Université de Nantes

UNIVERSITÉ DE NANTES

{département} INFORMATIQUE

Algorithmique et alignement des chaînes
Recherche exacte de motifs

Irena.Rusu@univ-nantes.fr

LS2N, bât. 34, bureau 303

tél. 02.51.12.58.16

Déroulement du cours, partie Recherche de Motifs

- 8h de CM (4h I.Rusu + 4h G. Fertin)
- 8h de TD (4h I. Rusu + 4h G. Fertin)
- …le tout de manière à assurer la continuité du sujet
- … mais pas des intervenants

Ordinateurs éteints

ni autres tablettes etc.

Algorithmique du texte (ou des chaînes)

- Information représentée sous forme texte :
  - Journaux, livres, revues, Internet
  - Information stockée sous forme numérique
  - Génétique, biologie moléculaire etc.

- L'algorithmique du texte apparait dans :
  - Le traitement de l'information (éditeurs de texte, recherche dans des fichiers, sur le Web etc.)
  - La compression des données
  - La fouille des génomes, la comparaison de génomes/ARN/protéines

Recherche de motifs

Principales Références

- Crochemore, M., & Rytter, W. (2003). *Jewels of stringology: text algorithms*. World Scientific.
- M. Crochemore, C. Hancart, T. Lecroq – Algorithms on Strings, Cambridge University Press 2001
- D. Gusfield – Algorithms on Strings, Trees and Sequences, Cambridge Univesity Press 1997

Plan du cours

- Recherche exacte de motifs
  - Par fenêtre glissante (I. Rusu)
  - Par indexation (G. Fertin)
- Recherche approchée de motifs (I. Rusu)
  - Distance de Hamming : Algorithme Kangourou
  - Distance de Levenshtein : Programmation dynamique

Recherche exacte de motifs
Méthodes par fenêtre glissante

- Rappels sur l'efficacité des algorithmes
- Algorithme Z
- Algorithme de Knuth-Morris-Pratt
- Aperçu de l'algorithme de Boyer-Moore

Recherche exacte de motifs
Méthodes par fenêtre glissante

- Rappels sur l'efficacité des algorithmes
- Algorithme Z
- Algorithme de Knuth-Morris-Pratt
- Aperçu de l'algorithme de Boyer-Moore

Quelques mots sur l'efficacité des algorithmes

- Deux points de vue:
  - Mémoire utilisée
  - Temps d'exécution (~ nombre d'opérations)
- Plusieurs paramètres qui fournissent l'unité de mesure:
  - La taille des données en entrée (principalement)
  - Des paramètres définissant la forme des données en entrée

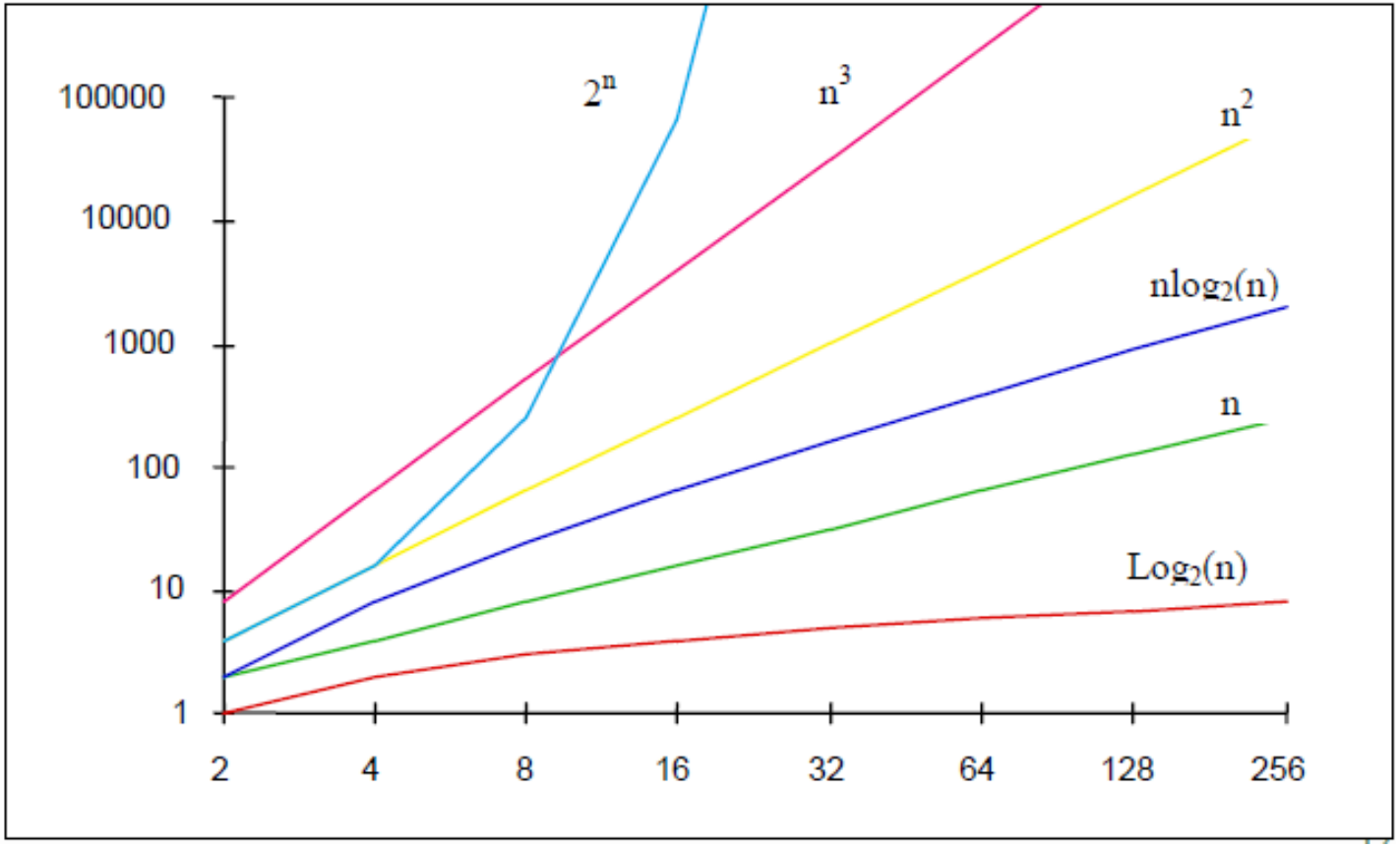
Exemple. Trouver le minimum d'une séquence de n entiers donnée.

Meilleur algorithme : $c_1 \cdot n + c_2$ opérations en tout $\rightarrow O(n)$

(c_1 et c_2 sont des constantes)

Très-très mauvais algorithme : $c_3 \cdot n!$ opérations $\rightarrow O(n!)$

Croissance des fonctions utilisées (échelle logarithmique)



Le plus souvent on écrit log n au lieu de $\log_2 n$

La position relative des courbes reste globalement la même lorsque n augmente, même si on multiplie par (ou on ajoute) des constantes

Conclusions sur le nombre d'opérations (≈ temps d'exécution)

Notation Theta

- Une fonction du genre

  $f(n)= 2n$ ou $g(n)=4n+2$

  suivra le même genre de courbe que $h(n)=n$, et restera assez proche de la courbe de $h(n)$.

- Et donc en général nous ne ferons aucune différence entre f, g ou h. Les ordres de grandeur de f, g, h sont similaires.

- On écrit : $f(n)=\theta(n)$, $g(n)=\theta(n)$, $h(n)=\theta(n)$ (lire « Theta »)

Conclusions sur le nombre d'opérations (≈ temps d'exécution)

Notation Grand « O »

- Une fonction du genre

  $f(n)= 2n$ ou $g(n)=4n+2$

  aura une courbe globalement comme $h(n)=n$, qui elle-même est globalement en dessous de la courbe de (par exemple) $r(n)=n \log n$, à partir d'une certaine valeur seuil (qui n'est pas importante).

- On écrit : $f(n)=O(n \log n)$, $g(n)=O(n \log n)$, $r(n)=O(n \log n)$ (lire grand « O ») pour dire que f, g, n sont « globalement » bornées supérieurement par $n \log n$.

- Evidemment $f(n)\neq\theta(n \log n)$, $g(n)\neq\theta(n \log n)$ mais $r(n)=\theta(n \log n)$

Exemples de temps d'exécution

Not.	O(1)	O(log n)	O($n^{1/2}$)	O(n)	O (nlogn)	O(n^2)	O(n^3)	O($n^{\log n}$)	O(e^n)	O(n!)
N=5	10ns	10ns	22ns	50ns	40ns	250ns	1.25µs	30ns	320ns	1.2µs
N=50	10ns	20ns	71ns	500ns	850ns	25µs	1.25ms	7µs	130j	10^{48}a ns
N=250	10ns	30ns	158ns	2.5µs	6µs	625µs	156ms	5ms	10^{59} ans	
N=10^3	10ns	30 ns	316ns	10µs	30µs	10ms	10s	10s		
N=10^5	10ns	60ns	10µs	10ms	60ms	2.8h	316ans	10^{20} ans		

Recherche exacte de motifs
Méthodes par fenêtre glissante

- Rappels sur l'efficacité des algorithmes
- Algorithme Z
- Algorithme de Knuth-Morris-Pratt
- Aperçu de l'algorithme de Boyer-Moore

Recherche de motifs – Pourquoi ? Comment ?

- Enormément d'applications
  - Internet: moteurs de recherche
  - Bases de données
  - Editeurs de texte (chercher/remplacer)
  - Recherches de mots dans des fichiers (grep …)

- Algorithmique qui se doit d'être efficace

Le problème

- T texte (séquence) de longueur |T|=n sur un alphabet Σ
- P motif de longueur |P|=m<<n

- P est supposé être une séquence aussi
- En général, peut être un ensemble de séquences

- Problème

  Entrée : Texte T, motif P

  Sortie : Toutes les occurrences exactes de P dans T.

Notation

- S séquence sur l'alphabet A (sous-entendu contiguë)
- |S| taille de S

- S[i..j] la sous-séquence de S entre position i et position j
- Préfixe de S : S[1..i] avec i≤ |S|
- Suffixe de S : S[i..|S|] avec i≤|S|
- Sous-séquence/préfixe/suffixe propre de S : différente de S

Problème

Entrée : Texte T, motif P

Sortie : Toutes les positions i dans T t.q.

$$T[i..i+|P|-1]=P$$

Méthode naïve

- |P|=m, |T|=n

Algorithme Naïf
pour j de 1 à n-m+1 faire
si P=T[j..j+m-1] alors afficher j

- Complexité : O(nm)
- Pire des cas: P, T formés d'une seule lettre (la même)

- But : O(m+n)

Idée

- Décaler P de plus d'un caractère après un échec
- Sans risquer de rater des occurrences

Exemple

T=xabxyabxyabxz

P=abxyabxz

Observation : le pré-traitement de P peut nous indiquer

- que le 1er a de P est en position 1
- que le 2ème a de P est en position 5
- Etc.

Pré-traitement fondamental d'une séquence S (1)

- S séquence, i>1 position dans S

- $Z_i(S)$ (ou Z_i si pas de confusion)
  la longueur de la plus longue sous-séquence de S qui commence en i et qui est identique à un préfixe de S

- Z-box de la position i>1 t.q. $Z_i>0$: l'intervalle $i..i+Z_i-1$
- $r_i(S)$ (ou r_i si pas de confusion)
  le point le plus à droite de toutes les Z-box commençant avant i ou en i

- $l_i(S)$ (ou l_i si pas de confusion)
  l'extrémité gauche d'une des Z-box (au choix si plusieurs) dont l'extrémité droite est r_i

Pré-traitement fondamental d'une séquence S (2)

- S séquence, i>1 position dans S
- $Z_i(S)$ (ou Z_i si pas de confusion)

  la longueur de la plus longue sous-séquence de S qui commence en i et qui est identique à un préfixe de S

- But : calculer Z_i, pour tous i>1, en O(|S|)

- Pourquoi ?

  Avec S=P$T, les Z_i t.q. Z_i = m=|P| donnent les occurrences de P

→ le problème de recherche de motif est résolu.

Calcul des Z_i (1)

Algorithme CalculZ(S, k, r, l, Z[ ])

Entrée : S, k>1

r, l : pour le dernier i, $1<i\leq k-1$, t.q. $Z_i>0$, $r=r_i$, $l=l_i$

Z_i, $1<i\leq k-1$ (déjà calculés)

Sortie : valeur de Z_k, nouveaux r et l si $Z_k>0$

Calcul des Z_i (2)

Algorithme CalculZ(S, k, r, l, Z[ ])
Début
si (k>r) alors q ← k
tant que (q≤|S|) et (S[q]=S[q-k+1]) faire q←q+1 ftq
Z_k ← q - k
si Z_k > 0 alors l←k; r←k+Z_k-1 fin si
sinon k'← k – l + 1; b ← r – k +1
si $Z_{k'}$<b alors Z_k←$Z_{k'}$
sinon q← r+1
tant que (q≤|S|) et (S[q]=S[q-r+b]) faire q←q+1 ftq
Z_k← q-k ; r← q-1; l← k
fin si
Fin si
Fin

Complexité : O(|S|) (globale)

Algo Z de recherche de P dans T

Algorithme AlgoZ

Entrée : P et T

Sortie : les positions des occurrences de P dans T

début

S=P$T; r←0; l←0; Z initialisé à 0 partout

pour k← 2 à |S| faire

CalculZ(S, k, r, l, Z[ ])

Si Z_k=m alors Afficher(k-(m+1)) fin si

fin pour

fin

Complexité :

O(|P|+|T|)

en temps et en espace

Pourquoi chercher d'autres algorithmes ?

- Knuth-Moris-Pratt : temps linéaire; importance historique; a été généralisé à un ensemble de motifs (toujours en temps linéaire)
- Boyer-Moore : le pire des cas est quadratique, mais en pratique il n'examine qu'une partie des caractères de T → sub-linéaire souvent. Une version linéaire dans le pire des cas existe aussi.
- Arbre des suffixes : pré-traitement de T, et recherche proportionnelle à |P|; des applications beaucoup plus complexes

Recherche exacte de motifs
Méthodes par fenêtre glissante

- Rappels sur l'efficacité des algorithmes
- Algorithme Z
- Algorithme de Knuth-Morris-Pratt
- Aperçu de l'algorithme de Boyer-Moore

Algorithme de Knuth-Morris-Pratt (KMP)

- $sp_i(P)$: longueur du plus long suffixe propre de P[1..i] qui s'aligne à un préfixe de P (sauf $sp_1=0$ par définition)

- $sp'_i(P)$: longueur du plus long suffixe propre de P[1..i] qui s'aligne à un préfixe de P (sauf $sp'_1=0$ par définition) avec la condition supplémentaire que $P[i+1] \neq P[sp'_i+1]$

$$\rightarrow sp'_i \leq sp_i$$

Idée

P[1..sp'$_{i-1}$] est aligné directement sur P[i-sp'$_{i-1}$ .. i-1]

- Avantages
  - P se déplace de plus d'un caractère
  - Les premiers sp'$_i$ caractères sont déjà calculés
- Complexité
  - Tout caractère de T comparé avec égalité exactement 1 fois
  - Au plus une comparaison avec différence par « shift »

- Total : O(|T|)

Correction

Théorème.

Pour tout alignement de P avec T, si les caractères 1 .. i-1 de P sont identiques avec les caractères alignés de T, mais P[i]≠T[k] (le caractère en face de P[i] sur T), alors P peut être déplacé à droite de i-1-sp'_{i-1} places sans perdre des occurrences de P dans T.

Preuve. Par l'absurde.

Algorithme KMP

Algorithme KMP

Entrée : P (taille m) et T (taille n)

Sortie : les positions des occurrences de P dans T

```
début
    CalculerSPprim(P); i←1; k←1; //pré-traitement
   tant que k+(m-i) ≤n faire
      tant que (i≤m) et (P[i]=T[k]) faire i←i+1; k←k+1 fin tq
      si i=m+1 alors Afficher(k-m) fin si
      si i=1 alors k←k+1 fin si
      i← sp'i-1+1
    fin tq
fin
```

Complexité recherche
(sans CalculerSPprim)
O(|T|)
en temps
et en espace

CalculerSPprim

Algorithme CalculerSPprim

Entrée : P (taille m)

Sortie : les valeurs sp'_i, i=1, 2, …, m

```
début
    pour i← 1 à m faire sp'_i←0 fin pour
   pour j← m à 2 (par pas de -1) faire
          si Z_j(P)>0 alors
                    i← j + Z_j(P)-1;
                    sp'_i←Z_j(P)
        fin si
   fin pour
fin
```

Complexité prétraitement

O(|P|)

en temps
et en espace

Conclusions jusqu'ici

- Plusieurs algorithmes en O(m+n) existent
- En théorie, tant qu'on se limite au O(), ils sont tous aussi bons
- En pratique, les constantes (oubliées par O()) peuvent faire la différence
- Sur une recherche de P dans T, la différence sera négligeable
- Sur plusieurs (voire beaucoup) de recherches, la différence totale peut être importante

→ autres algorithmes par la suite

Recherche exacte de motifs
Méthodes par fenêtre glissante

- Rappels sur l'efficacité des algorithmes
- Algorithme Z
- Algorithme de Knuth-Morris-Pratt
- Aperçu de l'algorithme de Boyer-Moore

Algorithme KMP (rappel) vs. Boyer-Moore (BM)

- Parcourt le texte et le motif de gauche à droite
- Pré-traitement en O(|P|)
- Recherche en O(|T|)
- Remarque : aucune des phases ne dépend de la taille de l'alphabet
- Généralisable à plusieurs motifs

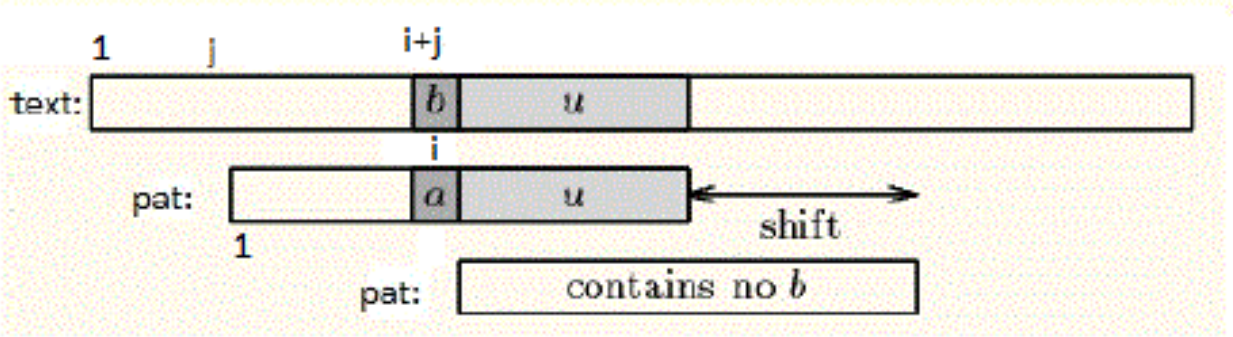
- Parcourt le texte de gauche à droite et le motif de droite à gauche
- Peut-être sous-linéaire (ne parcourt pas tout T)
- Plus rapide quand |P| augmente
- Plus rapide lorsque |Σ| augmente par rapport à |P|
- Pas très rapide pour petits alphabets, et motifs courts

Algorithme BM - Idées

- Aligner P avec une sous-séquence de T
- Parcourir P de droite à gauche
- Règle du mauvais caractère : éviter d'avoir plusieurs tentatives échouées pour une même position dans T
- Règle du bon suffixe : sur une position de T alignée avec succès une fois, toujours aligner avec succès

La règle du mauvais caractère (pat=P, text=T)

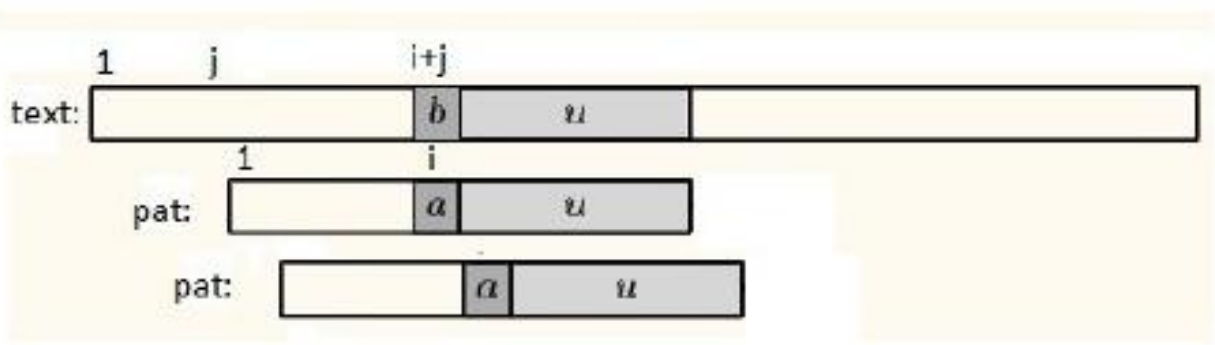
- P[i+1..m]=T[i+j+1..j+m]=u
- P[i] ≠ T[i+j]



- Si b n'apparaît nulle part dans P, alors P est déplacé après b
  → saut (« shift ») pouvant être important

La règle du mauvais caractère (pat=P, text=T)

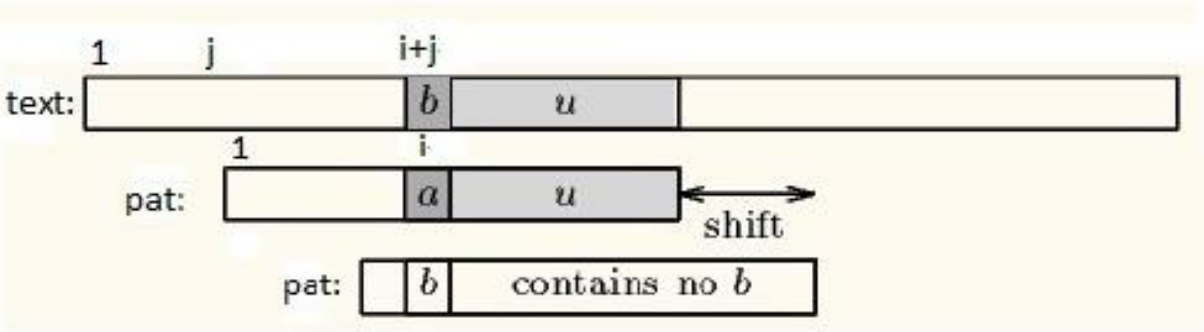
- P[i+1..m]=T[i+j+1..j+m]=u
- P[i] ≠ T[i+j]



- Si b apparaît dans P à droite de i, alors P est déplacé à droite de 1
  → pas (vraiment) de saut

La règle du mauvais caractère (pat=P, text=T)

- P[i+1..m]=T[i+j+1..j+m]=u
- P[i] ≠ T[i+j]

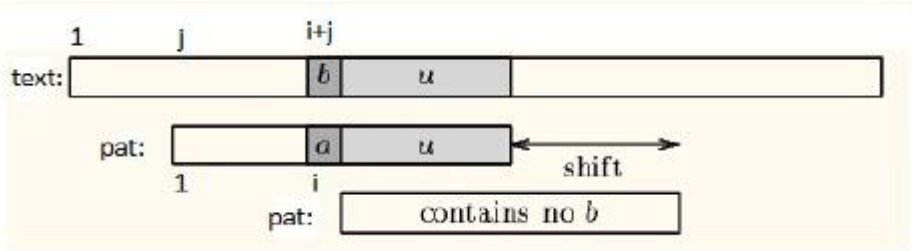


- Si b apparaît dans P seulement à gauche de i, alors P est déplacé de sorte à aligner son b le plus à droite sur le b de T

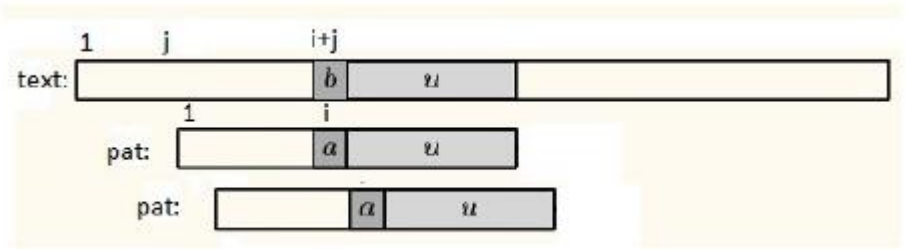
  → saut pouvant être important

En résumé – règle du mauvais caractère

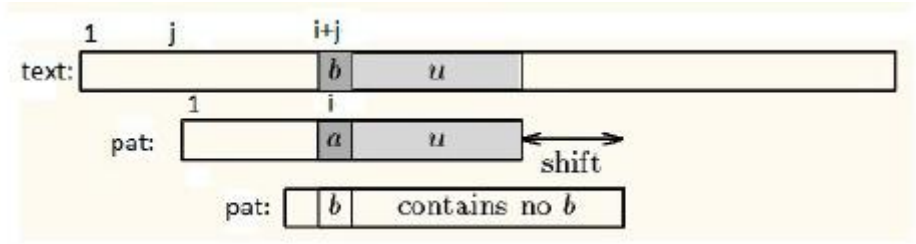
- Pas de b dans P



- Existe b à droite de i

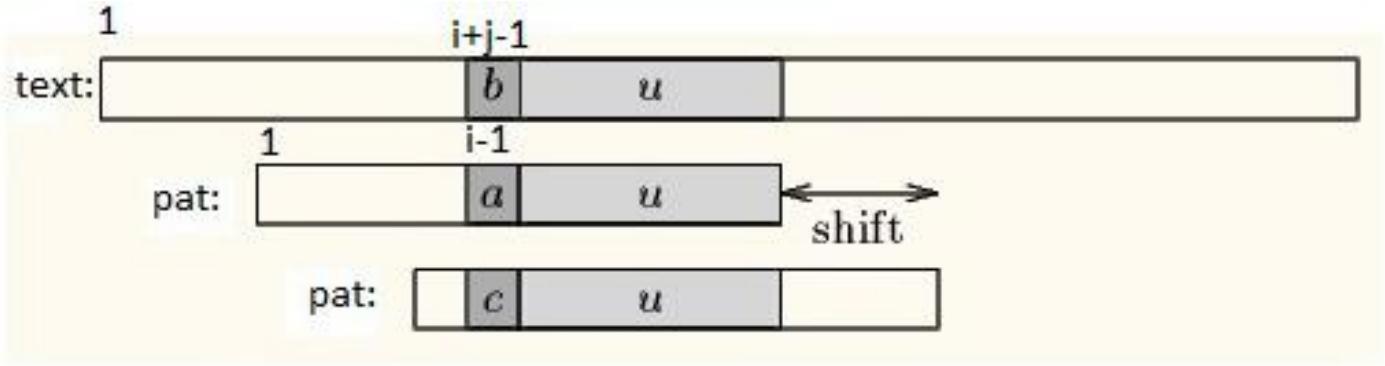


- Tous les b sont à gauche de i



La règle du bon suffixe (pat=P, text=T)

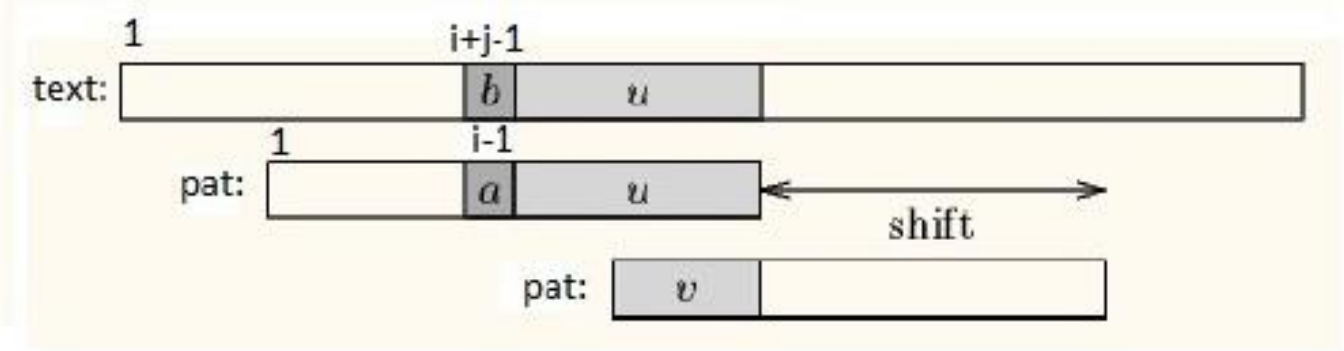
- P[i..m]=T[i+j..j+m]=u (même si u est le mot vide)
- P[i-1]≠T[i+j-1]



- Aligner u= T[i+j..j+m] avec l'occurrence de u dans P qui est la plus à droite avec les propriétés (si elle existe) :
  - Ce n'est pas un suffixe de P
  - Est précédée par un caractère différent de P[i-1]

La règle du bon suffixe (pat=P, text=T)

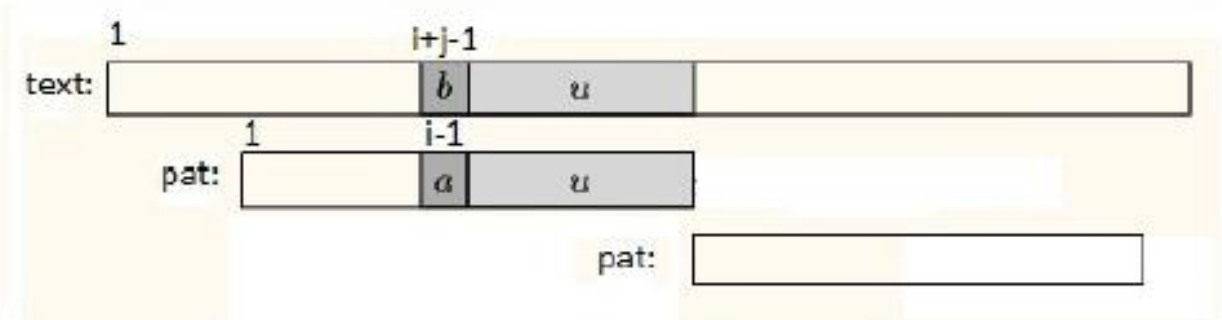
- P[i..m]=T[i+j..j+m]=u
- P[i-1]≠T[i+j-1]



- Si une telle occurrence n'existe pas, aligner un préfixe v de P avec le suffixe v de T[i+j..j+m] de sorte que v soit aussi long que possible (si un tel v existe).

La règle du bon suffixe (pat=P, text=T)

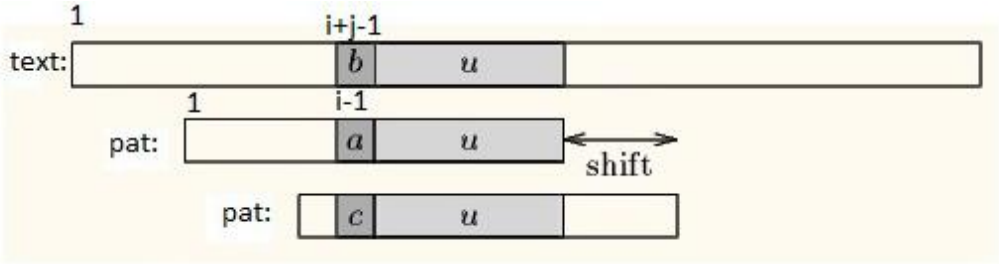
- P[i..m]=T[i+j..j+m]=u
- P[i-1]≠T[i+j-1]



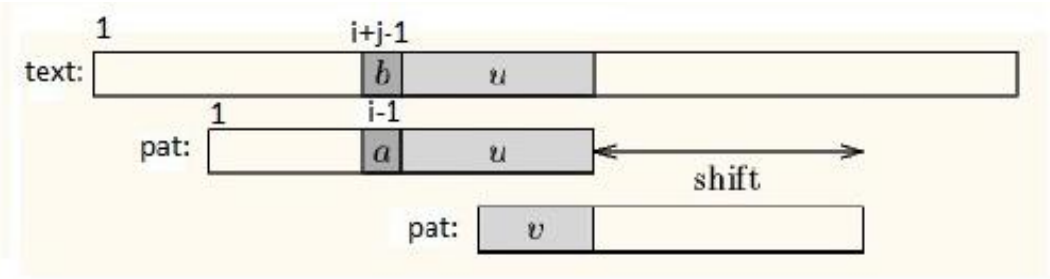
- Si un tel v n'existe pas, càd que le plus long suffixe v de u qui est identique à un prefixe de P est le mot vide, alors on déplace P juste après T[j+m]

En résumé – règle du bon suffixe

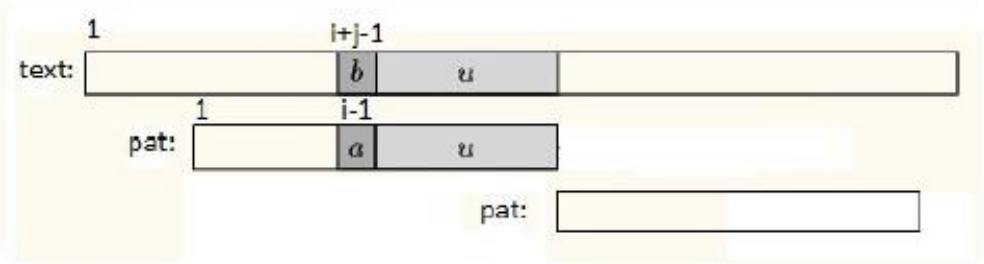
- Existe occurrence de u dans P qui n'est pas un suffixe de u



- Il n'existe pas de telle occurrence, mais existe prefixe de P identique à un suffixe de u



- Aucune des deux conditions précédentes



Correction

Théorème

La règle du bon suffixe permet de déplacer P le long de T sans perdre des occurrences de P dans T.

Preuve. Par contradiction.

Algorithme BM

début

k←m; l2←longueur du plus long suffixe de P[2..m] qui est aussi un préfixe de P;//pré-traitement

tant que k ≤n faire

i←m; h←k;

tant que (i>0) et (P[i]=T[h]) faire i←i-1; h←h-1 fin tq

si i=0 alors Afficher (k-m+1) ; k← k+m-l_2

sinon déplacer P (augmenter k) par le maximum des valeurs déterminées respectivement par les règles du mauvais caractère et du bon suffixe*

finsi

fin tq

fin

Attention, dans la règle du bon suffixe on aura i à la place du i-1 présenté sur les exemples lorsqu'on arrivera à cette ligne

Complexité

- En utilisant les deux règles (tel que l'algorithme est écrit) :
  O(|T|.|P|) si le motif se trouve dans le texte
  O(|T|+|P|) seulement si le motif ne s'y trouve pas

- Sur des textes en langage naturel, c'est presque toujours sub-linéaire : bien inférieur à |T|

- Une (relativement) petite modification réduit la complexité à O(|T|+|P|) même si le motif se trouve dans le texte

Conclusions

- La complexité théorique O() identifie les « grosses » différences de comportement entre algorithmes, et en plus « au pire »
- Mais pas les « petites » différences, qui – accumulées en cas d'utilisations multiples – peuvent faire la différence
- Ceci explique les divers algorithmes, tous linéaires en théorie, mais avec des comportements différents en fonction du type des données