

18.2

Algorithmes naïfs

NSI TERMINALE - JB DUTHOIT

18.2.1 Un premier algorithme naïf

Exercice 18.11

1. Créer une fonction `sous_chaine(texte,i,long)` avec `texte` un str, `i` un indice et `long` un entier non nul. La fonction renvoie une sous-chaine de `texte`, longueur `long` et commençant à l'indice `i` de `texte`. Par exemple, `sous_chaine("bonjour",2,3)` va renvoyer `njo`.
 2. Utiliser la fonction précédente pour réaliser une fonction `recherche(motif,texte)` qui affiche les indices si `motif` est présent dans `texte`.
 3. la fonction `sous_chaine(texte,i,long)` assez couteuse, proportionnelle au nombre de caractères , ici `long`. On considère maintenant `motif = 'b' * 1000` et `texte = 'a' * 2000`. On recherchant `motif` dans `texte`, combien va t-on faire d'opérations (on considère que créer une sous chaîne de M caractère a un coût de M opérations ?
- ☞ La comparaison échoue dès le premier caractère, et on a donc construit les sous-chaînes pour rien !
- ***

18.2.2 Un premier algorithme simple

Principe de la fenêtre glissante

Pour répondre à cela, la première approche consiste à placer notre motif ACG au niveau des premiers caractères de la chaîne. Si le premier caractère du motif ne correspond pas au premier caractère de la chaîne, on décale le motif d'un caractère vers la droite. S'ils correspondent, on fait le test entre les 2ème caractères du motif et de la chaîne. On répète le procédé jusqu'au moment (s'il existe) ou les 3 caractères du motif sont trouvés (ou pas!).

Etape 1 :

CAAGCGCACAAAGACGCCAGACCTTCGTTATAGGCGATGATT
ACG

Etape 2 :

CAAGCGCACAAAGACGCCAGACCTTCGTTATAGGCGATGATT
ACG

Etape 3 :

CAAGCGCACAAAGACGCCAGACCTTCGTTATAGGCGATGATT
ACG

....

Etape 7 :

CAAGCGCACAAAGACGCCAGACCTTCGTTATAGGCGATGATT
ACG

....

Implémentation en Python

Exercice 18.12

1. Construire la fonction suivante sans créer des sous-chaînes de caractères, afin d'être plus efficace :

```
def occurrence(motif, texte, i):
    """
        indique s'il y a une occurrence de la chaîne motif dans
        la chaîne texte à la position i. (renvoie donc True or
        False)
    """
    pass
```

Votre fonction devra notamment passer les tests suivants :

```
assert occurrence('bra', 'abracadabra', 0) == False
assert occurrence('bra', 'abracadabra', 1) == True
assert occurrence('bra', 'abracadabra', -3) == False
assert occurrence('rat', 'abracadabra', 9) == False
assert occurrence('', 'abracadabra', 9) == True
```

2. Construire la fonction suivante en utilisant la fonction précédente :

```
def rechercher(motif, texte):
    """
        Affiche toutes les occurrences de motif dans texte.
    """
    pass
```

Exemple :

```
>>> recherche('bra', 'abracadabra')
1
8
```

Exercice 18.13

- Quelle est la complexité temporelle au mieux de la Recherche Naïve ? Pour quelle forme des données ?
- Quelle est la complexité temporelle au pire de la Recherche Naïve ? Pour quelle forme des données ?
- Supposons que le motif ne contient pas deux fois la même lettre. Écrire un algorithme de recherche exacte de motif qui exploite cette information. Discuter de ses complexités temporelles au mieux et au pire.

18.2.3 En deux mots...

Cet algorithme est très gourmand car il n'a aucune "mémoire" des opérations qui ont été faites précédemment !

On peut l'améliorer considérablement, en particulier en cherchant à décaler le motif de plus d'une case à chaque fois