme naïf (voir début du cours) ?
2) Même question avec l'algorithme de Horspool (Boyer-Moore simplifié)

Exercice 2 : Table de sauts

1) Déterminer manuellement la table de sauts (pour la règle des mauvais caractères – algorithme de Horspool) pour les motifs suivants :
   a) « `informatique` »
   b) « `recherche` »
   c) « `TCGAGGCGAG` »
2) Mêmes questions pour la table de sauts des bons suffixes.

Exercice 3 : Création des tables de sauts

1) Ecrire une fonction `creation_table_sauts_BM(motif)` qui prend en argument une chaîne de caractère représentant le motif à chercher et renvoi la table de sauts selon la règle des bons suffixes correspondante.
2) Tester votre fonction avec les exemples de l'exercice 2.

Exercice 4 : L'algorithme de Raita

Faire une recherche sur l'algorithme de recherche textuelle de Raita et programmer une fonction de recherche `recherche_raita(motif, texte)` implémentant cet algorithme.