



SAE S1.02 – Comparaison d'approches algorithmiques

1. La description officielle au Programme National

Face à un problème algorithmique complexe, l'étudiant doit proposer plusieurs solutions. Chaque solution doit être comparée aux autres sur le plan de l'efficacité en temps d'exécution.

La situation professionnelle est celle du développeur au sein d'une équipe où ce dernier doit résoudre un problème de lenteur d'exécution d'une application.

2. Objectif

L'objectif principal est la mise en œuvre de différents algorithmes pour résoudre un même problème. Par comparaison d'approches algorithmiques distinctes, il est demandé de mettre en évidence par des mesures empiriques la rapidité d'exécution des différents algorithmes.

Il est fortement conseillé de s'inspirer pour cela des techniques de comparaison enseignées dans la ressource R1.01.P2.

3. Le problème de la recherche d'une sous-chaîne dans du texte

Trouver **toutes** les occurrences d'une sous-chaîne de caractères (ou motif) dans un texte est un problème que l'on rencontre fréquemment dans les éditeurs de texte. Le plus souvent, le texte est un document en cours d'édition et la chaîne de caractères recherchée est un mot particulier fourni par l'utilisateur. Des algorithmes efficaces pour ce problème peuvent grandement améliorer la réactivité du programme d'édition. **Toutes les occurrences** signifie qu'il faudra nécessairement parcourir TOUT le texte jusqu'au bout.

Les algorithmes de recherche de chaîne de caractères sont également utilisés, par exemple, pour trouver telle ou telle chaîne de caractères dans une séquence ADN (comparaison d'ADN dans une enquête policière par exemple).

Exemple :

soit le texte : *"L'éléphant phantasme sur les phases insensées de Stéphane"*

soit la sous-chaîne : *"pha"*

On doit pouvoir détecter les 4 apparitions de la sous-chaîne en rouge ci-dessous :

"L'éléphant phantasme sur les phases insensées de Stéphane"

L'objectif de cette SAÉ est d'écrire en Java plusieurs algorithmes qui répondent de manière plus ou moins efficace au problème de recherche de la sous-chaîne dans du texte.

4. Notations/définitions adoptées

Par convention, nous adoptons systématiquement les notations/définitions suivantes :

- le texte dans lequel se fera la recherche est un tableau nommé *text* (en anglais)
exemple : *text = "L'éléphant phantasme sur les phases insensées de Stéphane"*
- la sous-chaîne (motif) recherchée dans le texte est un tableau nommé *pattern* (en anglais)



exemple : *pattern* = "pha"

- on note entre | | la longueur d'une chaîne
exemple : si chaîne = "pha", alors | chaîne | = 3
- on note entre [ind1...ind2] la portion de la chaîne qui est considérée
exemple : si chaîne = "phalalec", alors chaîne[2...6] = "halal"
- *préfixe* d'une chaîne de caractères, définition : portion de la chaîne qui se trouve en tout début de chaîne (exemple : "L'élé" est préfixe de la chaîne "L'éléphant")
- *suffixe* d'une chaîne de caractères, définition : portion de la chaîne qui se trouve en toute fin de chaîne (exemple : "phant" est suffixe de la chaîne "L'éléphant")

5. Efficacité et développement

Il existe beaucoup d'algorithmes parfois assez complexes qui peuvent résoudre le problème de la recherche.

Dans cette SAE, vous mettrez en œuvre 4 algorithmes qui ont chacun des efficacités différentes : l'algorithme naïf, l'algorithme de Knuth-Morris-Pratt, l'algorithme de Rabin-Karp et l'algorithme de Boyer-Moore.

Efficacité

Pour l'évaluation de l'efficacité, on ne vous demande pas toujours de calculer $f(n)$ exacte et d'arriver à une expression de Θ .

Par contre, pour chaque algorithme (1 algo = 1 classe) développé, vous devrez impérativement :

- tester, avant toute chose, votre algorithme sur des exemples faciles à vérifier : méthodes *void testAlgo()* et *void testCasAlgo(...)*,
- coder une méthode intitulée *void testAlgoEfficacite()* qui, pour de (très) grandes tailles de texte et un motif « bien choisi », comptabilise le nombre d'opérations élémentaires exécutées par l'algorithme (souvent, on place un compteur *cpt* dans la boucle la + imbriquée),
- dans cette même méthode *void testAlgoEfficacite()*, mesurer également le temps d'exécution,
- consigner les mesures dans un tableau : pour chaque taille de texte en ligne, notez la valeur de *cpt* en colonne 1 et le temps d'exécution (préciser l'unité de temps) en colonne 2,
- mettre vos résultats en graphiques et commenter vos graphiques (voir + loin) : des résultats sans commentaires valables seront considérés comme inexistantes,
- faire vos mesures sur des textes de nature différentes :
 - textes aléatoires,
 - textes répétitifs (ex. séquences de "AAAAAA..." ou "BBBTTTAACCBTTTTEEEEEAAA").

Structure de données

On fera usage d'un type *ArrayList* de l'API Java qui est un tableau dynamique. Regardez dans l'API Java (<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>) les méthodes de *ArrayList* à votre disposition et en particulier : *size*, *add*, *clear*, *set* et *get*. Pour afficher le contenu d'une *ArrayList* *theList*, il suffit d'écrire l'instruction *System.out.println (theList.toString())*.

Dans notre cas :

- Le texte à examiner sera mémorisé dans un tableau de type *ArrayList<Character>*. Ce qui veut dire que dans chacune des cases de ce tableau, on trouve un caractère du texte (chaque case étant accessible à un indice compris entre zéro et *size()-1*).



- Le motif à rechercher sera une chaîne de caractères de type *String*.
- Le résultat de la recherche sera un tableau d'entiers de type *ArrayList<Integer>* où chaque entier est l'indice dans le texte où commence le motif. On va considérer, **par pure convention**, que le tout premier caractère du texte se trouve à l'indice **1** et que le tout dernier caractère se trouve à l'indice **size()**.
- Dans cette SAE, la signature d'un algorithme de recherche d'un motif dans du texte sera (quel que soit l'algorithme) :

ArrayList<Integer> nameAlgoSearch (ArrayList<Character> text, String pattern)

- Il y aura 2 manières possibles de mémoriser du texte dans une *ArrayList<Character>* :
 - pour du texte court, utiliser *Arrays.asList (char[] tab)* en paramètre au constructeur de *ArrayList<Character>* :
exemple : *ArrayList<Character> text = new ArrayList<> (Arrays.asList ('E', 'x', 'e', 'm', 'p', 'l', 'e'));*
 - pour du texte long (voir très long dans le cas d'une étude de l'efficacité, 1000 caractères par exemple), on peut faire appel à une méthode qui remplit le tableau *ArrayList<Character> text* automatiquement avec des caractères tirés au hasard sur une plage d'entiers correspondant aux codes ASCII des lettres de l'alphabet (par exemple) ou autres caractères.

Développement

- Pour chaque algorithme développé (4 au total) il correspond 1 seule classe *NomClasse.java* qui contient nécessairement la méthode *void principal()*, point d'entrée de l'exécution, et qui se lance avec la commande *java Start NomClasse*. La méthode *void principal()* contient l'appel au(x) *test(s)* et à *testAlgoEfficiency()*.
- Pour chaque version développée, votre classe doit contenir au minimum les méthodes qui testent vos algorithmes (*testAlgo* et *testCasAlgo*) et qui évaluent leur efficacités (*testAlgoEfficiency()*). Vous pouvez évidemment développer des méthodes supplémentaires mais toute méthode ajoutée doit être nécessairement testée.
- **Aucune** variable globale, hormis le compteur *cpt* (type *long*), ne peut être déclarée, ce qui veut dire que vos tableaux *text* et *pattern* doivent chaque fois être passé en paramètre ou déclaré en variables locales.

6. Les algorithmes

6.1. L'algorithme naïf

La classe à développer pour cet algorithme se nomme ***NaiveAlgo.java***.

La méthode qui met en œuvre l'algorithme naïf a comme signature :

ArrayList<Integer> naiveAlgo (ArrayList<Character> text, String pattern)

La méthode de test : *void testNaiveAlgo()* et *void testCasNaiveAlgo(...)*

L'évaluation de l'efficacité : *void testNaiveAlgoEfficiency()*

Ci-dessous, voici son fonctionnement.



soit le texte : "ABABAABCBAB"

soit le motif : "BA"

L'algorithme naïf compare les 2 lettres du motif avec chaque groupe de 2 lettres du texte en commençant par le début du texte.

Etape 1

texte **A B** A B A A B C B A B

motif **B A**

Pas de correspondance (la première lettre ne correspond déjà pas) donc on décale le motif (vers la droite) de **1 seule case**.

Etape 2 (après décalage à droite)

texte A **B A** B A A B C B A B

motif **B A**

Correspondance lettre par lettre de tout le motif avec le texte => on a trouvé une apparition de ce motif en case 2.

On continue ensuite sur le texte en décalant le motif d'une case (vers la droite). On trouvera ensuite les positions 4 et 9.

La méthode qui met en œuvre l'algorithme doit renvoyer *toutes* les positions dans le texte du motif lorsqu'il y a correspondance (donc plusieurs positions si il y a plusieurs correspondances).

Travail demandé

Coder l'algorithme, le tester puis évaluer son efficacité. Consigner les mesures dans un tableau et mettre vos résultats en graphiques. Commenter vos graphiques.

Evaluer **$f(n)$ exacte** et en déduire l'expression de Θ .

6.2. L'algorithme de Knuth-Morris-Pratt (KMP)

La classe à développer pour cet algorithme se nomme **KMPAlgo.java**.

La méthode qui met en œuvre l'algorithme *KMP* a comme signature :

`ArrayList<Integer> kmpAlgo (ArrayList<Character> text, String pattern)`

La méthode de test : `void testKmpAlgo()` et `void testCasKmpAlgo(...)`

L'évaluation de l'efficacité : `void testKmpAlgoEfficiency()`

KMP sur un exemple

Ci-dessous, voici comment celui-ci fonctionne sur un cas concret.

soit le texte : "ABABAABCBAB"

soit le motif : "ABABACA"



L'algorithme *KMP* va adopter une technique de décalage à droite supérieur à 1 (quand c'est possible) pour accélérer la comparaison du motif avec le texte.

Dans l'exemple ci-dessus, on va montrer que l'on peut décaler de plus de 1 seule case. Pour la clarté de l'explication on écarte les caractères entre eux.

indice k	1	2	3	4	5		6	7	8	9	10	11
t(k) =	A	B	A	B	A		A	B	C	B	A	B
m(i) =	A	B	A	B	A		C	A				
indice i	1	2	3	4	5		6	7				

En examinant une correspondance de gauche à droite de m avec t en commençant à $k = i = 1$, on constate que les 5 premières lettres de m « ABABA » coïncident avec t mais que la 6ème (la lettre 'C') ne coïncide pas.

Sur $m[1...5]$ = « ABABA », on voit que la chaîne $m[1...3]$ = « ABA » se répète en $m[3...5]$. Donc on peut décaler m de 2 cases vers la droite sur le texte t . **Le décalage est donc supérieur à 1** dans l'exemple ci-dessus, ce qui donne :

indice k	1	2	3	4	5		6	7	8	9	10	11
t(k) =	A	B	A	B	A		A	B	C	B	A	B
m(i) =				A	B	A		B	A	C	A	
indice i				1	2	3		4	5	6	7	

De plus, comme on sait que le **préfixe** « ABA » de m coïncide avec t , il suffit de recommencer la comparaison à partir de $k = 6$ et $i = 4$. Pour k (indice de comparaison dans le texte) on gagne donc 5 cases au final (encore mieux que 2 cases).

Calcul du décalage de 2 cases pour l'exemple ci-dessus.

Le problème du nombre de cases à décaler se ramène au problème de la comparaison du motif m avec lui-même car on se trouve bien dans une situation où la chaîne à étudier (« ABABA ») est exactement la même entre m et t sur la portion $m[1...5] = t[1...5]$.

Cette chaîne (ici « ABABA ») à étudier sera nommée U . Elle est par définition le plus long préfixe de m qui coïncide avec t (attention, ne pas confondre m et U dans la suite).

Pour déterminer le décalage maximum de m avec lui-même, on se base sur une fonction « préfixe » appelée fp qui calcule pour chaque préfixe de U **égale à son suffixe** (de ce même U), la longueur de ce préfixe.

Règle : pour un U fixé, le décalage de m sera optimal quand la fonction préfixe fp est maximum.

Bien entendu le maximum de fp va dépendre de U (qui lui va dépendre de l'endroit où l'on se trouve dans le texte). C'est pourquoi il faut construire un tableau de correspondance entre chaque U possible et le fp maximum.



Voici ce tableau dans le cas du motif $m = \text{« ABABACA »}$ (bien le comprendre !).

indice i/v	1	2	3	4	5	6	7
m	A	B	A	B	A	C	A
$U = m[1..v]$ v de 1 à 7	A	AB	ABA	ABAB	ABABA	ABABAC	ABABACA
$fp\ max$	1	0	1	2	3	0	1

Calcul en détail de $fp\ max$ pour chaque U :

- $U = m[1..1] = \text{« A »}$ préfixe = suffixe = « A », $|\text{« A »}| = 1$ $fp\ max = 1$
- $U = m[1..2] = \text{« AB »}$ préfixe = suffixe = \emptyset , $|\emptyset| = 0$ $fp\ max = 0$
- $U = m[1..3] = \text{« ABA »}$ préfixe = suffixe = « A », $|\text{« A »}| = 1$ $fp\ max = 1$
- $U = m[1..4] = \text{« ABAB »}$ préfixe = suffixe = « AB », $|\text{« AB »}| = 2$ $fp\ max = 2$
- $U = m[1..5] = \text{« ABABA »}$ préfixe = suffixe = « A », $|\text{« A »}| = 1$ $fp\ max = \max(1, 3) = 3$
 préfixe = suffixe = « ABA », $|\text{« ABA »}| = 3$
- $U = m[1..6] = \text{« ABABAC »}$ préfixe = suffixe = \emptyset , $|\emptyset| = 0$ $fp\ max = 0$
- $U = m[1..7] = \text{« ABABACA »}$ préfixe = suffixe = « A », $|\text{« A »}| = 1$ $fp\ max = 1$

Exploitation de ce tableau : dans l'exemple de départ, sachant que $U = m[1..5] = \text{« ABABA »}$, le tableau nous donne un $fp\ max = 3$ (nombre de lettres tel que préfixe = suffixe de U).

L'algorithme doit donc au final :

- appliquer un décalage du motif m de 2 cases = $(|U| - fp\ max)$, et donc également un décalage de l'indice k sur le texte t de 2 cases,
- de plus, k peut faire un saut de $fp\ max = 3$ cases supplémentaires puisque ces 3 premières cases coïncident forcément avec le motif (k augmente donc de 5 cases au total $(3 + 2)$).

Généralisation

L'algorithme de *Knuth-Morris-Pratt* s'écrit d'une manière générale en 2 étapes.

Connaissant le motif m qu'il faut rechercher dans un texte t quelconque, il faut :

1. Construire le tableau de la fonction préfixe maximum ($fp\ max$) pour chaque préfixe U de m (il y a $|m|$ possibilités de chaînes différentes pour U).
2. Connaissant U et à l'aide du tableau précédent, appliquer un décalage du motif égale à $(|U| - fp\ max)$ cases et appliquer sur l'indice k qui parcourt le texte un décalage de $|U|$ cases.

Travail demandé

Coder l'algorithme, le tester puis évaluer son efficacité. Consigner les mesures dans un tableau et mettre vos résultats en graphiques. Effectuez vos mesures sur des textes de nature différentes : textes aléatoires, textes répétitifs. Commentez vos graphiques.



6.3. L'algorithme de Rabin-Karp

La classe à développer pour cet algorithme se nomme ***RabinKarpAlgo.java***.

La méthode qui met en œuvre l'algorithme de *Rabin-Karp* a comme signature :

ArrayList<Integer> rabinKarpAlgo (ArrayList<Character> text, String pattern)

La méthode de test : *void testRabinKarpAlgo()* et *void testCasRabinKarpAlgo(...)*

L'évaluation de l'efficacité : *void testRabinKarpAlgoEfficiency()*

Algorithme naïf

Partant de l'algorithme naïf vu précédemment, on sait que cette méthode naïve procède de la façon suivante :

- soit t le texte et m le motif à rechercher dans le texte de longueur $|m|$,
- il y aura correspondance du motif avec le texte à l'indice k dans le texte si :

$$t[k \dots (k + |m| - 1)] = m[1 \dots |m|] \quad (1)$$

Algorithme de Rabin-Karp

L'accélération de l'évaluation de la comparaison (1) passe par un calcul « d'empreinte » à l'aide d'une fonction dite de hachage (notée *hach*) qui, appliquée à la portion de chaîne de caractères donne une valeur entière (que l'on appelle « empreinte »).

L'expression (1) devient alors :

$$hach(t[k \dots (k + |m| - 1)]) = hach(m[1 \dots |m|]) \quad (2)$$

Comme $hach(m)$ est constant, il suffit de calculer le membre de gauche de (2) à chaque nouvel indice k dans le texte.

Si l'égalité (2) est vérifiée, alors il reste à effectuer une comparaison naïve caractère par caractère car plusieurs portions $t[k \dots (k + |m| - 1)]$ pourtant distinctes peuvent avoir éventuellement la même « empreinte ». Si tous les caractères correspondent, on a trouvé une occurrence du motif dans le texte.

Exemple de fonction de hachage très simple :

Si on donne à chaque lettre de l'alphabet (A...Z) un numéro compris entre 1 (A) et 26 (Z), on obtient par addition une valeur qui est l'empreinte de la chaîne de caractères (bien sûr ici ça ne fonctionne que pour les caractères A à Z).

si $t[k \dots (k + |m| - 1)] = \text{"ABABACA"}$

alors $hach(\text{"ABABACA"}) = 1 + 2 + 1 + 2 + 1 + 3 + 1 = 11$

Amélioration1

Calculer $hach(t[k \dots (k + |m| - 1)])$ tel quel n'est pas intéressant par rapport à la version naïve car on effectue en réalité le même nombre d'opérations élémentaires.

Il « suffit » de remarquer qu'à chaque nouvel indice du texte (k), il ne faut pas recalculer l'entièreté du membre de gauche de (2).

En effet, en $k+1$, l'expression (2) devient :



$$hach(t[(k+1)...(k+|m|)]) = hach(t[k...(k+|m|-1)]) + hach(t[k+|m|]) - hach(t[k]) \quad (3)$$

Et donc il suffit de calculer $hach(t[k+|m|])$ et $hach(t[k])$ puisque $hach(t[k...(k+|m|-1)])$ est le membre de gauche de (2) qui a déjà été calculé à l'étape k précédente.

Cette façon de procéder s'appelle « utiliser une fonction de hachage déroulante ».

Amélioration2

Rappel : un nombre n en base 10 exprimé en base b est une suite de chiffres $a_n... a_1 a_0$ tel que :

$$a_n b^n + ... + a_1 b^1 + a_0 b^0 = n \text{ (exprimé en base 10)}$$

On admettra qu'une bonne fonction de hachage ($hach$) déroulante qui se fonde sur la somme des codes ASCII exprimés en base b est la suivante :

Soit $hach(t+1)$: la nouvelle empreinte correspondant à la nouvelle portion du texte $t[(k+1)...(k+|m|)]$

Soit $hach(t)$: l'empreinte précédente calculée sur la portion $t[k...(k+|m|-1)]$

La nouvelle fonction de hachage déroulante adoptée par *Rabin-Karp* s'écrit :

$$hach(t+1) = \{ [hach(t) - (t[k] * b^{(|m|-1)})] * b \} + t[k+|m|] * b^0 \quad (4)$$

Exemple concret :

indice k 1 2 3 4 5 6 7 8 9 10 11
 soit le texte (t) : a b r a c a d a b r a
 soit le motif (m) « aba » de 3 caractères ($|m| = 3$)
 soit $b = 101$ (proposée par *Rabin-Karp*)

On suppose que l'on démarre la comparaison du motif avec le texte donc $k = 1$. On utilise la formule (2) pour savoir si m coïncide avec t sur $[1...3]$.

L'empreinte du motif m est constante et s'écrit en base $b = 101$:

sachant que : ASCII de 'a' = 97, ASCII de 'b' = 98 et ASCII de 'c' = 99

$$hach(\text{« aba »}) = (97 * 101^2) + (98 * 101^1) + (99 * 101^0) = 999.494$$

En $k = 1$, on a pour $t[1...3] = \text{« abr »}$ (et sachant que ASCII de 'r' = 104) :

$$hach(\text{« abr »}) = (97 * 101^2) + (98 * 101^1) + (104 * 101^0) = 999.509$$

Donc en $k = 1$, le motif ne coïncide pas ($999.494 \neq 999.509$).

En $k = 2$, $t[2...4] = \text{« bra »}$ et on fait usage de la formule (4) pour calculer $hach(\text{« bra »})$ en fonction de $h(\text{« abr »})$ calculé précédemment :

$$hach(\text{« bra »}) = \{ [hach(\text{« abr »}) - (t[1] * 101^2)] * 101 \} + (t[4] * 101^0)$$

$$hach(\text{« bra »}) = \{ [999.509 - (97 * 101^2)] * 101 \} + (97 * 101^0) = 1.011.309$$

Donc en $k = 2$, le motif ne coïncide pas non plus ($999.494 \neq 1.011.309$).

**Travail demandé**

Coder l'algorithme en utilisant la fonction de hachage déroulante de l'amélioration2 (formule (4) ci-dessus). La comparaison avec le motif (formule (2)) et la confirmation caractère par caractère doit se faire comme expliqué précédemment.

Tester l'algorithme puis évaluer son efficacité. Consigner les mesures dans un tableau et mettre vos résultats en graphiques. Effectuez vos mesures sur des textes de nature différentes : textes aléatoires, textes répétitifs. Commentez vos graphiques.

6.4. L'algorithme de Boyer-Moore

La classe à développer pour cet algorithme se nomme **BoyerMooreAlgo.java**.

La méthode qui met en œuvre l'algorithme de *Boyer-Moore* a comme signature :

`ArrayList<Integer> boyerMooreAlgo (ArrayList<Character> text, String pattern)`

La méthode de test : `void testBoyerMooreAlgo()` et `void testCasBoyerMooreAlgo(...)`

L'évaluation de l'efficacité : `void testBoyerMooreAlgoEfficiency()`

Principe de l'algorithme

Contrairement aux autres méthodes de recherche d'un motif dans un texte, le principe est de comparer les caractères à rebours (en commençant donc par le **dernier caractère** du motif).

Exemple :

soit le texte : `"stupid_spring_string"`

soit le motif : `"string"`

Voici comment fonctionne la méthode en commençant par la fin du motif (pour la clarté de l'explication on écarte les caractères entre eux).

Etape 1 : on démarre la comparaison en $k = 1$

indice k 1 2 3 4 5 6 7 8 ...

t= s t u p i **d** _ s p r i n g _ s t r i n g

m= s t r i n **g**

Le dernier caractère du motif est « g » qui est différent de « d » dans le texte. De plus « d » n'apparaît pas dans *m* (motif). On est certain qu'il n'y aura aucune correspondance parfaite entre *m* et *t* (texte) sur cette portion. Donc on peut décaler le motif de $|m|$ cases soit 6 cases (donc nettement mieux qu'un décalage de 1 case dans la méthode naïve).

Etape 2 : après décalage de 6 cases

t= s t u p i d _ s p r i **n** g _ s t r i n g

m= s t r i n **g**

$g \neq n$ MAIS *n* se trouve en position 5 dans *m*. A ce moment-là on fait coïncider les deux *n* (de *t* et de *m*) car on pourrait potentiellement avoir correspondance complète. On décale donc le motif de 1 case.

**Etape 3 : après décalage de 1 case**

t = s t u p i d _ s **p** r i n g _ s t r i n g
 m = s **t** r i n g

Il y a correspondance des « g », « n », « i » et « r » mais pas de « t » ($\neq p$). Avant le « t » il n'y a aucun « p ». On est donc certain qu'il n'y aura aucune correspondance parfaite entre m et t sur cette portion [1,2] du motif. Donc on peut décaler le motif de 2 cases.

Etape 4 : après décalage de 2 cases

t = s t u p i d _ s p r i n g _ **s** t r i n g
 m = s t r i n **g**

$g \neq s$ MAIS « s » se trouve en position 1 dans m . A ce moment-là on fait coïncider les deux « s » (de t et de m) car on pourrait potentiellement avoir correspondance complète. On décale donc le motif de 5 cases (encore une fois nettement mieux qu'un décalage de 1 case dans la méthode naïve).

Etape 5 : après décalage de 5 cases

t = s t u p i d _ s p r i n g _ **s t r i n g**
 m = s t r i n **g**

On compare les caractères en commençant par la fin de m . Comme il y a correspondance complète, on a trouvé une apparition du motif dans le texte.

Et ainsi de suite (décalage de 1 case après l'étape 5) si on n'est pas en fin de texte.

Le tableau des sauts (tableau 1)

Le problème du nombre de cases suivant lequel le motif peut se décaler, se ramène en réalité à une analyse du motif avec lui-même (donc indépendamment du texte).

En partant de la fin du motif, dès que l'on rencontre un caractère du motif $m[k]$ qui ne coïncide **pas** au caractère du texte $t[k]$ ($m[k] \neq t[k]$), il faut, en commençant à l'indice $(k-1)$, parcourir le motif à rebours pour trouver la première occurrence d'un caractère dans ce motif identique à $t[k]$. Deux possibilités :

- soit on ne trouve pas de caractère dans le motif identique à $t[k]$: on décale alors le motif d'un nombre de cases égale à la position k du caractère $m[k]$ tel que $m[k] \neq t[k]$,
- soit on trouve en case « p » ($1 \leq p < k$) un caractère identique à $t[k]$: on décale alors le motif d'un nombre de cases égale à $k - p$.

Ne connaissant pas à priori $t[k]$ (puisque ça va dépendre du texte), il faut construire un tableau qui reprend tous les caractères du motif qui pourraient se retrouver dans le texte. Pour chaque caractère du motif qui se retrouve dans le texte, le tableau précise le décalage du motif à effectuer pour une position donnée k tel que $m[k] \neq t[k]$ (en commençant à rebours).

Exemple d'un tel tableau pour le motif « wikipedia ». Ce tableau correspond uniquement à la situation où c'est la lettre « a » de « wikipedia », en position $k = 9$, qui est tel que $m[k] \neq t[k]$.

caractère du motif (1)	décalage (2)
a	0 (cas particulier sans intérêt)
i	1 (première occurrence à rebours)
d	2
e	3



p	4
k	6
w	8
autre	9 (cas particulier : aucun $t[9]$ dans m)

(1) : caractère dont on veut calculer la distance en nombre de cases par rapport à $m[9] = \text{« a »}$, tel que $m[9] \neq t[9]$ (en supposant une comparaison du motif en tout début de texte)

(2) : décalage du motif en nombre de cases :

- soit égale à 9 (= position de la lettre dans le motif m tel que $m[k] \neq t[k]$) car le caractère du texte $t[9]$ ne se retrouve pas dans le motif),
- soit égale $k - p$ où p ($1 \leq p < k$) est la position dans le motif du caractère égale à $t[k]$.

Le tableau du bon suffixe (tableau 2)

En premier lieu, définissons quelques notations sur un exemple (pour la clarté de l'explication on écarte les caractères entre eux) :

				$t[k]$	$t[k+1]$		$t[k+p-i]$
texte $t = \dots$	a	a	b	c	b	a	b...
motif $m =$	a	a	b	a	b	a	b
	$m[1]$			$m[i]$	$m[i+1]$		$m[p]$
					$u = m[(i+1) \dots p]$		

On a :

i : l'indice sur le motif $m[1 \dots p]$ qui commence à 1 et tel que $|m| = p$

k : l'indice sur le texte qui commence également à 1

u est par définition le **suffixe de m qui coïncide avec t**

donc $u = m[(i+1) \dots p] = t[(k+1) \dots (k+p-i)]$ ET $m[i] \neq t[k]$

Principe général (qui sera précisé ultérieurement) :

- Règle 1 - si le suffixe u se retrouve dans le motif (sur une plage allant de $m[1]$ à $m[p-1]$) alors un décalage > 1 est possible.
- Règle 2 - si le suffixe u ne se retrouve pas dans le motif alors il y a 2 possibilités :
 - soit on trouve un préfixe z de m égale à un suffixe de u : dans ce cas le décalage est égale à $(|m| - |z|)$,
 - soit aucun préfixe z de m égale à un suffixe de u n'existe : dans ce cas le décalage est maximum et égale à $|m|$.

Complément pour la règle 1 : le décalage sera maximum (égale à $|m|$) si le motif identique retrouvé sur la plage $m[1]$ à $m[p-1]$ est précédé du **même caractère** que celui qui précède u (i.e. $m[i]$).

Voici quelques exemples :

**Exemple 1**

texte $t =$ a m i m **a** p o

motif $m =$ c o n d **a**

$u = a$ (et $d \neq m$)

règle 1 : il n'existe aucun ($u = \text{« } a \text{ »}$) sur $m[1...4]$

règle 2 : aucun préfixe de m n'existe

donc : décalage maximum de $|m| = 5$ cases

Exemple 2

texte $t =$ o b o **b a b** a b o b a

motif $m =$ x b a **b a b**

$u = bab$ (et $a \neq o$)

règle 1 : il existe ($u = \text{« } bab \text{ »}$) sur $m[2...4]$ et ce « bab » est précédé de « x » ($x \neq a$)

donc : décalage de 2 cases pour faire coïncider « bab » sur $m[2...4]$ avec le texte

Exemple 3

texte $t =$ s a o n a n a r **a u n a** o n a

motif $m =$ n a o n a s **a u n a**

$u = auna$ (et $s \neq r$)

règle 1 : il n'existe aucun ($u = \text{« } auna \text{ »}$) sur $m[1...9]$

règle 2 : il existe le préfixe $z = \text{« } na \text{ »}$ de m également suffixe de u

donc : décalage de $|m| - |z|$ cases (soit $10 - 2 = 8$ cases) pour faire coïncider « na » sur $m[1...2]$ avec le texte

On constate, à travers les exemples, qu'il « suffit » de comparer le motif m avec lui-même et ceci indépendamment du texte.

La bonne façon de procéder est alors de construire un tableau (tableau des bons suffixes) qui énumère tous les décalages possibles suivant les différentes chaînes possibles pour le suffixe u du motif m .

Soit le motif $m =$ « aababab » ($|m| = 7$)

motif m a a b a b a b

(1) indice i 0 1 2 3 4 5 6 7

(2) $d(i)$ 7 7 7 2 7 4 7 1

(1) i est tel que $u = m[(i+1)..... p] = t[(k+1)....(k+p-i)]$ (et $m[i] \neq t[k]$) est un suffixe de m qui coïncide avec t



(2) $d(i)$ est le décalage du motif m en fonction de u

Pour l'exemple de $m = \text{« aababab »}$ ci-dessus, voici le détail du calcul des différents $d(i)$ obtenus.

Pour $i = 0$ (cas particulier)

$u = \text{aababab}$ ce qui signifie que le motif est trouvé (cas particulier), le décalage est maximum donc $d(0) = |m| = 7$

Pour $i = 1$

$u = \text{ababab}$ (et $m[i] = a$)

aucun « ababab » trouvé sur $m[1 \dots 6]$ et aucun suffixe z de u également préfixe de m

donc $d(1) = |m| = 7$ (décalage maximum)

Pour $i = 2$

$u = \text{babab}$ (et $m[i] = a$)

aucun « babab » trouvé sur $m[1 \dots 6]$ et aucun suffixe z de u également préfixe de m

donc $d(2) = |m| = 7$

Pour $i = 3$

$u = \text{abab}$ (et $m[i] = b$)

« abab » trouvé sur $m[2 \dots 5]$ et précédé de « a » \neq « b »

donc $d(3) = 2$ cases pour faire coïncider « abab » sur $m[2 \dots 5]$ avec le texte

Pour $i = 4$

$u = \text{bab}$ (et $m[i] = a$)

aucun « bab » sur $m[1 \dots 6]$ n'est possible car tous précédés de « a »

et aucun suffixe z de u également préfixe de m

donc $d(4) = |m| = 7$ (décalage maximum)

Pour $i = 5$

$u = \text{ab}$ (et $m[i] = b$)

seul « ab » sur $m[2 \dots 3]$ est possible car précédé de « a » \neq « b »

donc $d(5) = 4$ cases pour faire coïncider « ab » sur $m[2 \dots 3]$ avec le texte

Pour $i = 6$

$u = \text{b}$ (et $m[i] = a$)

aucun « b » sur $m[1 \dots 6]$ n'est possible car tous précédés de « a »

et aucun suffixe z de u également préfixe de m

donc $d(6) = |m| = 7$ (décalage maximum)

Pour $i = 7$ (cas particulier)

$u = \text{« rien »}$ (et $m[i] = b$)

Ceci revient à ne trouver aucun suffixe de m qui coïncide avec le texte donc on ne peut que décaler de 1 case (méthode naïve).

donc $d(7) = 1$



Mise en oeuvre

Pour mettre en œuvre correctement l'algorithme de *Boyer-Moore* il faut après chaque décalage :

- construire le tableau 1 (des sauts) pour en déduire un potentiel décalage D1 possible
- construire le tableau 2 (des bons suffixes) pour en déduire un deuxième décalage D2 possible
- choisir au final le + grand des décalages entre D1 et D2

Travail demandé

Tester l'algorithme puis évaluer son efficacité. Consigner les mesures dans un tableau et mettre vos résultats en graphiques. Effectuez vos mesures sur des textes de nature différentes : textes aléatoires, textes répétitifs. Commentez vos graphiques.

7. Modalités pratiques et rendu de votre travail

Le projet se déroule en binôme (ou seul mais **pas en trinôme**) au sein d'un même groupe TD.

Début de la SAE : semaine 48.

Un créneau par semaine en autonomie sur les semaines 49, 50, 02.

Deux créneaux en autonomie sur la semaine 03/2026.

Créneaux encadrés : un en semaine 48, un en semaine 51 et un créneau en semaine 03.

Date limite du rendu : **vendredi 16 janvier 2026 à 23h55 au + tard (attention : -2 points de malus si retard)**.

Votre travail sera rendu sur Moodle sous forme d'une archive *nomBinôme1_nomBinôme2.zip* (pas de *.rar* !) qui contiendra plusieurs fichiers :

- Tous vos sources *java* correspondant aux 4 versions des algorithmes de recherche d'une sous-chaîne dans du texte (1 classe par version avec la bonne orthographe !). Ces sources *java*, doivent être soigné, documenté (*javaDoc*). Ils doivent non seulement contenir le code correspondant à la bonne version mais ces sources doivent également contenir le test des algorithmes de même que le test d'efficacité. **La javaDoc doit être écrite en anglais.**
- Un document (maximum 15 pages) au format PDF nommé *nomBinôme1_nomBinôme2.pdf*, qui contient :
 - les objectifs du travail en introduction,
 - l'ensemble des courbes obtenues pour chaque version + les tableaux qui reprennent pour chaque texte de grandes tailles (500...1000 caractères etc.) la valeur du compteur *cpt* et le temps d'exécution,
 - une conclusion finale qui est un comparatif de l'efficacité de chacun des 4 algorithmes (ce comparatif tiendra compte de la nature du texte : textes aléatoires, textes répétitifs).

Evaluation finale de cette SAE :

- 50% : note du projet « Recherche sous-chaîne »
- 50% : note de contrôle de votre niveau en *Java*