Algorithme de Knuth - Morris - Pratt

Problème : recherche d'un motif chaine de caractères M dans un texte T et compter ses occurrences On suppose bien sûr que la longueur du motif est inférieur à celle du texte.

Une introduction

Un algorithme naïf consiste à rechercher le premier caractère du motif M dans le texte T puis chercher si le suivant correspond jusqu'à la fin éventuelle du motif et recommencer jusqu'à la fin du texte. En langage naturel, on peut l'écrire ainsi :

$\begin{array}{l} \text{occurence} \leftarrow 0 \\ \text{position} \leftarrow [] \\ p \leftarrow \text{longueur de M} \\ n \leftarrow \text{longueur de T} \\ \text{Pour i allant de 0 à $n - p$ faire} \\ \qquad\begin{array}{|l} j \leftarrow 0 \\ \text{Tant que $j < p$ et $M[j] == T[i + j]$ faire}\\ \qquad\begin{array}{|l} j \leftarrow j + 1 \\ \end{array} \\ \text{Fin Tant que} \\ \text{Si $j == p$ alors}\\ \qquad\begin{array}{|l} \text{position} [\text{occurrence}] \leftarrow i\\ \text{occurence} \leftarrow \text{occurence} + 1 \\ \end{array} \\ \text{Fin Si} \\ \end{array} \\ \text{Fin Pour} \end{array}$

Une implémentation en langage Python peut être la suivante :

In [ ]:
```
def occurencesMotifDansTexte(motif, texte):
    position = []
    p = len(motif)
    n = len(texte)
    for i in range(n-p+1):
        j = 0
        while (j < p) and (motif[j] == texte[i + j]):
            j += 1
        if j == p:
            position.append(i)
    return position
```

In [ ]:
```
texte = "abracadabra et blabla"
motif = "abr"
occurencesMotifDansTexte(motif, texte)
```

Le coût en calcul d'un tel algorithme est de l'ordre du produit des longueurs du motif et du texte. On peut lui reprocher de ne pas prendre en considération l'information acquise lors de chaque recherche du motif pour éviter de repartir à chaque fois du caractère suivant du texte.

Une amélioration naïve de cet algorithme peut conduire à sauter directement après avoir identifier une partie commune entre le motif et le texte.

In [ ]:
```
def occurencesMotifDansTexte_Essai(motif, texte):
    position = []
    p = len(motif)
    n = len(texte)
    ind_texte = 0
    while ind_texte < n - p + 1:
        ind_motif = 0
        while (ind_motif < p) and (motif[ind_motif] == texte[ind_te
xte + ind_motif]):
            ind_motif += 1
        if ind_motif == p:
            position.append(ind_texte)
        ind_texte += ind_motif + 1
    return position
```

In [ ]:
```
texte = "abracadabra et blabla"
motif = "abr"
occurencesMotifDansTexte_Essai(motif, texte)
```

In [ ]:
```
texte = "abababa"
motif = "aba"
occurencesMotifDansTexte(motif, texte)
```

In [ ]:
```
texte = "abababa"
motif = "aba"
occurencesMotifDansTexte_Essai(motif, texte)
```

Ainsi, une approche trop brute conduit à un algorithme incorrect et convaint qu'il est nécessaire d'analyser la composition du motif pour obtenir une version améliorée de l'algorithme naïf qui soit correcte.

L'algorithme de Knuth-Morris-Pratt

L'algorithme de Knuth-Morris-Pratt repose sur une telle démarche.

La résolution du problème 1 a conduit à un algorithme qui, pour une chaîne de caractère donnée, produit le tableau contenant, pour tout nombre entier naturel k compris entre 1 et la longueur de la chaîne, la plus grande valeur de l'entier naturel j tel que
$$ \text{motif}[0:j] = \text{motif}[k - j:k] $$

In [ ]:
```
def tableauPrefixes(motif):
    p = len(motif)              #longueur du motif
    resultat = [0] * p          #tableau des prefixes à 0
    ind_motif = 0               #indice du caractère pivot du motif
    for k in range(1, p):       #on parcourt les caractères 1 à p-1
du motif
        while motif[ind_motif] != motif[k] and ind_motif > 0:
            ind_motif = resultat[ind_motif - 1]   #on teste la chai
ne precedente
        if motif[ind_motif] == motif[k]:          #un caractere rec
onnu on indente
            ind_motif += 1
        resultat[k] = ind_motif
    return resultat
```

In [ ]:
```
tableauPrefixes("baobab")
```

In [ ]:
```
tableauPrefixes("bébé")
```

In [ ]:
```
tableauPrefixes("bonbons")
```

In [ ]:
```
tableauPrefixes("bonhomme")
```

Le coût en instruction de cet algorithme est de l'ordre de la longueur de la chaîne de caractère passée en paramètre.

La mise à disposition du tableau en sortie permet alors d'adapter l'algorithme naïf d'amélioration en obtenant une solution correcte cette fois.

In [ ]:
```
def KMP(motif, texte):
    position = []
    p = len(motif)
    n = len(texte)
    ind_texte = 0
    ind_motif = 0
    prefixe = [0] + tableauPrefixes(motif) #liste à p + 1 éléments
    while ind_texte + ind_motif < n:
        while (ind_motif < p) and (motif[ind_motif] == texte[ind_te
xte + ind_motif]):
            ind_motif += 1
        if ind_motif == p:
            position.append(ind_texte)
        ind_motif = prefixe[ind_motif]
        ind_texte = ind_texte + ind_motif + 1
    return position
```

In [ ]:
```
texte = "abababa"
motif = "aba"
KMP(motif, texte)
```

In [ ]:
```

```