# NSI Terminale - Algorithmie

Recherche textuelle

qkzk

2020/03/14

# Recherche textuelle

# Pourquoi parler de recherche textuelle ?

**Qu'est-ce qu'un texte ?**

Quelques exemples :

- 101010101010001
- ATCGTTTATGCGAA
- un texte
- la concaténation de toutes les pages web

# Qu'est-ce qu'un texte ?

**Définition**

Un texte est une suite finie de symboles.

# Quelle recherche textuelle ?

- CTRL + F dans un document ?
- sur un moteur de recherche en ligne ?

# Recherche dans un texte connu à l'avance (livres, sites. . . )

On dispose alors généralement d'un **index**

Index

Les numéros de page en gras renvoient aux cartes.

# Quand ne construit-on pas d'index ?

Lorsque le texte possède certaines propriétés

- court

# Quand ne construit-on pas d'index ?

Lorsque le texte possède certaines propriétés

- court
- modifiable

# Quand ne construit-on pas d'index ?

Lorsque le texte possède certaines propriétés

- court
- modifiable
- non connu à l'avance

Notez qu'un pdf de 1000 pages ne remplit aucun de ces critères

# Quand ne construit-on pas d'index ?

Lorsque le texte possède certaines propriétés

- court
- modifiable
- non connu à l'avance

Notez qu'un pdf de 1000 pages ne remplit aucun de ces critères

***Comment rechercher dans un texte, sans index ?***

# Parcours de texte - recherche naïve

T = a t a g a c a c a a t a t a c t g a c a c g a t

*Puis-je trouver le mot* $P = atatac$ *?*

# Parcours de texte - recherche naïve

```
T = a t a g a c a c a a t a t a c t g a c a c g a t
    a t a t a c
      a t a t a c
      x
```

*Puis-je trouver le mot* $P = atatac$ *?*

# Parcours de texte - recherche naïve

```
T = a t a g a c a c a a t a t a c t g a c a c g a t
    a t a t a c
      a t a t a c
        a t a t a c
        v x
```

*Puis-je trouver le mot $P$ = atatac ? Tester la présence de $P$ à chaque position de $T$*

# Parcours de texte - recherche naïve

```
T = a t a g a c a c a a t a t a c t g a c a c g a t
    a t a t a c
      a t a t a c
        a t a t a c
          a t a t a c
            x
```

*Puis-je trouver le mot* $P$ = `atatac` *?*

*Tester la présence de* $P$ *à chaque position de* $T$

*Au pire :* $|T| \times |P|$ *comparaisons.*

# Algorithme de Boyer Moore Horspool

# Exemple

Imaginons qu'on cherche le motif `ALUN` dans le texte `LUNALINALUNA`.

On commence par positionner le motif en début de texte :

```
LUNALINALUNA
ALUN
```

On lit le motif **de droite à gauche** :

Le `N` du motif ne correspond pas au `A` du texte. On décale le motif jusqu'à positionner le `A` du motif : c'est-à-dire qu'on effectue un décalage de 3.

# Exemple (suite)

```
LUNALINALUNA
   ALUN
```

Cette fois, problème entre le `U` du motif et le `I` du texte. Comme il n'y a pas de `I` dans le motif, décalage de 3 :

```
LUNALINALUNA
      ALUN
```

Non correspondance du `N` du motif avec le `U` du texte, décalage de 1.

```
LUNALINALUNA
       ALUN
```

Et on a trouvé !

# Avec des indices

**Dernière occurrence**

On commence par créer un tableau associant chaque caractère possible à la longueur du motif.

Ensuite, pour chaque caractère d'indice `i` du motif, la distance est donnée par `taille - 1 - i`

ce qui donne :

# pseudo code : dernière occurrence

```
dernière occurrence (motif)
  m = longueur du motif
  créer un dictionnaire associant chaque lettre à m
  pour i allant de 0 à m-2,
      dictionnaire [ motif[i] ] = m - 1 - i
  fin du pour
  retourner le dictionnaire
```

# dernière occurence : remarque importante

Faîtes bien attention à la boucle:

```
pour i allant de 0 à m-2
```

Si on va jusqu'au *dernier* caractère du motif, il se verra attribuer la distance 0 . . . ce qui entrainera une boucle infinie dans la partie suivante !

# Boyer-Moore-Horspool

- on commence avec `j` $= 0$
- on itère jusqu'à ce que `j = taille du texte - taille du motif`

  on parcourt le motif à partir de la fin, donc `i = taille du motif`.

  on recule sur `i` jusqu'à arriver à 0 ou jusqu'à ce que les caractères ne se correspondent plus.

  - si `i = -1` alors

    le motif commence en j et on augmente `j` de 1

  - sinon

    on augmente `j` de la distance correspondant à cette position différente dans le texte.

# Pseudo-code Boyer-Moore-Horspool

```
Algorithme Boyer-Moore-Horspool(x, t):
  '''
  x : motif, t : texte, m : longueur motif, n : celle du te
  d : tableau des dernières occurrences du motif
  '''
  tant que j <= n - m,
    i = m - 1
    tant que i >= 0 et t[j + i] = x[i]:
      i = i-1
    fin tant que
    si i = -1 alors
      j est une occurrence de x
      j = j + 1
    sinon
      j = j + d[ t[j + i] ]
```

# Preuve

- L'algorithme se termine bien.

  En effet, le tableau des dernières occurrences comporte forcement des nombres strictement positifs. (cf remarque le concernant)

  Dans le pire des cas, on augmente `j` de 1 à chaque tour de la boucle extérieure. La boucle intérieure voit `i` augmenter de 1 à chaque fois.

# Preuve (suite)

- L'algorithme trouve toutes les occurrences du motif dans le texte
  - En effet, si `j` augmente de 1 à chaque tour, on réalise le même travail que l'algorithme naïf et on rencontre forcément toutes les occurrences.

# Preuve (suite)

- L'algorithme trouve toutes les occurrences du motif dans le texte
  - En effet, si `j` augmente de 1 à chaque tour, on réalise le même travail que l'algorithme naïf et on rencontre forcément toutes les occurrences.
  - Sinon, c'est que la distance issue du tableau des occurrences est supérieure à 1.

    Deux cas se présentent :

# Preuve (suite)

- L'algorithme trouve toutes les occurrences du motif dans le texte
  - En effet, si `j` augmente de 1 à chaque tour, on réalise le même travail que l'algorithme naïf et on rencontre forcément toutes les occurrences.
  - Sinon, c'est que la distance issue du tableau des occurrences est supérieure à 1.

    Deux cas se présentent :
    - `d = m` (la lettre n'est pas dans le motif.) Alors il est impossible que le motif commence dans le texte avant `j+d`

# Preuve (suite)

- L'algorithme trouve toutes les occurrences du motif dans le texte
  - En effet, si `j` augmente de 1 à chaque tour, on réalise le même travail que l'algorithme naïf et on rencontre forcément toutes les occurrences.
  - Sinon, c'est que la distance issue du tableau des occurrences est supérieure à 1.

    Deux cas se présentent :
    - `d = m` (la lettre n'est pas dans le motif.) Alors il est impossible que le motif commence dans le texte avant `j+d`
    - `d < m` et cette fois, la redondance de la lettre dans le motif nous impose de la faire correspondre.

      Si `j` augmente de `k < d`, il est impossible que ce caractère corresponde à un caractère `[j + k]` du texte.

# Compléxité

- En *pratique* et en *moyenne* on constate que la complexité est *sous-linéaire*

  Elle de l'ordre de $O(3n)$ où $n =$ taille du texte.

- Le *pire des cas* se présente pour un motif de la forme `baa...aa` avec un texte `aaaaa...aaaa`. La fonction de décalage n'apporte aucune amélioration par rapport à la méthode naïve puisque $d(a) = 1$.

  On doit donc comparer chaque caractère du motif avec chaque caractère du texte.

  Ainsi, on a fait `n-m` décalages et chaque décalage demande `m` comparaisons.

  Le pire des cas est en $O(n \times m)$

# Implémentation

Ici l'implémentation en Python : bmh.py

# Compléments

Cet algorithme comporte deux des trois idées principales de la version complète, dîtes de *Boyer-Moore* :

1. comparer en parcourant le motif par la droite,
2. utiliser un tableau de distances pré-calculé sur les motifs,
3. utiliser un autre tableau, dît du bon préfixe.

Pour en savoir plus, vous pouvez consulter l'algorithme complet dans les éléments d'algorithmique de *Beauquier, Berstel* et *Chrétienne* ou l'article wikipédia.

Remarquons finalement que l'algorithme de Boyer-Moore (complet) est généralement implémenté nativement dans les langages modernes (méthode `find` et `index`, mot clé `in` de Python) et qu'il est souvent le plus efficace.

Citons aussi l'algorithme de *Knuth-Morris et Pratt* qui emploie aussi

# Travaux dirigés

1. Implémenter la recherche naïve en Python
2. Implémenter l'algorithme de Boyer-Moore-Horspool en Python
3. Comparer les vitesses d'exécution sur un jeu d'exemple avec `time` ou mieux `timeit`

- on illustrera les pires cas,
- les cas moyens (mot choisi au hasard dans un texte)
- construire quelques figures pour des tailles de motifs croissantes (5, 10, 20).

# Compléments pour les travaux dirigés

4. Retour sur l'indexation :

   4.1 Partons d'un texte de grande taille (>100ko), créer un index des mots qu'il contient.

   - doit-on choisir la première occurrence ? Toutes les occurrences ?
   - on commencera par transformer le texte en minuscule, retirer la ponctuation et les accents puis le découper à chaque espace.

   4.2 Comparer les temps d'execution pour la recherche d'une centaine de mots du texte

   - pour la recherche naïve,
   - par l'algorithme de Boyer-Moore-Horspool,

# Extensions

5. Autres algorithmes

   - implémentez l'algorithme de Boyer-Moore présenté dans le *Beauquier, Berstel et Chrétienne*
   - implémentez l'algorithme de *Knuth, Morris et Pratt*.
   - comparez les temps d'exécution de tout le monde sur différents cas.

6. Distance de *Leveinshtein*

   - Étudiez et implémentez la distance *Levenshtein*.
   - Créez un correcteur orthographique.

     Votre programme doit prendre un texte en paramètre d'entrée et proposer une des corrections pour chaque mot mal orthographié.

     Une option doit permettre de rectifier sauvagement tout le texte.