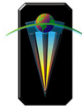
CSI2510
Structures de données et algorithmes

Recherche des motifs



Chaînes de caractères (string)

- Exemples de chaînes (ou séquences) de caractères:
  - ✓ Programme Java
  - ✓ HTML document
  - ✓ séquence DNA
  - ✓ Image numérique
- Un alphabet S est l'ensemble des caractères utilisés dans les mots d'un texte; Exemple d' alphabets:
  - ✓ ASCII
  - ✓ Unicode
  - ✓ {0, 1}
  - ✓ {A, C, G, T}

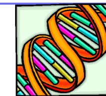
Recherche de motif

- Soit P une séquences de caractères de taille m
  - ✓ Une sous- séquence P[i .. j] de P est composée par les caractères entre i et j
  - ✓ Un préfixe de P est une sous- séquence de type P[0 .. i]
  - ✓ Un suffixe de P est une sous- séquence de type P[i ..m - 1]
- Étant donné deux séquences T (texte) et P (motif), le problème de recherche du motif est de trouver une sous-séquences de T égale à P
- Applications:
  - ✓ Éditeurs de texte
  - ✓ Moteurs de recherche
  - ✓ Recherche biologique

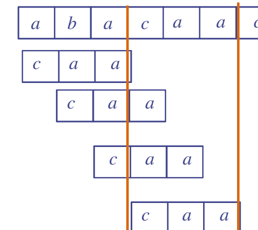
a	b	a	c	a	a	c

c	a	a

Recherche de motif : 'Brute force'



- L'algorithme de recherche de motif "brute force" compare le motif P avec le texte T pour chaque décalage de P relatif à T, jusqu'à ce que:
  - ✓ Une correspondance est trouvée ou
  - ✓ on a essayé toutes les possibilités



Algorithme exhaustif *Force brute*

```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while ( j < m && T[i + j] = P[j])
            j ← j + 1
        if j = m
            return i {match at i}
    return -1 {no match anywhere}
```

Temps d'execution: $O(nm)$

Exemples du pire cas:

- ✓ $T = aaa \ldots ab$ et $P = aaab$
- ✓ Il pourrait se présenter en images et en séquences ADN
- ✓ Rare dans les langues naturelles

Algorithme de Boyer-Moore

- Alphabet de taille moyenne et motif long
- L'algorithme de recherche des motifs de Boyer-Moore

est basé sur deux heuristiques:

Heuristique Looking-glass:

Comparer P avec une sous-séquence de T en commençant par la fin de P (jusqu'au premier caractère)

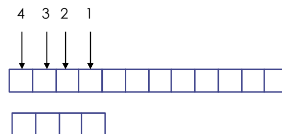
Heuristique character-jump:

Quand un "mismatch" arrive àT[i] = c (avec P[j])

- Si P contient c, décaler P pour aligner la dernière occurrence (la plus à droite) de c en P avec T[i]
- Si P ne contient pas c, décaler P complètement après T[i]

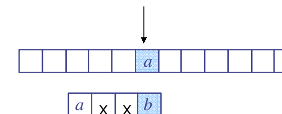
Algorithme de Boyer-Moore

Heuristique looking-glass:



Algorithme de Boyer-Moore

Heuristique character-jump:

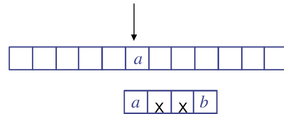


Si P contient c, décaler P pour aligner la dernière occurrence (la plus à droite) de c en P avec T[i]

Algorithme de Boyer-Moore

Heuristique character-jump:

Si P contient c, shift P pour aligner la dernière occurrence (la plus à droite) de c en P avec T[i]

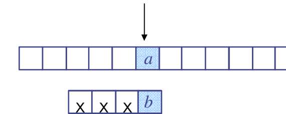


Si c est à la fin de P (parmi les caractères déjà vérifiés) se déplacer d'une seule unité (voir exemple plus tard)

Algorithme de Boyer-Moore

Heuristique character-jump:

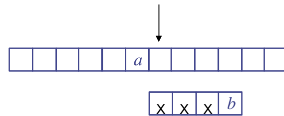
Si P ne contient pas c, décaler P pour aligner P[0] avec T[i + 1]



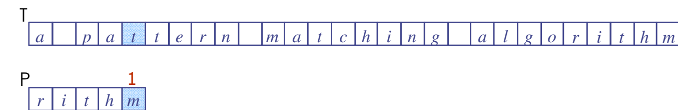
Algorithme de Boyer-Moore

Heuristique character-jump :

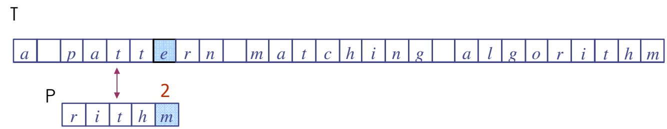
Si P ne contient pas c, décaler P pour aligner P[0] avec T[i + 1]



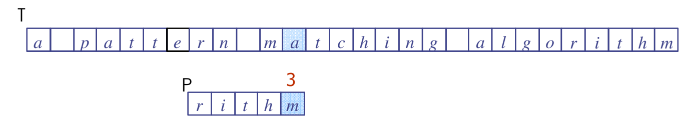
Algorithme de Boyer-Moore



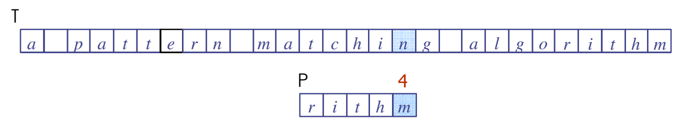
Algorithme de Boyer-Moore



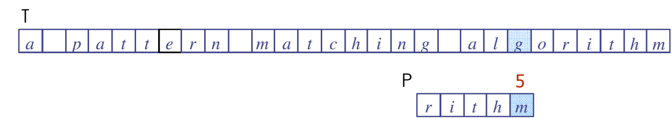
Algorithme de Boyer-Moore



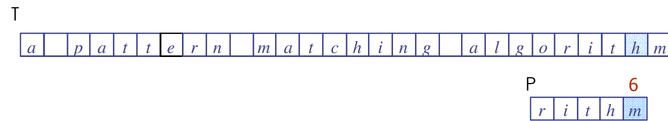
Algorithme de Boyer-Moore



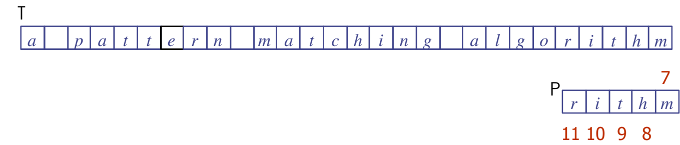
Algorithme de Boyer-Moore



Algorithme de Boyer-Moore



Algorithme de Boyer-Moore



Algorithme de Boyer-Moore

Algorithm *BMMatch(T, P)*

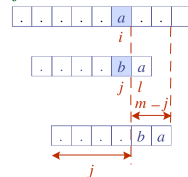
Input: Text T with n characters, pattern P with m characters

Output: Starting index first substring of T matching P, or indication there is no match

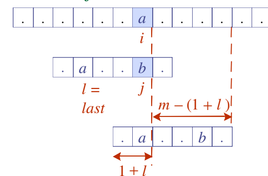
// last(c)=-1 si P ne contient pas c

$i \leftarrow m - 1$
$j \leftarrow m - 1$
repeat
  if $T[i] = P[j]$
    if $j = 0$
      return i {a match ! }
    else
      $i \leftarrow i - 1$
      $j \leftarrow j - 1$
  else {mismatch => a jump !}
    $i \leftarrow i + m - \min(j, 1 + \text{last}(T[i]))$
    $j \leftarrow m - 1$
until $i > n - 1$
return "There is no match"

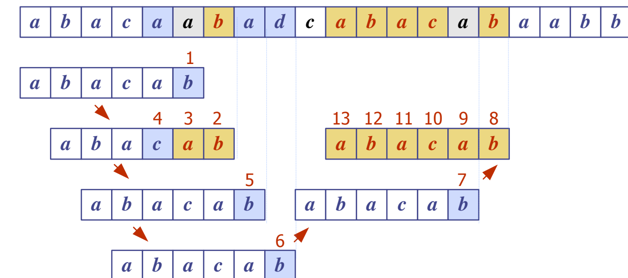
Cas 1: $j \le 1 + l$



Cas 2: $1 + l \le j$

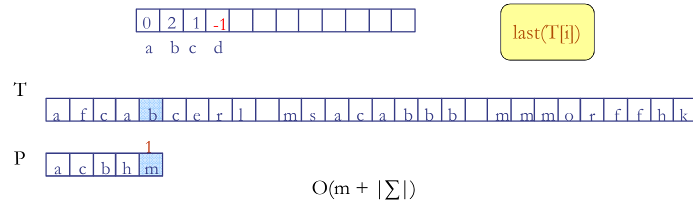


Algorithme de Boyer-Moore



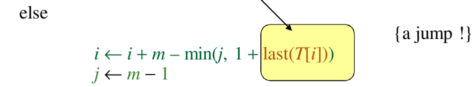
Algorithme de Boyer-Moore: Complexité

Garder un tableau de taille $|\sum|$ (alphabet). Pour chaque entrée mémoriser la dernière occurrence en P de la lettre correspondante.



Algorithme de Boyer-Moore: Complexité

Construire cette fonction a coûté
O(m + |∑|)



La partie de recherche: O(mn) au pire des cas comme brute-force

Algorithme de Boyer-Moore: Complexité

Le temps d'exec. de l'algorithme Boyer-Moore est O(nm + |∑|) au pires des cas

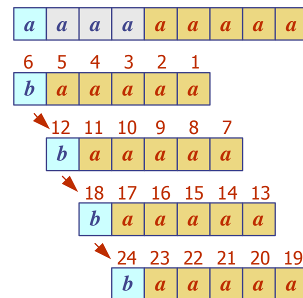
Exemple du pire cas :

$T = aaa \ldots a$

$P = baaa$

Le pire de cas peut arriver dans les images et les séquences ADN; il est rare dans le texte;

Boyer- Moore beaucoup plus vite que l'algorithme 'brute force' pour les textes



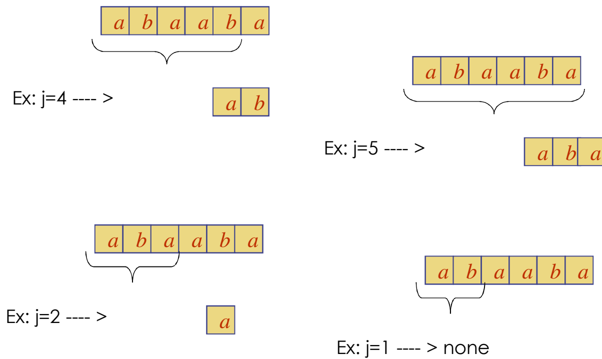
Algorithme KMP

- L'algorithme Knuth-Morris-Pratt compare le motif avec le texte de-gauche-a-droite, mais le décalage est choisi de façon plus intelligente que dans l'algorithme 'brute force'
- Quand il y a un "mismatch" quel est le décalage maximal que l'on puisse faire pour éviter les comparaisons redondantes ?
- Réponse:

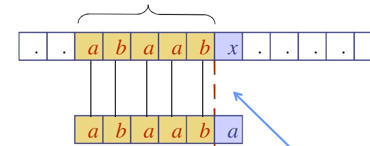
Aligner le plus long préfixe de $P[0..j]$ qui est un suffixe de $P[1..j]$

Algorithme KMP

Le plus long préfixe de P[0..j] qui est un suffixe de P[1..j]



Algorithme KMP



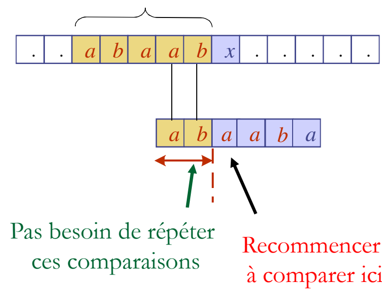
Je dois trouver le préfixe (aussi long que possible)

qui s'apparie avec un suffixe de la chaine en haut

mais ces deux chaines sont identiques!

Algorithme KMP

Décalage tel que le préfixe coïncide avec le suffixe

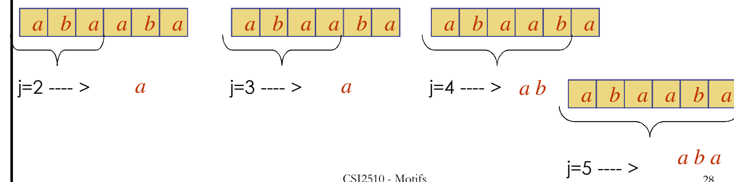


Algorithme KMP

✓ L'algorithme Knuth-Morris-Pratt calcule en avance les préfixes du motif qui coïncident avec ses suffixes

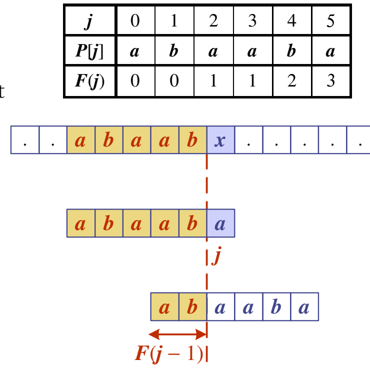
j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*a*	*b*	*a*
F(*j*)	0	0	1	1	2	3

✓ La "failure function" *F*(*j*) est définie comme la taille du plus long préfixe de *P*[0..*j*] qui est aussi un suffixe de *P*[1..*j*]



Algorithme KMP

L'algorithme Knuth-Morris-Pratt modifie l'algorithme de 'brute force' de manière que si le "mismatch" arrive à (P[j] ≠ T[i]) on échange j par F(j - 1) et i (la position dans T) reste la même (ou si j =0, i est incrémentée de 1)



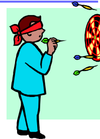
j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

Algorithme KMP

- La "failure function" peut être représentée par un tableau qui peut être calculé en un temps **O(m)**
- A chaque itération de la boucle while :
  - ✓ i est incrémenté de un, OU
  - ✓ La quantité de décalage **i - j** est incrémentée de au moins un (observez que **F(j - 1) < j**)
- Donc, il n'y a pas plus que **2n** itérations de la boucle while
- Donc, l'algorithme KMP a un temps optimal de **O(m + n)**

```
Algorithm KMPMatch(T, P)
  F ← failureFunction(P)
  i ← 0
  j ← 0
  while i < n
    if T[i] = P[j]
      if j = m − 1
        return i − j { match }
      else
        i ← i + 1
        j ← j + 1
    else
      if j > 0
        j ← F[j − 1]
      else
        i ← i + 1
  return −1 { no match }
```

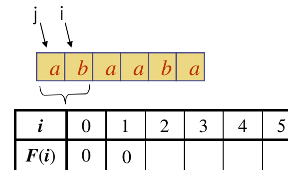

Algorithme KMP



- La "failure function" peut être représentée par un tableau qui peut être calculé en temps **O(m)**
- La construction est similaire à celle de l'algorithme KMP
- A chaque itération de la boucle
  - ✓ **i** augmente de un, OU
  - ✓ la quantité de décalage **i - j** augmente de au moins un (observez que **F(j - 1) < j**)
- Donc il n y a pas plus que **2m** itérations de la boucle while

```
Algorithm failureFunction(P)
  F[0] ← 0
  i ← 1
  j ← 0
  while i < m
    if P[i] = P[j]
      {we have matched j + 1 chars}
      F[i] ← j + 1
      i ← i + 1
      j ← j + 1
    else if j > 0 then
      {use failure function to shift P}
      j ← F[j − 1]
    else
      F[i] ← 0 { no match }
      i ← i + 1
```

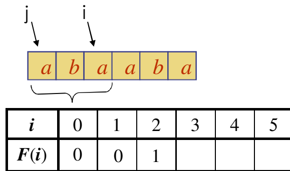

Algorithme KMP



i	0	1	2	3	4	5
$F(i)$	0	0				

```
Algorithm failureFunction(P)
  F[0] ← 0
  i ← 1
  j ← 0
  while i < m
    if P[i] = P[j]
      {we have matched j + 1 chars}
      F[i] ← j + 1
      i ← i + 1
      j ← j + 1
    else if j > 0 then
      {use failure function to shift P}
      j ← F[j − 1]
    else
      F[i] ← 0 { no match }
      i ← i + 1
```

Algorithme KMP

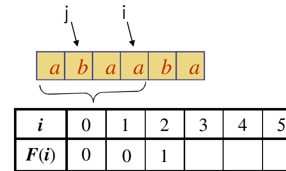


i	0	1	2	3	4	5
$F(i)$	0	0	1			

{on a 'matché' 1 car.}

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```


Algorithme KMP

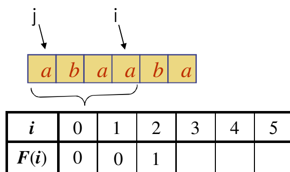


i	0	1	2	3	4	5
$F(i)$	0	0	1			

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

Il n'y a pas de 'match' de 2 car… Il faut verifier s'il y a un 'match' avec 1 car.

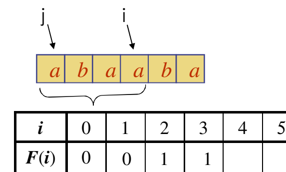
Algorithme KMP



i	0	1	2	3	4	5
$F(i)$	0	0	1			

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

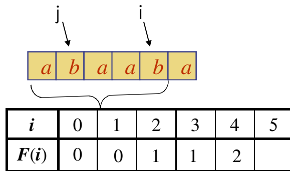

Algorithme KMP



i	0	1	2	3	4	5
$F(i)$	0	0	1	1		

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

Algorithme KMP

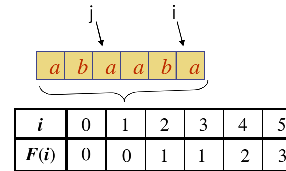


i	0	1	2	3	4	5
***F*(*i*)**	0	0	1	1	2	

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

On avait un ‘match’ de 1 car., maintenant de 2 !

Algorithme KMP

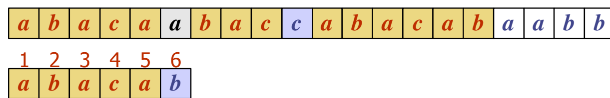


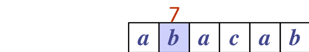
i	0	1	2	3	4	5
***F*(*i*)**	0	0	1	1	2	3

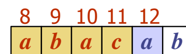
```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

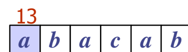
On avait un ‘match’ de 2 car., maintenant de 3 !

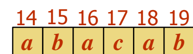
Exemple











j	0	1	2	3	4	5
***P*[*j*]**	*a*	*b*	*a*	*c*	*a*	*b*
***F*(*j*)**	0	0	1	0	1	2