Introduction

La recherche d'occurrences d'une sous-chaîne de caractères, désignée comme motif, dans une chaîne de caractères plus longue, désignée comme texte, est un sujet très important en informatique. C'est un composant de base des logiciels de traitement de texte et une problématique incontournable dans des domaines très divers : correcteurs orthographiques, bio-informatique (étude du génome), traitement du signal, analyse de fichier (détection de virus) ...
Ce TP est inspiré du document ressources pour l'enseignement de NSI rédigé par Laurent Cheno, Inspecteur Général désormais à la retraite. Les illustrations en sont tirées.
La partie sur l'algorithme de Boyer-Moore complet est directement tirée du manuel de MPI/MP2I de la collection tortue rédgé par Balabonski, Filliâtre, Conchon, N'Guyen, Sartre.

Méthode

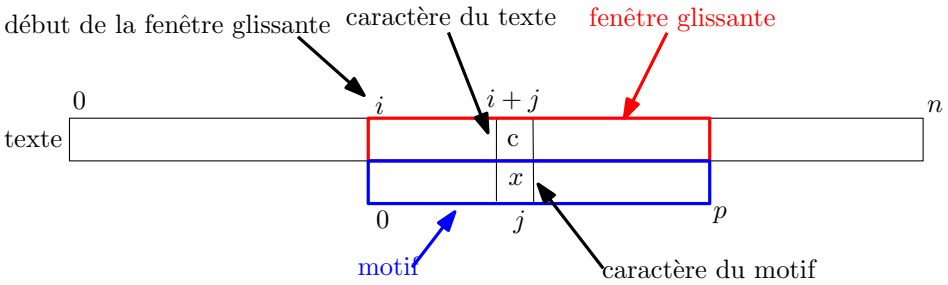
Récupérer l'archive `tp_motif.zip` et l'extraire dans son dossier personnel.
Ouvrir le squelette de code `tp_motif.py` dans un environnement de développement Python.
On séparera les codes de chaque exercice dans des **cellules** qui peuvent s'exécuter de façon indépendante.

- Dans Spyder, on introduit une cellule avec la séquence de caractères `#%%`.
- Dans Pyzo, on introduit une cellule avec la séquence de caractères `##`.

1 Présentation du problème

Méthode

Un problème classique en algorithmique est la recherche de la première occurrence d'une sous-chaîne de caractères, le **motif**, dans une chaîne plus longue, le **texte**.
En notant p la longueur du motif et n celle du texte, une **précondition** est $0 \leqslant p \leqslant n$ et on a souvent n beaucoup plus grand que p.
Les trois algorithmes présentés procèdent par **fenêtre glissante** : on déplace dans le texte une fenêtre de taille p sur laquelle on aligne le motif, et on compare les caractères de la fenêtre et du motif à l'intérieur de la fenêtre. Si tous les caractères correspondent on a trouvé une occurrence du motif, sinon le premier caractère du texte où il y a une différence s'appelle le **mauvais caractère**.



On déplacera toujours la fenêtre glissante de gauche à droite mais à l'intérieur de la fenêtre on va comparer les caractères de gauche à droite pour ***l'algorithme naïf*** *et de droite à gauche pour les* ***algorithmes de Boyer-Moore-Horspool****.*

On utilise les notations suivantes :

- on note `i` la position de la fenêtre dans le texte : c'est l'index du premier caractère du texte qui apparaît dans la fenêtre ;
- on note `j` l'index dans le motif du caractère que nous comparons avec son analogue du texte : on compare `motif[j]` avec `texte[i + j]`.

2 Algorithme naïf ou force brute

Méthode

Algorithme naïf ou force brute

☞ On déplace la **fenêtre glissante** de gauche à droite dans le texte.

☞ Pour chaque position `i` de la **fenêtre glissante**, on compare les caractères superposés du motif et du texte de gauche à droite et on s'arrête soit parce ce qu'on a dépassé la fin de la fenêtre, soit parce qu'on a une différence avec un **mauvais caractère** du texte.

☞ Si on s'arrête parce ce qu'on a dépassé la fin de la fenêtre on a trouvé une occurrence du motif en position `i` dans le texte, sinon on décale la fenêtre glissante de 1 caractère vers la droite et on recommence.

Par exemple, dans la situation suivante,

```
    i   .  .  . 12 13 14 15 16 17 18 19 20 21 22 .  .  .
texte   .  .  .  a  b  r  a  c  a  d  a  b  r  a .  .  .
motif                  a  d  a
    j                  0  1  2
```

on compare le `'a'` du motif avec le `'r'` du texte, obtenant immédiatement une différence : on peut avancer la fenêtre en incrémentant `i`, qui passe de 14 à 15.
Dans la nouvelle fenêtre, le premier caractère coïncide bien :

```
    i   .  .  . 12 13 14 15 16 17 18 19 20 21 22 .  .  .
texte   .  .  .  a  b  r  a  c  a  d  a  b  r  a .  .  .
motif                  a  d  a
    j                  0  1  2
```

et on incrémente `j` pour tester les caractères suivants, `'d'` et `'c'` :

```
    i   .  .  . 12 13 14 15 16 17 18 19 20 21 22 .  .  .
texte   .  .  .  a  b  r  a  c  a  d  a  b  r  a .  .  .
motif                  a  d  a
    j                  0  1  2
```

Un outil de visualisation par Laurent Abbal :

https://boyer-moore.codekodo.net/recherche_naive.php

Exercice 1 *Algorithme naïf*

Si on utilise le *slicing*, une version de la recherche naïve d'un motif dans un texte pourrait être :

```
def premiere_occurrence_naif_slicing(motif, texte):
    """
    Paramètres : motif et texte deux chaînes de caractères
    Précondition : 0 < len(motif) <= len(texte)
    Valeur renvoyée : un entier
    Postcondition : renvoie l'index de la première occurence de motif dans
          texte
    ou -1 s'il n'y a pas d'occurence
    """
    p = len(motif)
    n = len(texte)
    assert 0 < p <= n #précondition
    i = 0
    for i in range(0, n - p + 1):
        if texte[i:i+p] == motif:
            return i
    return -1
```

1. Dans le fichier `tp_motif.py`, ajouter un jeu de tests unitaires pour la fonction précédente, en essayant de couvrir le plus de cas possibles (pas de correspondance, correspondance au début, à la fin ...).

2. On veut remplacer le test `texte[i:i+p] == motif` par `recherche_motif_pos(motif, texte, pos)`.

   Écrire la fonction dont on donne la spécification ci-dessous.

   Proposer une nouvelle version `premiere_occurence_naif(motif, texte)` de la recherche naïve de motif.

   ```
   def recherche_motif_pos(motif,texte,pos):
       """Paramètre : motif et texte deux chaînes de caractères et pos
           un entier
       Préconditions: 0<=len(motif)<=len(texte) et 0<=pos+len(motif)<=
           len(texte)
       Valeur renvoyée : un booléen
       Postcondition : détermine si motif == texte[i:i+len(motif)] sans
           slicing"""
   ```

3. Modifier les fonctions précédentes pour que la recherche de motif renvoie un `tuple` constitué de l'index de la première occurrence et du nombre de comparaisons effectué.

   ```
   >>> premiere_occurrence_naif_trace('aaabbba','aabaaabaabcaaabbba')
   (11, 31)
   ```

4. Soit p et n des entiers naturels avec $1 \leqslant p \leqslant n$.

Quel est le nombre de comparaisons effectué par l'algorithme de recherche naïf pour la recherche du motif `'a' * (p-1) + 'b'` dans le texte `'a' * n`?

Déterminer le nombre de comparaisons effectué dans le **pire des cas**, par l'algorithme de recherche naïf d'un motif de longueur `p` dans un texte de longueur `n`.

5. Compléter la fonction `nombre_occurrence(motif, texte)` fournie dans `tp_recherche.py`.

```
def nombre_occurrence(motif, texte):
    """Paramètres : motif et texte deux chaînes de caractères
    Précondition : 0 < len(motif) <= len(texte)
    Valeur renvoyée : un entier
    Postcondition : renvoie le nombre d'occurences de motif dans
        texte"""
    p, n = len(motif), n = len(texte)
    assert 0 < p <= n #précondition
    # à compléter
    return c
```

Effectuer quelques tests sur le fichier `LeRougeEtLeNoir.txt` :

```
Première occurrence du motif 'Julien' dans 'LeRougeEtLeNoir.txt' :
    25377
Nombre d'occurrences du mot 'Julien' dans 'LeRougeEtLeNoir.txt' :
    1908
Première occurrence du motif 'Mme de Rênal' dans 'LeRougeEtLeNoir.txt
    ' : 11917
Nombre d'occurrences du mot 'Mme de Rênal' dans 'LeRougeEtLeNoir.txt'
     : 412
Première occurrence du motif 'Mathilde' dans 'LeRougeEtLeNoir.txt' :
    484768
Nombre d'occurrences du mot 'Mathilde' dans 'LeRougeEtLeNoir.txt' :
    358
```

6. Considérons la recherche du motif `'aabaac'` dans le texte `'aabaabaac'`.

L'algorithme naïf, en décalant la fenêtre de 1 cran vers la droite à chaque itération, consomme 15 comparaisons :

```
Position dans texte : 0
Texte : aabaabaac
Motif : aabaac
------------------------------------------------------------------
Position dans texte : 1
Texte : aabaabaac
Motif :  aabaac
------------------------------------------------------------------
Position dans texte : 2
Texte : aabaabaac
Motif :   aabaac
------------------------------------------------------------------
Position dans texte : 3
```

```
Texte : aabaabaac
Motif :    aabaac
------------------------------------------------------------
```

Commentez la trace d'exécution d'un autre algorithme donnée ci-dessous. Comment pourrait-on améliorer les décalages de la fenêtre ?

```
Position dans texte : 0
Texte : aabaabaac
Motif : aabaac
Position suivante : 3
------------------------------------------------------------
Position dans texte : 3
Texte : aabaabaac
Motif :    aabaac
```

3 Algorithme de Boyer-Moore, simplification de Horspoole

Méthode

Les algorithmes de Boyer-Moore procèdent comme l'algorithme naïf en déplaçant la fenêtre du motif de gauche à droite dans le texte mais dans l'examen de la correspondance à l'intérieur de la fenêtre, ils s'en distinguent en parcourant les caractères de droite à gauche.
Nous présentons d'abord la version simplifiée par Horspool.

Algorithme de Boyer-Moore, simplification de Horspool

- Pour chaque position `i` de la **fenêtre glissante**, on compare les caractères superposés du motif et du texte de droite à gauche, dans l'autre sens que pour l'algorithme naïf. On s'arrête soit parce ce qu'on a dépassé la début de la fenêtre, soit parce qu'on a une différence avec un **mauvais caractère** du texte :
  - Si on s'arrête sur un **mauvais caractère** alors on s'intéresse au dernier caractère de la fenêtre `texte[i+p-1]` et on recherche s'il apparaît dans le motif avant cette dernière position. On prend son occurrence la plus à droite dans le motif, c'est-à-dire la plus proche du bord droit de la fenêtre. Si on note `k` sa position dans le motif, on décale la fenêtre de `p - 1 - k` pour amener `motif[k]` en face de `texte[i+p-1]` (bord droit de la fenêtre précédente).
  - S'il n'y a pas d'occurrence, on décale de la longueur du motif.
- Ainsi on peut améliorer l'algorithme naïf, parce qu'on peut avoir des décalages vers la droite supérieurs à 1. La complexité est difficile à étudier car elle dépend du motif et du texte.

Un outil de visualisation de l'algorithme de Horspool est proposé par Jean Diraison :

https://diraison.github.io/BMH/

Remarque 1

On désigne par **alphabet** l'ensemble des des caractères qui peuvent apparaître dans le texte.

Les recherches de décalage dans l'**algorithme de Boyer-Moore-Horspool** peuvent sembler coûteuses mais en fait les décalages ne dépendent que des positions des caractères de l'alphabet par rapport au bord droit du motif.

On peut donc réaliser un **prétraitement** en calculant pour chaque caractère de l'alphabet son décalage et en le stockant dans une table de type *dictionnaire*.

Exemple 1

Voici un exemple avec la recherche du motif `'babbb'` dans le texte `'bbabbcabbabbabbb'`. Le motif est trouvé en consommant 16 comparaisons alors qu'il en faut 29 avec l'algorithme naïf :

```
Table de décalage : {'b': 1, 'a': 3}
------------------------------------------------------------
Position dans texte : 0
Texte : bbabbcabbabbabbb
Motif : babbb
Position suivante : 1
------------------------------------------------------------
Position dans texte : 1
Texte : bbabbcabbabbabbb
Motif :  babbb
Position suivante : 6
------------------------------------------------------------
Position dans texte : 6
Texte : bbabbcabbabbabbb
Motif :       babbb
Position suivante : 7
------------------------------------------------------------
Position dans texte : 7
Texte : bbabbcabbabbabbb
Motif :        babbb
Position suivante : 8
------------------------------------------------------------
Position dans texte : 8
Texte : bbabbcabbabbabbb
Motif :         babbb
Position suivante : 11
------------------------------------------------------------
Position dans texte : 11
Texte : bbabbcabbabbabbb
Motif :            babbb
```

Exercice 2 *Implémentation de l'algorithme de Boyer-Moore-Horspool*

1. Donner la table de décalage pour le motif `NEEDLE`.

2. Déterminer comme dans la méthode, les différents décalages de fenêtre et le nombre de comparaisons effectués pour la recherche du motif `NEEDLE` dans le texte `FINDINAHAYSTACKNEEDLE`.

3. Compléter la fonction `calcul_decalage` ci-dessous dont on fournit la spécification et un jeu de tests unitaires.

```
def calcul_decalage(motif):
    """Paramètre : motif une chaîne de caractères
    Précondition : motif non vide
    Valeur renvoyée : un dictionnaire avec clef de type str et valeur
          de type int
    Postcondition : associe à chaque caractère de motif (sauf au
         dernier) le décalage
    de sa dernière occurence par rapport au dernier caractère du
         motif"""
    p = len(motif)
    assert p > 0
    decalage = {}
    decalage[motif[-1]] = p #cas particulier du dernier caractère
    for j in range(len(motif) - 1):
        ......
    return decalage

#tests unitaires
assert calcul_decalage("nouille") == {'e': 7, 'n': 6, 'o': 5, 'u': 4,
      'i': 3, 'l': 1}
assert calcul_decalage("GATTACA") == {'A': 2, 'G': 6, 'T': 3, 'C': 1}
```

4. Dans `tp_motif.py` compléter le squelette de la fonction donnée ci-dessous pour qu'elle réponde à sa spécification.

```
def premiere_occurence_bmh(motif, texte):
    """Paramètre : motif et texte deux chaînes de caractères
    Précondition : 0 < len(motif) <= len(texte)
    Valeur renvoyée : tuple d'entiers
    Postcondition : renvoie (index première occurence de motif dans
         texte, nombre de comparaisons)
    avec index = -1 si pas d'occurence de motif dans texte"""
    n, p = len(texte), len(motif)
    decalage = calcul_decalage(motif)
    i, nb_comparaisons = 0, 0
    while i <= n - p:
        j = p - 1
        nb_comparaisons += 1
        #à  compléter
    return (-1, nb_comparaisons)
```

5. On fournit un fichier `'LeRougeEtLeNoir.txt'` avec le texte intégral du roman Le rouge et le noir de Stendhal téléchargé sur le site du projet Gutenberg.

En exécutant le code ci-dessous avec la méthode `find` des chaînes de caractères on obtient 1020806 pour le nombre de comparaisons et 161411 pour la première occurrence du motif.

```
f = open('LeRougeEtLeNoir.txt', 'r', encoding='utf-8')
texte = f.read()
f.close()
motif = "Julien tremblait"
print(len(texte))
print(texte.find(motif))
```

Afficher la documentation de la méthode `find` avec `str.find`, comparer avec celles des fonctions de recherche implémentées dans le TP.

Tester les fonctions `premiere_occurence_naif_trace` et `premiere_occurence_bmh` sur cette recherche en affichant aussi le nombre de comparaisons. Comparer les performances des deux algorithmes.

6. Compléter avec le nombre de comparaisons effectuées.

Motif	Texte	Algorithme naïf	Algorithme de Boyer-Moore-Horspoole
`'aab'`	`'aaaaaaaaaa'`	...	...
`'bbb'`	`'aaaaaaaaaa'`	...	...

4 Algorithme de Boyer-Moore

Méthode

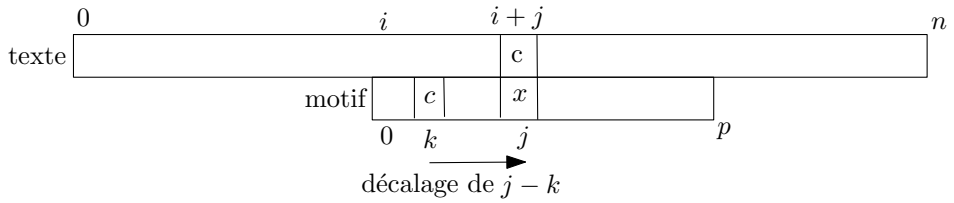
Par rapport à la version simplifiée de Horspool, dans **l'algorithme de Boyer-Moore** le décalage dépend de la position du caractère du motif qui est différent de celui qui correspond dans le texte à l'intérieur de la fenêtre glissante.

Algorithme de Boyer-Moore

- Pour chaque position `i` de la **fenêtre glissante**, on compare les caractères superposés du motif et du texte de droite à gauche, dans l'autre sens que pour l'algorithme naïf. On s'arrête soit parce ce qu'on a dépassé la début de la fenêtre, soit parce qu'on a une différence avec un **mauvais caractère** du texte.
- Si on a un **mauvais caractère** en position `i+j` dans le texte qui diffère de `motif[j]`, alors on recherche si `texte[i+j]` apparaît dans le motif avant la position `j` :
  - si c'est le cas on prend la position `k` la plus proche de `j` et on décale la fenêtre glissante de `j-k` pour amener ce caractère du motif en face de `texte[i+j]` qui lui est égal ;
  - sinon, on prend -1 et on décale la fenêtre de `j-(-1)` ce qui revient à décaler la fenêtre glissante pour la faire commencer juste après la position `i+j`. C'est correct puisque dans ce cas, aucun caractère du motif avant la position `j` ne correspond à `texte[i+j]`.
- Ainsi on peut améliorer l'algorithme naïf, parce qu'on peut avoir des décalages vers la droite supérieurs à 1. La complexité est difficile à étudier car elle dépend du motif et du texte.

Comme pour la version simplifiée de Horspool, on peut précalculer la table des décalages. Elle est plus compliquée car les décalages doivent être calculés pour chaque caractère de l'alphabet mais aussi pour chaque position dans le motif.
Un outil de visualisation de l'algorithme de Boyer-Moore est proposé par Laurent Abbal :

https://boyer-moore.codekodo.net/recherche_boyer.php



Exemple 2

On donne ci-dessous la trace d'une recherche du motif `"abbbb"` dans le texte `"abaabbababaabbbb"`.
La tableau ci-contre donne pour chacun des caractères `"a"` ou `"b"` la position `k` de son occurrence la plus proche avant la position `j` fixée par la ligne du tableau. S'il n'y a pas d'occurrence on met -1. Avec cette table on peut calculer les décalages nécessaires lors de l'exécution de l'algorithme de Boyer-Moore.

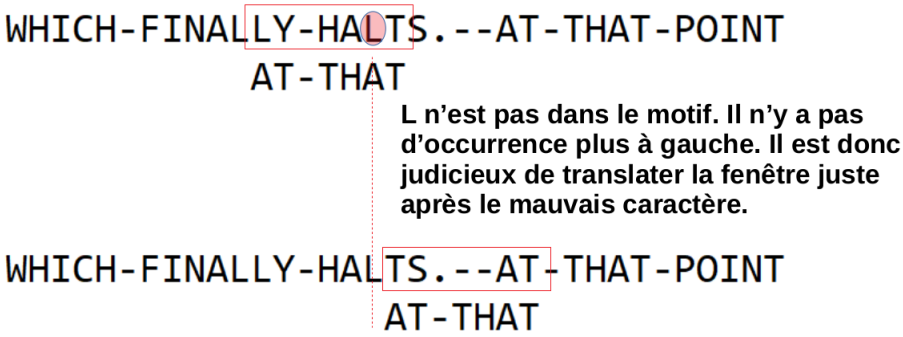
	"a"	"b"
0	-1	-1
1	0	-1
2	0	1
3	0	2
4	0	3

```
Position dans texte : 0
Texte : abaabbababaabbbb
Motif : abbbb
```

```
Position dans texte : 3
Texte : abaabbababaabbbb
Motif :    abbbb

Position dans texte : 6
Texte : abaabbababaabbbb
Motif :       abbbb

Position dans texte : 10
Texte : abaabbababaabbbb
Motif :          abbbb

Position dans texte : 11
Texte : abaabbababaabbbb
Motif :           abbbb
```

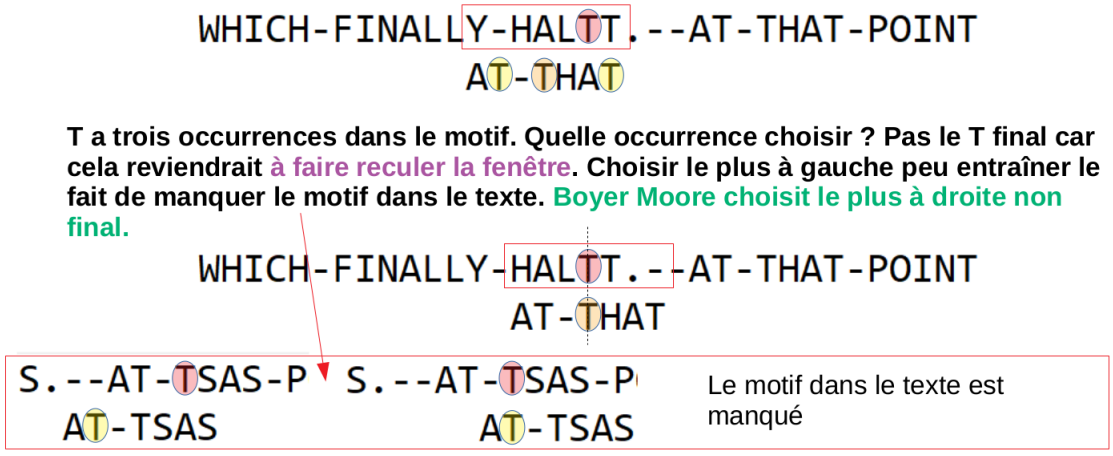
Exemple 3

Source : *Sébastien Aubert.*



Exemple 4

Source : *Sébastien Aubert.*



Exercice 3 *Implémentation de l'algorithme de Boyer-Moore*

1. Compléter un tableau similaire à celui de l'exemple pour le motif `"abracadabra"`.

2. Compléter dans `tp_motif.py` le squelette de la fonction `construire_table` qui prend en paramètres un motif et un alphabet et renvoie une table comme celle de l'exemple sous la forme d'un tableau de dictionnaires.

```
>>> construire_table("abbbb", "ab")
[{'a': -1, 'b': -1},
 {'a': 0, 'b': -1},
 {'a': 0, 'b': 1},
 {'a': 0, 'b': 2},
 {'a': 0, 'b': 3}]
```

3. Compléter dans `tp_motif.py` le squelette de la fonction `premiere_occurrence_boyer_moore` qui prend en paramètres un motif, un texte et un alphabet et renvoie un couple constitué de l'index de la première occurrence de motif dans texte (ou -1 s'il n'est pas trouvé) et du nombre de comparaisons.

```
>>> premiere_occurrence_boyer_moore_trace(
        "abbbb", "abaabbababaabbbb", "ab"
    )
(11, 15)
```

4. Dans la correction `tp_motif_correction.py`, récupérer les codes de `premiere_occurrence_boyer_moore_trace` et `premiere_occurrence_bmh_trace` et comparer leurs déroulements pour la recherche du motif `"abbbb"` dans le texte `"abaabbababaabbbb"`.

5 Recherche de motif, des algorithmes performants

Remarque 2

On peut se demander quel algorithme est implémenté par la méthode `find` de recherche de motif dans une chaîne de caractères Python. Comme la plupart des méthodes des types standards (ou `builtins`), elle est implémentée en C. Python étant *open source*, le code est public : https://github.com/python/cpython/blob/main/Objects/stringlib/fastsearch.h. On y lit :

```
/* fast search/count implementation, based on a mix between boyer-moore and horspool,
with a few more bells and whistles on the top.for some more background, see:
   https://web.archive.org/web/20201107074620/http://effbot.org/zone/stringlib.htm */
```

Une version simplifiée et lisible en Python de l'algorithme est disponible sur la dernière page citée.

Comparaison de différents algorithmes de recherche de motif dans un texte

Cost of searching for an M-character pattern in an N-character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA (Algorithm 5.6)*	$2\,N$	$1.1\,N$	*no*	*yes*	MR
	mismatch transitions only	$3\,N$	$1.1\,N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3\,N$	N/M	*yes*	*yes*	R
	mismatched char heuristic only (Algorithm 5.7)	MN	N/M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo (Algorithm 5.8)*	$7\,N$	$7\,N$	*no*	*yes* †	1
	Las Vegas	$7\,N$ †	$7\,N$	*yes*	*yes*	1

† *probabilisitic guarantee, with uniform hash function*

Source : *Algorithms* de *Robert Sedgewick* et *Kevin Wayne*, disponible au CDI.