

## Exercice 1 : Chercher un mot

Soit une chaîne de caractères  $d$  de longueur  $n$  et un mot  $t$  de longueur  $m < n$ .

1. Donner l'algorithme naïf retournant la position de la première occurrence du mot  $t$  dans la chaîne  $d$  (ou -1 s'il est absent). Quelle est sa complexité?
2. Donner de même l'algorithme retournant la position de toutes les occurrences du mot  $t$  dans la chaîne  $d$  (ou une liste vide s'il est absent). Quelle est sa complexité?
3. On suppose qu'on peut tester si un caractère  $c$  appartient au motif  $t$  en temps constant. Proposez un algorithme plus efficace que l'algorithme naïf.

on note  $d$  le texte et  $t$  le motif recherché dans le texte

```
Algo : recherche_simple
Données : d, t : chaînes de caractères
n = len (d)
m = len (t)
i <-- 0
tant que i < n - m:
    j <-- i
    tant que j < m et d[i+j] = t[j] :
        j += 1
    si j == m :
        return i
    sinon :
        i <-- i + 1
```

Le deuxième est plus ou moins pareil



```
Algo : recherche_multiple
Données : d, t : chaînes de caractères
n = len (d)
m = len (t)
l = []
i <-- 0
tant que i < n - m:
    j <-- i
    tant que j < m et d[i+j] = t[j] :
        j += 1
    si j == m :
        l.append(i)
    i <-- i + 1
```

Cette approche a un inconvénient : après une comparaison infructueuse, la comparaison suivante débutera à la position  $i + 1$ , sans tenir aucun compte de celles qui ont déjà eu lieu à l'itération précédente, à la position  $i$ .

L'algorithme de Knuth-Morris-Pratt examine d'abord la chaîne  $t$  et en déduit des informations permettant de ne pas comparer chaque caractère plus d'une fois.

On regardera une version simplifiée de l'algo de Knuth-Morris-Pratt :

- On suppose qu'on peut tester si un caractère  $c$  appartient au motif  $t$  en temps constant
- Le but est de calculer un décalage permettant de ne pas inspecter les positions où il n'y a aucune chance de trouver le motif  $t$ .
- On commence par chercher la position  $i = m - 1$
- On inspecte la chaîne  $d$  de  $i$  à  $i - m$  décroissant, et on s'arrête au premier caractère commun (i.e.  $j \leftarrow i$  t.q.  $d[i-j]$  appartient à  $t$ )
- Soit  $c = d[i-j]$  le caractère commun
- On note  $k$  la position de la première occurrence de  $c$  dans  $t[:m-j-1]$
- Le décalage est égal à  $m - 1 - j - k$
- Si on ne trouve rien, le décalage vaut  $m$

A illustrer au tableau : 

Si vous en avez le courage, voici le code :



Algo : recherche améliorée

Données :  $d, t$  : chaînes de caractères

```
n = len (d)
m = len (t)
i <-- m - 1
tant que i < n :
    # PRE-TRAITEMENT
    j <-- 0
    tant que j < m - 1:
        c = d[i - j]
        si c appartient à t[:m-j-1]
            k <-- dernière_occurrence(c, t[:m-j-1])
            decalage <-- m - 1 - j - k
            break
        sinon
            j += 1
    si j == m - 1:
        decalage <-- m
    # TRAITEMENT
    j <-- 0
    tant que j < m :
        si t[m - j - 1] = d[i - j]
            j += 1
        sinon
            break
    si j = m:
        retourner i - m + 1
    # DECALAGE
    i <-- i + decalage
retourner -1
```

## Exercice 2 : Compter les mots

1. Écrire un algorithme qui compte le nombre de mots dans un texte.

Remarque : On considère comme caractère d'espacement tout caractère qui n'est pas alphanumérique (alphabétique accentué ou non et chiffres).

2. Dessiner l'automate fini correspondant.

Algorithme à expliquer avec un petit automate fini à deux états



```
def compte_mots(d):  
    state = 0  
    cpt = 0  
    for i in range(len(d)):  
        if state == 0 and is_alpha(d[i]):  
            state = 1  
            cpt += 1  
        if state == 1 and is_sep(d[i]):  
            state = 0  
    return cpt
```

## Exercice 3 : Palindrome

1. Écrire un algorithme récursif permettant de savoir si un tableau de caractères est un palindrome (un palindrome se lit "à l'endroit" et "à l'envers" de la même façon, comme par exemple "à l'étape, épate-la!").

Remarque : on ne considère ni la ponctuation, ni les espaces, ni les accents.

Quelle est sa complexité?



**C'est un algo qu'on a déjà vu**

2. Pouvez-vous définir un automate fini capable de reconnaître les palindromes?



L'intérêt de cet exo est de montrer que certains motifs ne sont pas reconnaissables par les automates finis. Ici dans le cas du palindrome, il faudrait un automate qui accepte a, b, aa, bb, aaa, aba, bbb, bab, aaaa, abba, etc... l'automate existe si on limite la taille max des palindromes mais pas dans le cas général. Par ailleurs, le nb d'états est exponentiel.

La solution consisterait à utiliser un automate à pile (non vu en cours). On a alors deux



états principaux : un état correspondant à l'empilage des symboles d'entrée, et un état correspondant au dépileage de la pile et à la comparaison avec les symboles d'entrée. La comparaison n'est acceptée que si les symboles sont identiques, et le mot est reconnu lorsque la pile est vide.

## Exercice 4 : Expressions régulières

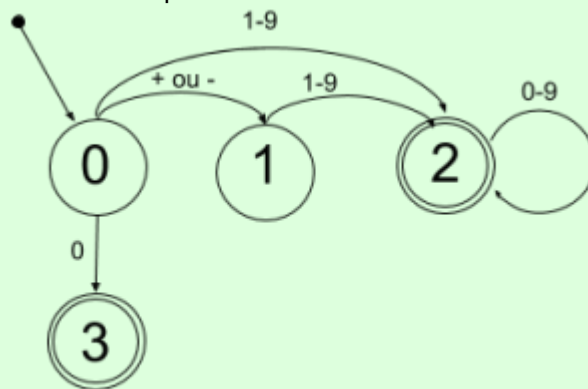
- Les langages réguliers sont des types de langages formels qui peuvent être reconnus par un automate fini.
- Le langage des expressions régulières est un langage régulier qui permet de décrire des motifs (c'est à dire des classes de mots) dans une chaîne de caractère.

**1. Donnez l'expression régulière permettant de reconnaître les entiers relatifs et dessiner l'automate fini correspondant.**

Réponse :  $([+-]?[1-9][0-9]^*|0)$

- le chiffre commence par +, - ou rien du tout
- il n'y a pas de 0 au début de la partie entière
- il n'y a pas de caractère entre, seulement des chiffres au milieu.

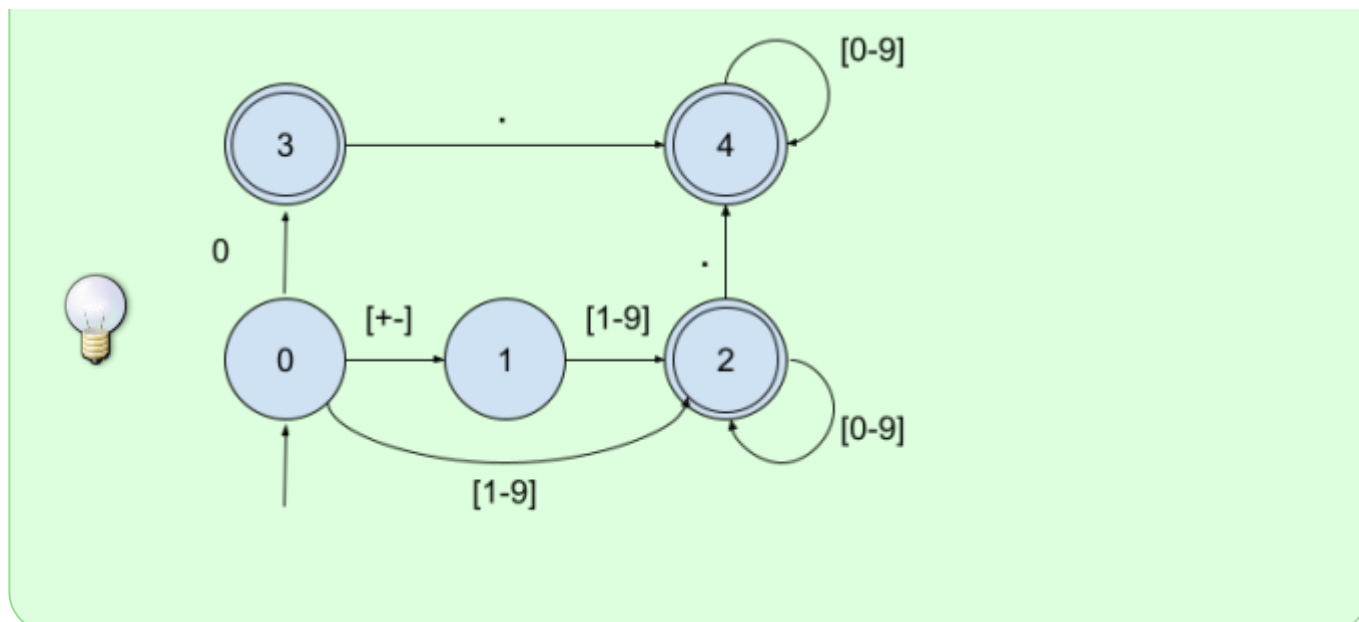
Automate qui reconnaît les nombres entiers:



**2. Donnez l'expression régulière permettant de reconnaître les nombres décimaux (par exemple -3, 12.3, -12.34, +3, 0.) et dessiner l'automate fini correspondant.**

Réponse :  $([+-]?[1-9][0-9]^*|0)\.[0-9]^*$





3. Donnez l'expression régulière expression régulière qui valide les noms de fichiers se terminant par l'une des extensions spécifiées : .jpg, .png, ou .gif et dessiner l'automate fini correspondant.



TODO

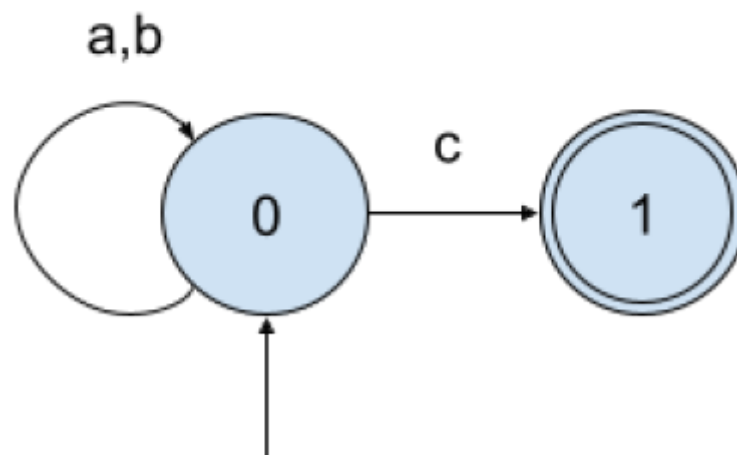
## Exercice 5

1. Ecrire l'algorithme de reconnaissance de l'expression régulière  $(a|b)^*c$

```
def reconnait(s):
    final = False
    for c in s:
        if final:
            final = False
            break
        if (c not in 'abc'):
            break
        if c=='c':
            final = True
    return final
```

2. Dessiner son automate fini

## Déterministe:



3. Soit  $G$  le graphe orienté décrivant cet automate, chaque arête étant indexée par un caractère. Donner l'algorithme qui indique si oui ou non l'expression est reconnue dans une chaîne  $s$  à partir de son automate fini.

```
G = { 0 : {'a':0, 'b':0, 'c':1}, 1:{}}

def reconnait_auto(s, G):
    state = 0
    for c in s:
        if c in G[state]:
            state = G[state][c]
        else:
            return False
    if state == 1:
        return True
    else:
        return False
```

4. Quelle est sa complexité?

Longueur de la chaîne

## Exercice 6

1. Même exercice avec l'expression régulière  $(a|b)^*ab$

```
def reconnait_nd(s):
    transit_a = False
    final_b = False
    for c in s:
```

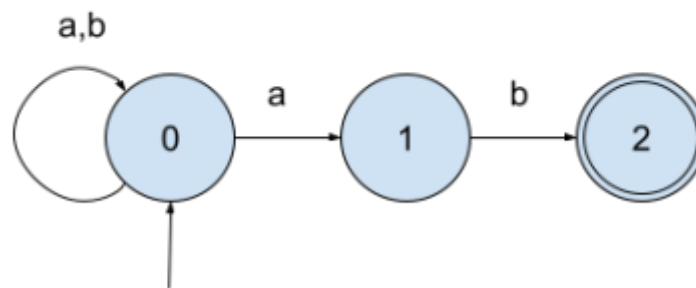
```

if (c not in 'ab'):
    break
if c=='b':
    if transit_a:
        transit_a = False
        final_b = True
    else:
        final_b = False
if c == 'a':
    final_b = False
    transit_a = True
return final_b

```

## 2. Dessiner son automate fini

Non déterministe:



Voir le graphe G ci-dessous **Attention, graphe non déterministe!!**

```

G = { 0 : {'a':(0,1), 'b':(0,)}, 1:{'b':(2,)}, 2:{}}

```

3. Soit G le graphe orienté décrivant cet automate, chaque arête étant indexée par un caractère. Donner l'algorithme qui indique si oui ou non l'expression est reconnue dans une chaîne s à partir de son automate fini.

```

def reconnait_auto_nd(s, G):
    states = set((0,))
    for c in s:
        previous_states = states
        states = set()
        for state in previous_states:
            if c in G[state]:
                next_states = G[state][c]
                for state in next_states:
                    states.add(state)
        print(c, states)
    if len(states) == 0 :
        return False
    if 2 in states:
        return True

```

```
else:  
    return False
```

#### 4. Complexité



$O(n \times l)$  avec  $n$  le nb d'états et  $l$  la longueur de la chaîne

From:  
<https://wiki.centrale-med.fr/informatique/> - WiKi informatique

Permanent link:  
[https://wiki.centrale-med.fr/informatique/tc\\_info:2023-td-texte-corr?rev=1701258669](https://wiki.centrale-med.fr/informatique/tc_info:2023-td-texte-corr?rev=1701258669)

Last update: **2023/11/29 12:51**

