DIU Enseigner l'informatique au lycée

Notes de cours

Jean-Christophe Filliâtre

Jean-Christophe.Filliatre@lri.fr

ressources, pour les projets notamment

beaucoup de données librement accessibles

- projet Gutenberg
  57 000 livres sur `https://www.gutenberg.org/`
- Wikipedia
  `https://dumps.wikimedia.org/`
- OpenStreetMap
  `https://openstreetmap.fr/`

Recherche textuelle

le problème

On cherche les occurrences d'une chaîne de caractères, appelée le motif, dans une autre chaîne de caractères, appelée le texte.

Exemple : il y a deux occurrences du motif `"bra"` dans le texte `"abracadabra"`, aux positions 1 et 8.

objectif

On cherche à écrire une fonction Python

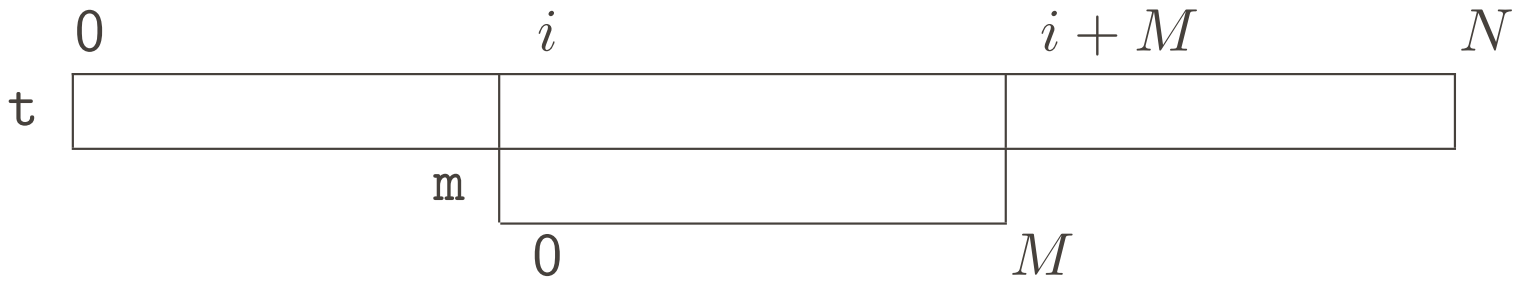
```
def recherche(m, t):
    ...
```

qui affiche toutes les occurrences du motif `m` dans le texte `t` sous la forme suivante :

```
occurrence à la position 1
occurrence à la position 8
```

visualisation

Il est utile de bien visualiser une occurrence de `m` de longueur M dans `t` de longueur N, à la position i, comme ceci :



En particulier,

$$0 \leq i \leq N - M$$

Exemple : $N = 11$, $M = 3$, $i = 1$

```
  1     4           11
a b r a c a d a b r a
  b r a
  0     3
```

rappels

- une chaîne de caractères `s` est immuable (ce n'est pas un tableau)
- sa longueur s'obtient avec **len**`(s)`
- son `i`-ième caractère s'obtient avec `s[i]`,
  pour $0 \leq$ `i` $<$ `len(s)`
- `s[i:j]` est la sous-chaîne de `s` contenant les caractères `i` inclus à `j` exclu
- les chaînes de caractères peuvent être comparées avec l'opérateur ==

un algorithme simple

```
def recherche(m, t):
    for i in range(0, len(t) - len(m) + 1):
        if t[i : i + len(m)] == m:
            print("occurrence à la position", i)
```

Le coût total est $M(N - M + 1)$

(car chaque extraction de la sous-chaîne coûte M).

Exemple : on cherche un motif qui contient $M = 1000$ caractères a dans un texte qui contient $N = 2000$ caractères b.

$\Rightarrow$ plus d'un million d'opérations !

un peux mieux

On peut se passer de `t[i:i+len(m)]`.

```
def occurrence(m, t, i):
    """indique s'il y a une occurrence de la chaîne m
       dans la chaîne t à la position i"""
    if i < 0 or i > len(t) - len(m):
        return False
    for j in range(len(m)):
        if t[i + j] != m[j]:
            return False
    return True
```

un peux mieux

La recherche s'en déduit :

```
def recherche(m, t):
    """affiche toutes les occurrences de m dans t"""
    for i in range(0, len(t) - len(m) + 1):
        if occurrence(m, t, i):
            print("occurrence à la position", i)
```

Sur le même exemple, un millier d'opérations au lieu d'un million.

cependant

On reste sur une complexité $M(N - M + 1)$ dans le pire des cas, par exemple

- si on cherche `"a"*1000` dans `"a"*2000` (1001 occurrences),
- si on cherche `"a"*1000+"b"` dans `"a"*2000` (aucune occurrence)

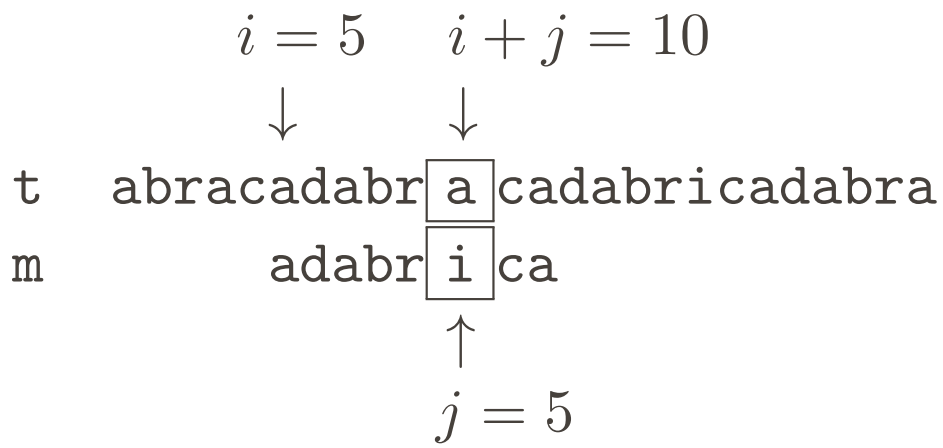
accélération de la recherche

On va essayer d'accélérer la recherche, en avançant plus vite dans le texte, c'est-à-dire en ne testant pas toutes les positions `i` possibles.

Pour cela, il nous faut un critère correct (sans quoi on risquerait de rater des occurrences).

une idée sur un exemple

On est en train de tester la position $i = 5$ et elle échoue sur le caractère $j = 5$.



Alors

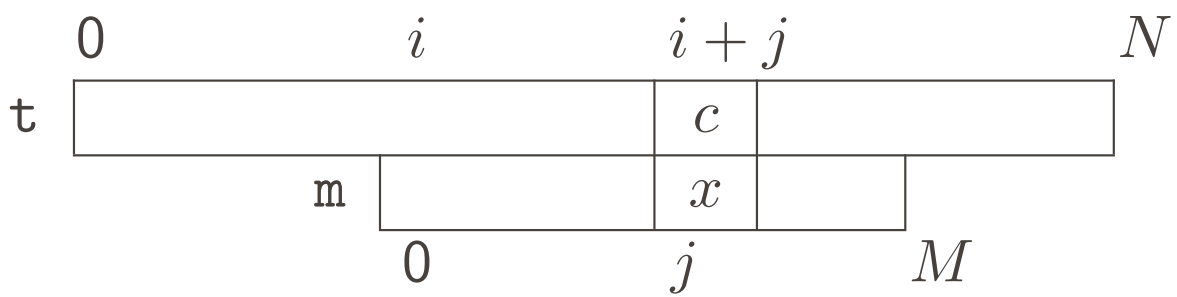
- il ne sert à rien de tester $i = 6$ ensuite, car cela amènerait un `r` sous le `a` ;
- il ne sert à rien de tester $i = 7$ non plus, car cela amènerait un `b` sous le `a`.

Du coup, on peut passer directement à $i = 8$.

l'algorithmique de Boyer-Moore

Son principe est le suivant

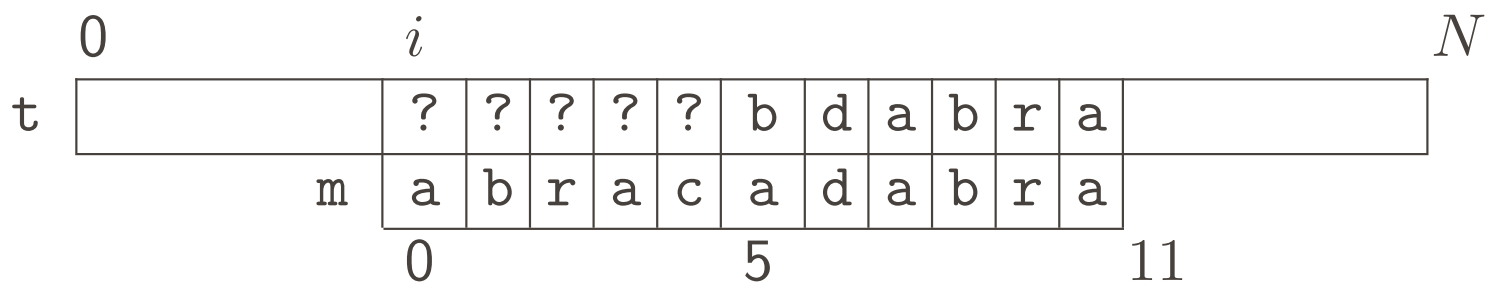
- On teste les occurrences de `m` dans `t` à des positions i de plus en plus grandes, en partant de $i = 0$.
- Pour une position i donnée, on compare les caractères de `m` et de `t` de la droite vers la gauche.
  - Si tous les caractères coïncident, on a trouvé un occurrence. On la signale et on incrémente i.
  - Sinon, on consulte une table de décalages, indexée par l'indice j de la première différence et le caractère `t`$[i + j]$ du texte. Elle nous indique de combien augmenter i.



l'algorithme de Boyer-Moore

On peut précalculer la table de décalages à l'avance, sur la base du motif uniquement (i.e. quel que soit le texte dans lequel on cherchera).

Exemple :



Pour $j = 5$ et le caractère b, la table indiquera un décalage de 4.

représenter la table de décalages

Elle a deux clés :

- l'indice j du caractère du motif qui diffère ;
- le caractère c du texte.

Elle donne l'indice du caractère c le plus à droite dans le motif avant l'indice j.

On utilise un tableau (ici de taille 11) contenant des dictionnaires.

Exemple : pour $j = 5$ et $c =$ `b`, la table donne 1 et on décale donc de $j - 1 = 5 - 1 = 4$.

`abracadabra`

	a	b	r	c	d
0					
1	0				
2	0	1			
3	0	1	2		
4	3	1	2		
5	3	1	2	4	
6	5	1	2	4	
7	5	1	2	4	6
8	7	1	2	4	6
9	7	8	2	4	6
10	7	8	9	4	6

utilisation de la table

```
def recherche(m, t):
    d = table_bm(m)
    i = 0
    while i <= len(t) - len(m):
        k = 0
        for j in range(len(m) - 1, -1, -1):
            if t[i + j] != m[j]:
                k = decalage(d, j, t[i + j])
                break
        if k == 0:
            print("occurrence à la position", i)
            k = 1
        i += k
```

calcul du décalage

```
def decalage(d, j, c):
    if c in d[j]:
        # c apparaît en d[j][c]
        # et on décale de la différence
        return j - d[j][c]
    else:
        # c n'apparaît pas du tout dans m[0..j-1]
        return j + 1
```

construction de la table

Par force brute :

```
def table_bm(m):
    """d[j][c] est le plus grand k < j tel
       que m[k] == c, s'il existe"""
    d = [{} for _ in range(len(m))]
    for j in range(len(m)):
        for k in range(j):
            d[j][m[k]] = k
    return d
```

efficacité

Dans le pire des cas, la comparaison entre le motif et le texte se fait systématiquement jusqu'au bout du motif.

Exemple : on cherche `abbb...bb` dans `bbbb...bb` (aucune occurrence) ou `bbb...bb` dans `bbbb...bb` (une occurrence à chaque position).

C'est en $M(N - M + 1)$ (pas meilleur que la recherche simple).

efficacité

Dans le meilleur des cas, en revanche, la comparaison peut être négative immédiatement, dès le premier caractère testé, c'est-à-dire pour $j = M - 1$, et le décalage être aussi grand que M.

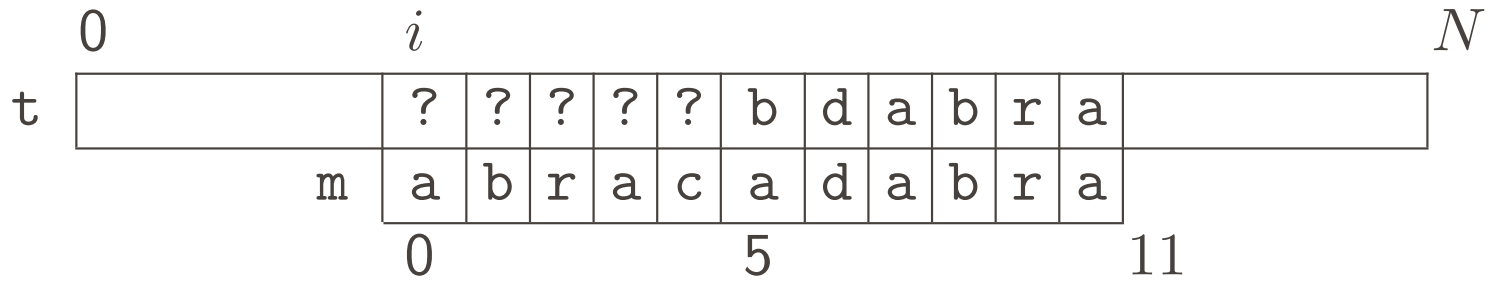
Exemple : on cherche `aaa...aa` dans `bbb...bb`.

Le nombre total de comparaisons sera alors N/M,

car on ne compare plus qu'un caractère sur M.

plus efficace encore

Toujours sur le même exemple,



on aurait pu décaler de 7 au lieu de 4.

Il est en effet inutile d'amener `racad` sous `dabra`.

De tels décalages plus subtils peuvent être également précalculés.

(Mais cela dépasse largement le programme de terminale.)

Programmation dynamique

point de départ

Quand on propose une solution récursive à un problème, on est parfois amené à calculer plusieurs fois la même chose.

exemple 1 : Fibonnaci

La très célèbre suite de Fibonnaci

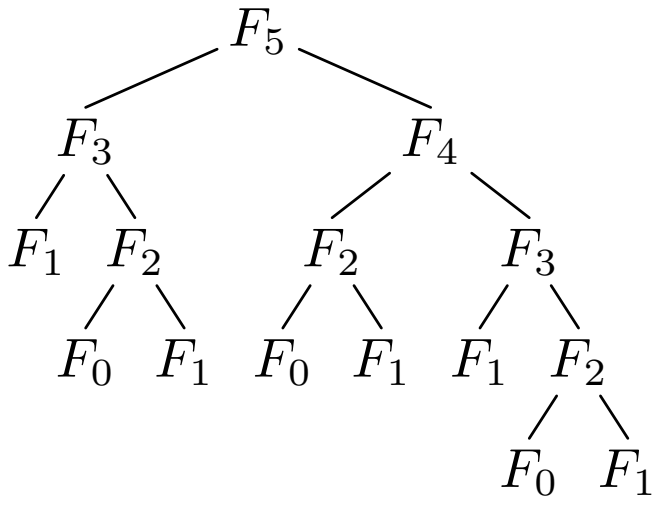
$$F_n = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ F_{n-2} + F_{n-1} & \text{si } n > 1. \end{cases}$$

s'écrit facilement en Python comme une fonction récursive :

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

problème

On calcule plein de fois la même chose.



Le temps de calcul de F_n est exponentiel en n.

(Par exemple, `fib(100)` ne termine pas en un temps raisonnable.)

solution

On calcule les valeurs de F_0, F_1, ..., F_n dans un tableau.

```
def fib(n):
    f = [0] * (n+1)
    f[1] = 1
    for i in range(2, n + 1):
        f[i] = f[i - 2] + f[i - 1]
    return f[n]
```

Maintenant, le calcul de `fib(100)` est instantané

(99 tours de boucle).

exemple 2 : rendu de monnaie

Étant donné un système monétaire (par exemple, des pièces de 1, 2 et 5 euros), quel est le nombre minimal de pièces nécessaires pour obtenir une somme s donnée ?

exemple 2 : rendu de monnaie

On se donne le système monétaire sous la forme d'un tableau ; par exemple

```
pieces = [1, 2, 5]
```

On propose une solution récursive :

```
def rendu_monnaie(pieces, s):
    if s == 0:
        return 0
    r = s   # s = 1+1+...+1 dans le pire des cas
    for p in pieces:
        if p <= s:
            r = min(r, 1+rendu_monnaie(pieces, s-p))
    return r
```

calcul exponentiel

Comme avec Fibonnaci, le calcul prend rapidement trop de temps.

Par exemple, le calcul de

```
rendu_monnaie([1, 2], 100)
```

ne termine pas dans un temps raisonnable.

En effet,

- l'appel avec `s`=100 provoque deux appels avec `s`=99 et `s`=98
- l'appel avec `s`=99 provoque deux appels avec `s`=98 et `s`=97
- etc.

On peut montrer que le nombre d'appels à `rendu_monnaie` est exactement $F_{103} - 1$ (qui dépasse $1,5 \times 10^{21}$).

avec de la programmation dynamique

```
def rendu_monnaie(pieces, s):
    nb = [0] * (s + 1)
    for n in range(1, s + 1):
        nb[n] = n # n=1+1+...+1 dans le pire des cas
        for p in pieces:
            if p <= n:
                nb[n] = min(nb[n], 1 + nb[n - p])
    return nb[s]
```

Le calcul de `rendu_monnaie([1, 2], 100)` est maintenant instantané. (Un tableau de taille 101 et quelques centaines de calculs.)

remarque

Pour construire la solution, il faudrait se fatiguer un peu plus, en la stockant par exemple dans un second tableau.

résumé

La programmation dynamique est une technique pour améliorer l'efficacité d'un algorithme en évitant les calculs redondants.

Pour cela, on stocke dans un tableau les résultats intermédiaires du calcul, afin de les réutiliser au lieu de les recalculer.

Diviser pour régner

principe

Le principe « **diviser pour régner** » consiste à

1. décomposer un problème en un ou plusieurs **sous-problèmes** de même nature, mais plus petits ;
2. résoudre ces sous-problèmes, éventuellement en les décomposant à leur tour **récursivement** en problèmes plus petits encore ;
3. déduire des solutions des sous-problèmes la solution du problème initial.

exemple 1 : la recherche dichotomique

Principe : on cherche une valeur `v` dans un tableau trié, en divisant par deux le segment de recherche `g..d` à chaque étape.

Il y a ici un seul sous-problème (le demi-segment sur lequel on continue).

la recherche dichotomique

```
def recherche(t, v, g, d):
    """renvoie une position de v dans t[g..d],
       supposé trié, et None si elle ne s'y trouve pas"""
    if g > d:
        return None
    m = (g + d) // 2
    if t[m] < v:
        return recherche(t, v, m + 1, d)
    elif t[m] > v:
        return recherche(t, v, g, m - 1)
    else:
        return m
```

```
def recherche_dichotomique(t, v):
    """renvoie une position de v dans le tableau t,
       supposé trié, et None si elle ne s'y trouve pas"""
    return recherche(t, v, 0, len(t) - 1)
```

propriétés

terminaison la taille du segment `g..d` diminue strictement à chaque appel récursif

correction la valeur `v` n'apparaît pas en dehors du segment `g..d`

efficacité le nombre d'appels récursifs, et donc le temps de calcul, est logarithmique en la taille du tableau (en particulier, pas de risque de `RecursionError`)

exemple 2 : le tri fusion

Pour trier une liste, on peut

1. la découper en deux listes de même taille (à 1 près) ;
2. trier ces deux listes récursivement ;
3. puis enfin les fusionner.

le tri fusion 1/3

On suppose des listes construites avec cette classe (cf mardi) :

```
class Cellule:
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

Commençons par l'essentiel :

```
def tri_fusion(lst):
    if lst is None or lst.suivante is None:
        return lst
    l1, l2 = coupe(lst)
    return fusion(tri_fusion(l1), tri_fusion(l2))
```

le tri fusion 2/3

Il reste à faire le découpage :

```
def coupe(lst):
    """sépare une liste en deux listes ayant le même
       nombre d'éléménts, à un près"""
    l1, l2 = None, None
    while lst is not None:
        l1, l2 = Cellule(lst.valeur, l2), l1
        lst = lst.suivante
    return l1, l2
```

le tri fusion 3/3

et la fusion :

```
def fusion(l1, l2):
    """fusionne deux listes triées"""
    if l1 is None:
        return l2
    if l2 is None:
        return l1
    if l1.valeur <= l2.valeur:
        return Cellule(l1.valeur,
                       fusion(l1.suivante, l2))
    else:
        return Cellule(l2.valeur,
                       fusion(l1, l2.suivante))
```

(peut provoquer `RecursionError`)

efficacité

Le tri fusion a une complexité proportionnelle à $N \log_2 N$, où N est la longueur de la liste à trier.

C'est optimal pour un tri

- qui procède uniquement par comparaisons de deux éléments ;
- qui ne connaît rien quant aux valeurs à trier, notamment leur distribution.