Recherche textuelle

Algorithmique

① Position du problème

On dispose d'un **texte** de longueur n et d'un **motif** de longueur m (avec $m \leqslant n$). On cherche à déterminer si le motif apparaît dans le texte, et si oui, à quelles positions.

La recherche d'un motif dans un texte est un problème fondamental en informatique : elle intervient dans les éditeurs de texte (Ctrl+F), les moteurs de recherche, l'analyse de séquences ADN ou encore les outils comme `grep`.

Par exemple, pour `texte = 'abracadabra'` et `motif = 'bra'`, le motif apparaît aux positions 1 et 8. Pour résoudre ce problème, on va « glisser » le motif le long du texte. La question centrale est : *de combien de positions avancer après chaque tentative ?*

② Algorithme naïf (de force brute)

2.1 Principe

L'idée la plus simple consiste à tester chaque position de départ possible : pour chaque indice i de 0 à $n - m$, on compare les caractères du motif un par un avec ceux du texte à partir de la position i. Dès qu'un caractère ne correspond pas, on passe à la position suivante.

On avance toujours d'une seule position, même quand le premier caractère ne correspond pas : c'est ce qui rend cet algorithme lent.

2.2 Implémentation en Python

```python
def algo_naif(texte, motif):
    n = len(texte)
    m = len(motif)
    indices = []
    for i in range(n - m + 1):
        trouvé = True
        for j in range(m):
            if texte[i + j] != motif[j]:
                trouvé = False
                break
        if trouvé:
            indices.append(i)
    return indices
```

Remarques sur le code :

— La borne de la boucle est `n - m + 1` (et non `n - m`) car la position $n - m$ est la dernière position valide pour aligner le motif.

— La variable `trouvé` est initialisée **avant** la boucle intérieure.

— On utilise `break` pour interrompre la comparaison dès le premier caractère qui ne correspond pas.

2.3 Complexité

La boucle extérieure parcourt $n - m + 1$ positions. Pour chacune, la boucle intérieure effectue au plus m comparaisons. La complexité dans le pire des cas est donc :

$$O\left((n - m + 1) \times m\right) \approx O(n \times m)$$

Pour un motif court (m petit devant n), ce coût est proportionnel à n. Il devient **quadratique**, en $O(n^2)$, quand m est de l'ordre de n (par exemple $m = n/2$).

Exemple de pire cas : chercher `'aab'` dans `'aaaaaaaaa'`. À chaque position, les deux premiers caractères correspondent avant d'échouer sur le troisième. On effectue presque $3 \times (n - 2)$ comparaisons.

③ L'algorithme de Boyer-Moore-Horspool

3.1 Deux idées clés

Nous allons étudier une version simplifiée de l'algorithme de Boyer-Moore, proposée par Nigel Horspool en 1980. Cet algorithme repose sur deux idées :

1. On compare le motif **de droite à gauche**, en commençant par sa dernière lettre.
2. Au lieu d'avancer d'une seule position après un échec, on utilise un **décalage** calculé à partir du caractère du texte qui a provoqué l'échec. Ce décalage peut être bien supérieur à 1.

3.2 Déroulement sur un exemple

Cherchons `motif = 'dab'` dans `texte = 'abracadabra'`.

Étape 1 : on aligne le motif en début de texte et on compare le dernier caractère :

```
abracadabra
dab
  ^ on compare ici : 'r' != 'b'
```

Le caractère `'r'` n'apparaît nulle part dans le motif. On décale de toute la longueur du motif, soit 3 positions.

Étape 2 : on aligne le motif à la position 3 :

```
abracadabra
   dab
     ^ 'a' != 'b'
```

Le caractère `'a'` apparaît dans le motif en avant-dernière position. On décale d'une position pour aligner les deux `'a'`.

Étape 3 : position 4 :

```
abracadabra
```

```
  dab
    ^ 'd' != 'b'
```

Le caractère `'d'` apparaît en première position du motif. On décale de 2 positions.

Étape 4 : position 6 :

```
abracadabra
      dab
```

De droite à gauche : `'b'='b'`, `'a'='a'`, `'d'='d'`. Le motif est trouvé à la position 6.

Bilan : pour trouver cette occurrence, on a testé 4 positions au lieu des 7 de l'algorithme naïf (positions 0 à 6). Pour parcourir tout le texte, la fonction du cours teste 5 fenêtres (0, 3, 4, 6 et 7) au lieu de 9.

3.3 La table de décalages (pré-traitement)

Pour chaque caractère c du motif (sauf le dernier), on calcule la distance entre sa dernière occurrence dans le motif (hors dernière position) et la fin du motif. Si un caractère n'est pas dans le motif, le décalage vaut m.

Pour un motif de longueur m, si le caractère c apparaît à la position i (la plus à droite, hors dernière position) :

$$\text{décalage}(c) = m - 1 - i$$

On stocke ces valeurs dans un **dictionnaire**.

Exemple avec `motif = 'dab'` (longueur 3) :

Caractère	Position	Décalage
`'d'`	0	$3 - 1 - 0 = 2$
`'a'`	1	$3 - 1 - 1 = 1$
`'b'`	(exclu)	—
autre	(absent)	3

Le dictionnaire renvoyé est `{'d': 2, 'a': 1}`.

```python
def pre_traitement(motif):
    """Construit la table de décalages
    pour Boyer-Moore-Horspool."""
    m = len(motif)
    décalages = {}
    for i in range(m - 1):
        décalages[motif[i]] = m - 1 - i
    return décalages
```

On exclut le dernier caractère pour éviter un décalage de 0 (boucle infinie).

3.4 Implémentation complète

```python
def recherche_boyer_moore(texte, motif):
    n = len(texte)
    m = len(motif)
    décalages = pre_traitement(motif)
    indices = []

    i = 0
    while i <= n - m:
        j = m - 1                    # droite à gauche
        while j >= 0 and texte[i + j] == motif[j]:
            j -= 1
        if j < 0:                    # tous correspondent
            indices.append(i)
            i += 1
        else:                        # décalage
            c = texte[i + m - 1]
            if c in décalages:
                i += décalages[c]
            else:
                i += m
    return indices
```

Points importants :

— La variable `i` représente la position de début de la fenêtre de comparaison.

— La comparaison se fait de droite à gauche grâce à la boucle `while j >= 0`.

— En cas d'échec, le décalage est déterminé par le caractère du texte aligné avec la **fin** du motif.

— En cas de correspondance complète, on avance de 1 pour trouver les éventuelles occurrences suivantes.

3.5 Complexité

— **Meilleur cas :** si le dernier caractère du motif ne correspond jamais et que le caractère du texte n'est pas dans le motif, on saute de m positions à chaque fois. Le nombre de comparaisons est environ $\frac{n}{m}$, soit une complexité **sous-linéaire** $O\left(\frac{n}{m}\right)$.

— **Pire cas :** $O(n \times m)$, identique à l'algorithme naïf.

— **En pratique :** sur du texte naturel, Boyer-Moore-Horspool est souvent trois à cinq fois plus rapide que l'algorithme naïf.

④ Comparaison expérimentale

On peut comparer les deux algorithmes avec le module `timeit` sur un texte long :

```python
import timeit

texte = open("un_fichier.txt").read()
motif = "algorithme"

t_naif = timeit.timeit(
```

```
    lambda: algo_naif(texte, motif), number=1000)
t_boyer = timeit.timeit(
    lambda: recherche_boyer_moore(texte, motif),
    number=1000)

print(f"Naif    : {t_naif:.4f} s")
print(f"Boyer-Moore : {t_boyer:.4f} s")
```

L'appel `timeit.timeit(f, number=k)` exécute `k` fois la fonction sans paramètre `f` et renvoie la durée totale, en secondes. L'expression `lambda: algo_naif(texte, motif)` fabrique justement cette fonction sans paramètre, qui lance la recherche.

Plus le motif est long et l'alphabet varié, plus l'avantage de Boyer-Moore-Horspool est marqué.