# Recherche textuelle

par **Julien Velcin**
Université Lumière Lyon 2, laboratoire ERIC
http://mediamining.univ-lyon2.fr/velcin

DIU Enseigner l'informatique au lycée
juin 2020

# Plan du cours

- Traitement Automatique des Langues
- Recherche d’information
- Recherche de motifs exacts
- Ouverture

# Traitement automatique des Langues

- Champ de l'**Intelligence Artificielle** qui porte sur le traitement automatique de la (des) langue(s)
- Les applications du *Natural Language Processing* (NLP) sont nombreuses :
  - recherche d'information dans les grandes bases textuelles
  - résumé automatique pour la veille
  - fouille de l'opinion et des sentiments
  - traduction automatique
  - robots conversationnels

# Recherche d'information

- On a déjà une idée de ce qu'on cherche (*top down*) et on exprime une requête sous forme de :
  - une chaîne de caractères (mot voire suite de mots) : coronavirus
  - un motif : covid-*
  - un ensemble de mots-clefs : raoult chloroquine
  - une question : A quelle date a débuté le confinement en France ?
- L'indexation de (la plupart) des milliards de sites Web est réalisée par des robots et l'on peut ensuite interroger l'index via les moteurs de recherche (*search engines*)

# Recherche d'un motif dans une chaîne

- Un **texte** est codé comme une suite (chaîne) de caractères dans un alphabet donné, par ex. : « L'objectif de cette formation est d'accompagner les futur·e·s enseignant·e·s d'informatique dans l'acquisition des connaissances et compétences minimales nécessaires à l'enseignement de la nouvelle spécialité Numérique et Sciences Informatiques (NSI) en classes de Première et de Terminale, dans le cadre de la réforme du lycée. Elle vise aussi à proposer aux enseignants ».
- Alphabet = {A,B...Z,a,b...Z,0,1...9, etc.[1]}
- Rechercher un **motif** (par ex. "enseignant") dans cette chaîne est une tâche similaire à celle de chercher un motif dans une séquence biologique, par ex. "GCAG" dans "GGCAGCCGAACCGCAGCAGCAC"
- On ne va s'intéresser qu'à la recherche de **motifs exacts** ici

[1] : etc. contient tous les caractères « spéciaux » que l'on veut : ù,à,é,@...

# Recherche de motifs exacts

- **Texte :** chaîne de caractères

Texte appelé **T**, de longueur n

- **Motif :** un texte aussi, mais de « petite » taille

Motif appelé **M**, de longueur **m**

- Recherche d'un motif M dans un texte T :
  - recherche de toutes les occurrences de M dans T
  - recherche exacte ⇒ on n'autorise pas d'erreur

# Applications fréquentes

- Recherche d'un mot dans un document (ctrl-f)

  chercher M = « algorithme » dans le programme NSI de Terminale

  n = 201 097, m = 10 ; 9 occurrences

- Recherche d'une sous-séquence d'intérêt dans une séquence biologique

  chercher M = « TTGACA » (promoteur de gène) dans le chromosome 1

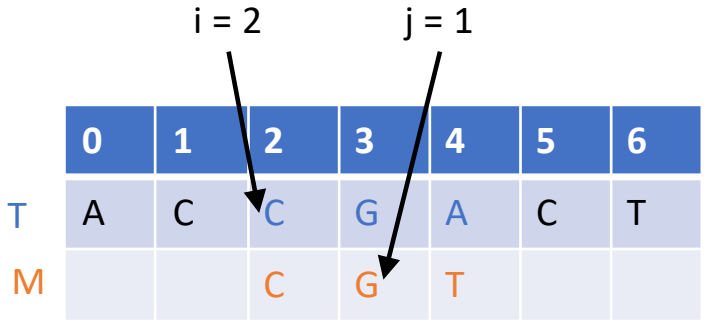
- On considérera dans la suite que $m << n$.

# Recherche par fenêtre glissante

- **Idée :** positionner le motif M à différentes positions de T
- Pour chaque position choisie, tester si M apparaît dans T (c'est-à-dire M[**0**..m-1]=T[**i**..i+m-1]?)
- Décaler M (changement de position dans T) et recommencer

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **…** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | G | G | C | A | G | C | C | G | A | A | C | C | G | C | A | G | C | A | G | C |
| M | | | | A | G | C | A | | | | | | | | | | | | | |
| M | | | | | A | G | C | A | | | | | | | | | | | | |
| M | | | | | | | | | | | | | | | A | G | C | A | | |

# Précisions et vocabulaire




Dans toute la suite :

- T et M sont stockés dans des tableaux

⇒ Accès à T[i] ou M[j] en temps constant **O(1)**

- T[] et M[] numérotés à partir de zéro

⇒ T[0..n-1] et M[0..m-1] (rappels : T de longueur n et M de longueur m)

⇒ au besoin, on pourra écrire T[0..i] pour dire « le texte T pris jusqu'à la position i incluse » (idem pour M[0..j])

- i = position dans T et j = position dans M
- Tester si M est présent à la position i de T se fait caractère par caractère (c'est-à-dire : M[j] est-il égal à T[i+j]?)
- M[j]=T[i+j] ⇒ match
- M[j]≠T[i+j] ⇒ mismatch
- Alphabet Σ, de taille σ (ex : Σ={A,C,G,T}, de taille σ=4)

# Algorithme de recherche naïve

- Algorithme de recherche par fenêtre glissante :
  - tester si M apparaît dans T
  - pour chaque position i de T, à partir de 0

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | G | C | A | G | C | C | G | A | A | C | C | G | C | A | G | C | A | G | C |
| A | G | C | A | | | | | | | | | | | | | | | | |
| | A | G | C | A | | | | | | | | | | | | | | | |
| | | A | G | C | A | | | | | | | | | | | | | | |
| . | . | . | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | A | G | C | A | | |

# Algorithme de recherche naïve - exercices

1. Quelle est la dernière valeur de i à tester ?
2. Écrire l'algorithme de Recherche Naïve
3. Quelle est la complexité temporelle au mieux de la Recherche Naïve ? Pour quelle forme des données ?
4. Quelle est la complexité temporelle au pire de la Recherche Naïve ? Pour quelle forme des données ?
5. Supposons que le motif M ne contient pas deux fois la même lettre. Écrire un algorithme de recherche exacte de motif qui exploite cette information. Discuter de ses complexités temporelles au mieux et au pire.

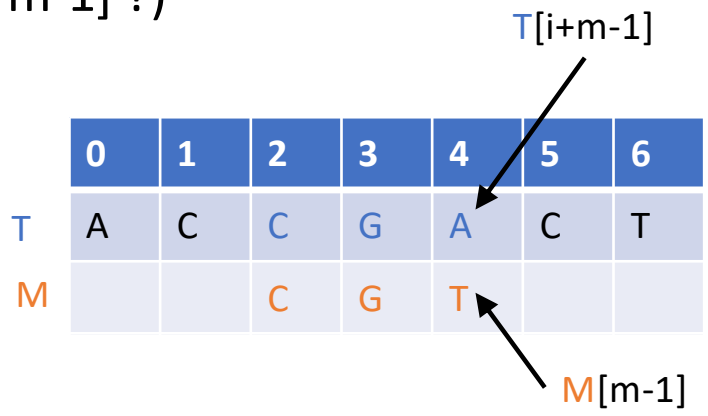
# Quelques mots sur cet algorithme

- Cet algorithme est très gourmand car il n'a aucune "mémoire" des opérations qui ont été faites précédemment
- On peut l'améliorer considérablement, en particulier en cherchant à **décaler** le motif de plus d'1 case à chaque fois
- Ce décalage est souvent permis par un prétraitement adapté du motif (par ex. en trouvant les sous-motifs qui le composent)

# Algorithme de Boyer-Moore - généralités

- dû à Robert S. Boyer et J. Strother Moore – 1977
- utilisé le plus souvent dans les éditeurs de texte (tel quel ou optimisé)
- algorithme de recherche par fenêtre glissante :
  - M « glisse » de gauche à droite le long de T
  - mais la comparaison M[0..m-1] vs T[i..i+m-1] se fait **de droite à gauche**

  (on commence donc par interroger : M[m-1]=T[i+m-1] ?)
- décalage de M en fonction de deux règles :
  - bon suffixe
  - mauvais caractère

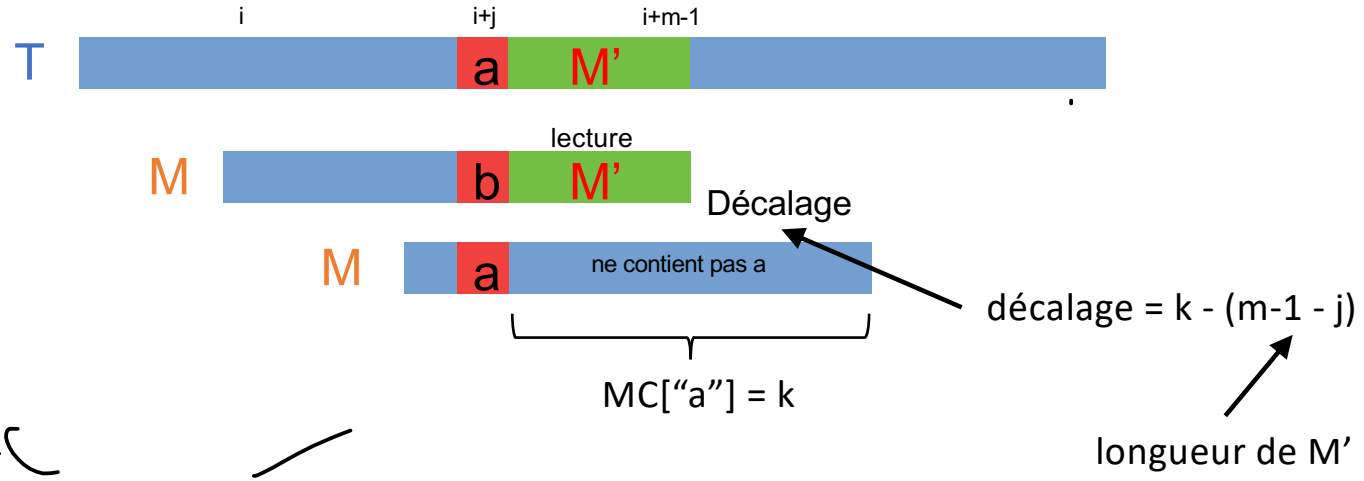


# Règle du mauvais caractère (cas 1)

- On aligne le caractère T[i+j] = a avec son occurrence la plus à droite dans M[0..m-2]

a,b : caractères

M' : suffixe de M



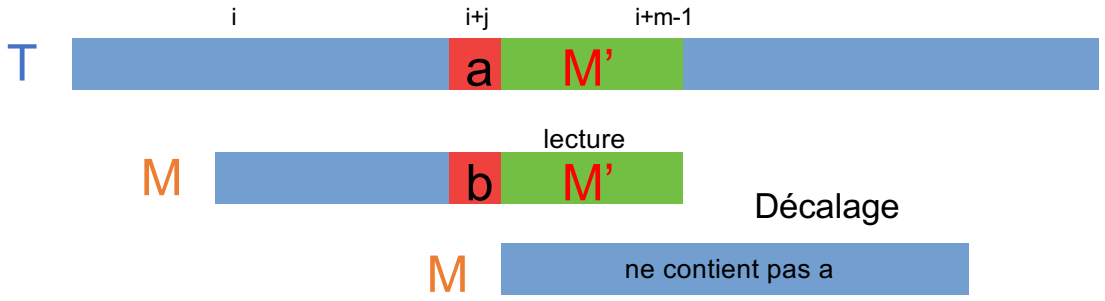

On cherche le prochain a de M qui s'aligne avec le a qui est dans T

# Règle du mauvais caractère (cas 2)

- Si T[i+j]=a **n'est pas** dans M, la prochaine position à tester pour M est la position i+j+1 (attention si i+j+1+m >= n)

a,b : caractères

M':suffixe de M



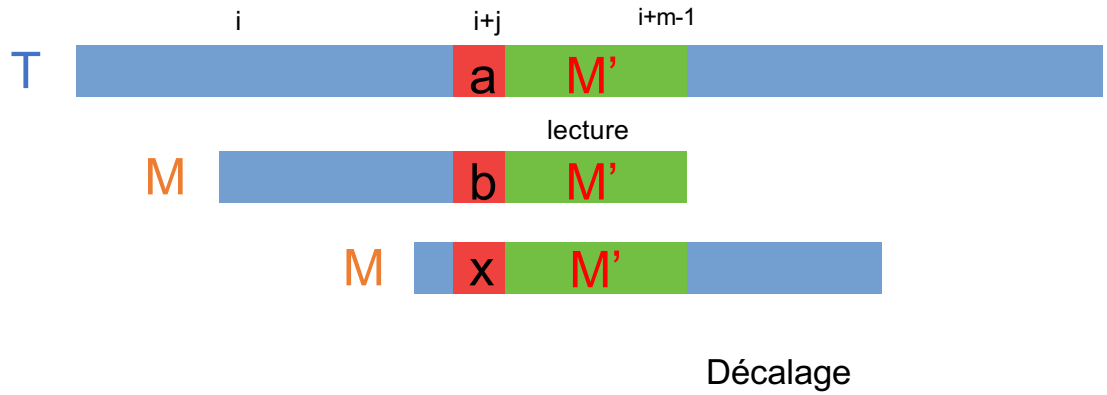
# Règle du bon suffixe (cas 1)

- S'il **existe** un x.M' à gauche de b.M' dans M, avec x≠b...

...on choisit celui qui est le plus proche de b.M' (donc le plus à droite dans M)

M' : suffixe de M

a,b,x : caractères

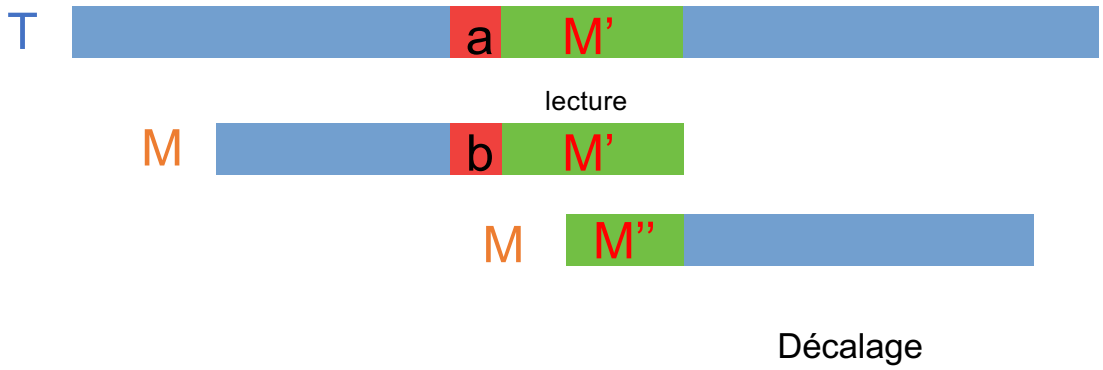


# Règle du bon suffixe (cas 2)

- Si un tel x.M’ **n’existe pas** : trouver M’’, le plus long préfixe de M qui en est aussi un suffixe

M’ : suffixe de M

a,b : caractères

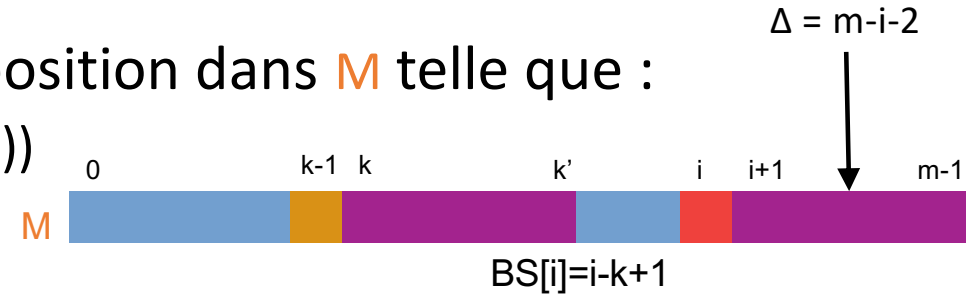


# Mise en œuvre des règles BS et MC (1)

- Pré-traitement du motif M
  - deux tableaux, construits à partir de M :
    - MC (Mauvais Caractère)
    - BS (Bon Suffixe)
- Tableau MC : indicé sur les caractères de Σ
  - MC[0..σ-1] (rappel: σ = taille de l'alphabet)
  - Pour tout caractère c de Σ, MC[c] = nombre de positions à « remonter » dans M depuis M[m-1] pour trouver c
- Cas particuliers :
  - si c=M[m-1], ignorer M[m-1]
  - si c n'est pas dans M, MC[c]=m
- Exercice : donner le contenu de MC pour M=GCAGAGAG avec Σ={A,C,G,T}

# Mise en œuvre des règles BS et MC (2)

- Tableau BS : BS[0..m-1]
- BS[i] = i-k+1, où k est la plus grande position dans M telle que :
  - M[k..k'] = M[i+1..m-1] (avec k'=k+(m-i-2))
  - M[k-1] ≠ M[i]




- si un tel k n'existe pas :
  - rechercher dans M la position p du plus long suffixe M[p..m-1] de M qui est aussi un préfixe de M
  - BS[i]=p


- Exercice : donner le contenu de BS pour M=GCAGAGAG

# Algorithme/Complexité du Pré-traitement de M

- On parle ici de la complexité temporelle

(officiellement : pas au programme !)

- Remplissage de MC (Mauvais Caractère) : $O(m+\sigma)$

remarque : ce n'est pas si compliqué (laissé en exercice)

- Remplissage de BS (Bon Suffixe) :

peut se faire sans (trop) de difficultés en $O(m^2)$

mais il existe un algorithme plus « subtil » en $O(m)$

- Conclusion : pré-traitement de M en $O(m+\sigma)$

# Algorithme de Boyer-Moore

```
Entrée : Texte T, Motif M
## Pré-traitement de M
Calcul de BS[]
Calcul de MC[]
## Recherche de M dans T
i←0
Tant que i <= n-m faire
        j←m-1 ## on lit M de droite à gauche !
        Tant que j >= 0 et M[j] = T[i+j] faire
                j← j - 1
        FinTantQue
Si j < 0 alors
        Ecrire ("Motif trouvé en", i)
        i ← i + BS[0]
        ## décalage du motif
Sinon
        i ← i + max(BS[j], MC[T[i+j]]+j-m+1)
        ## décalage du motif
FinSi
```

FinTantQue

# Exemple

T = G C A T C G C A G A G A G T A T A C A G T A C G

M = G C A G A G A G

BS[7]=1 ; MC[A]+7-8+1=1 ⇒ décalage de 1

T = G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

BS[5]=4 ; MC[C]+5-8+1=4 ⇒ décalage de 4

T = G C A T C G C A G A G A G T A T A C A G T A C G

G C A G A G A G

Motif trouvé ⇒ décalage de BS[0]=7

T = G C A T C G C A G A G A G T A T A C A G T A C G

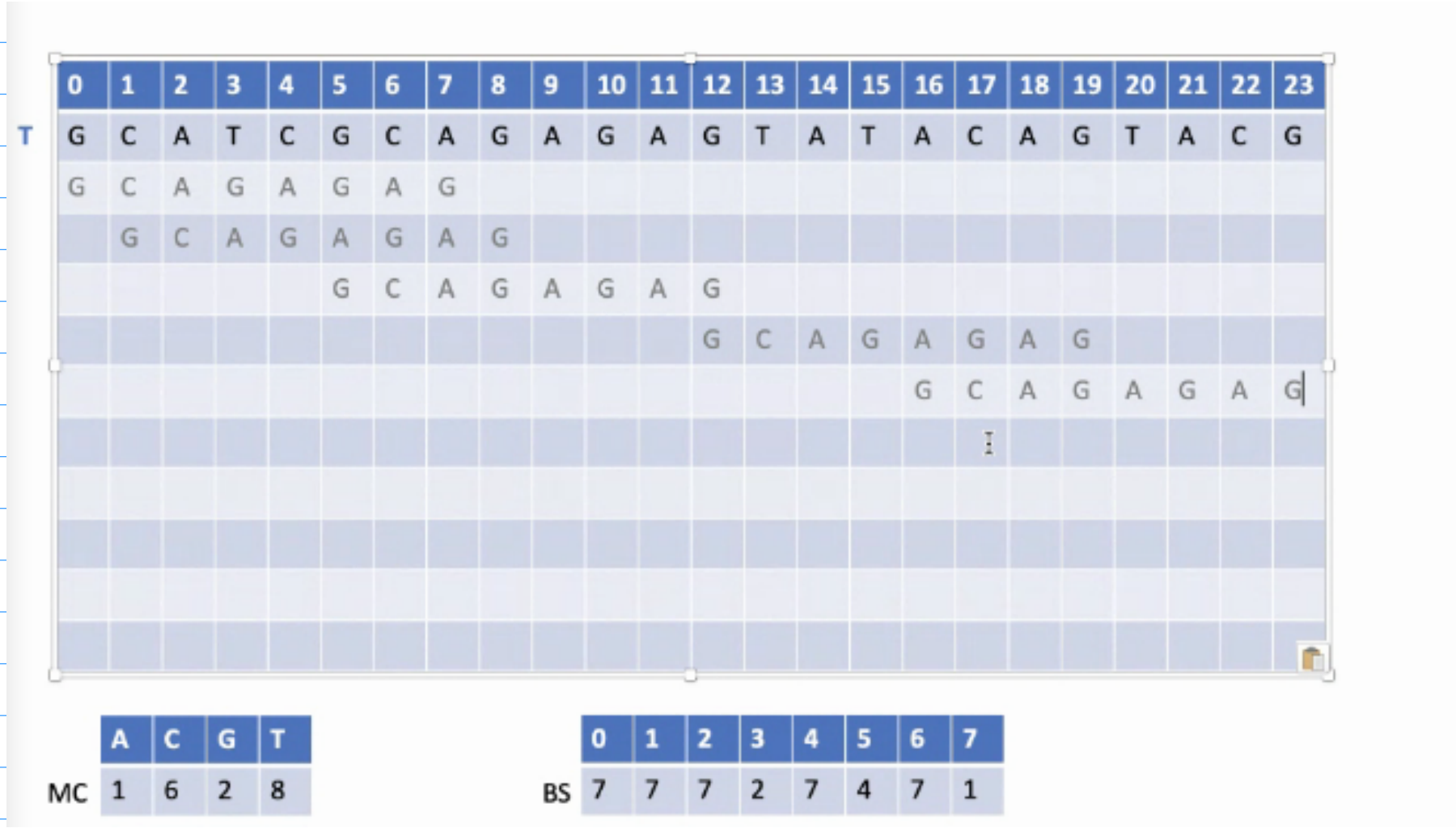
BS[5]=4 ; MC[C]+5-8+1=4 ⇒ décalage de 4

G C A G A G A G

T = G C A T C G C A G A G A G T A T A C A G T A C G

⇒ 17 comparaisons de caractères, là où la recherche naïve en fait 30

G C A G A G A G



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | G | C | A | T | C | G | C | A | G | A | G | A | G | T | A | T | A | C | A | G | T | A | C | G |
| | G | C | A | G | A | G | A | G | | | | | | | | | | | | | | | | |
| | | G | C | A | G | A | G | A | G | | | | | | | | | | | | | | | |
| | | | | | | G | C | A | G | A | G | A | G | | | | | | | | | | | |
| | | | | | | | | | | | | | G | C | A | G | A | G | A | G | | | | |
| | | | | | | | | | | | | | | | | | G | C | A | G | A | G | A | G |
| | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | | | | |

| | A | C | G | T |
|---|---|---|---|---|
| MC | 1 | 6 | 2 | 8 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| BS | 7 | 7 | 7 | 2 | 7 | 4 | 7 | 1 |

# Algorithme de Boyer-Moore - exercices

- Quelle est la complexité temporelle au mieux de la partie « Recherche de Motif » de Boyer-Moore ? Pour quelle forme des données ?
- Quelle est la complexité temporelle au pire de la partie « Recherche de Motif » de Boyer-Moore ? Pour quelle forme des données ?

# Comparatif Naïf/Boyer-Moore

- Réalisé par Ben Langmead (John Hopkins University, USA)
- Remarque : dans ce tableau, le motif M est noté... P (pour « Pattern »)

Simple Python implementations of naïve and Boyer-Moore:

| | Naïve matching | | Boyer-Moore | | |
|---|---|---|---|---|---|
| | # character comparisons | wall clock time | # character comparisons | wall clock time | |
| **P**: "tomorrow"<br>**T**: Shakespeare's complete works | 5,906,125 | 2.90 s | 785,855 | 1.54 s | 17 matches<br>$\|T\| = 5.59$ M |
| **P**: 50 nt string from Alu repeat*<br>**T**: Human reference (hg19) chromosome 1 | 307,013,905 | 137 s | 32,495,111 | 55 s | 336 matches<br>$\|T\| = 249$ M |

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

# Recherche Exacte de Motif – en résumé

- Recherche par fenêtre glissante :
  - algorithme naïf de complexité quadratique dans le pire des cas
  - algorithme de Boyer-Moore : au pire comme naïf, au mieux sous-linéaire
- En pratique, très rapide et très utilisé...
- D'autre algorithmes existent en recherche de motifs exacts (ex. KMP) et approchés (càd admettant des erreurs, basé par ex. sur la distance de Hamming ou la distance d'édition)
- La recherche de motifs dans le langage naturel se base souvent sur des expressions régulières (utilisation d'automates)

# Références

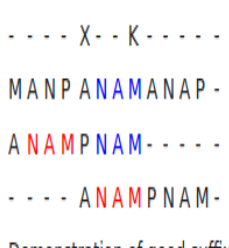
- Tout d'abord un grand remerciement à l'équipe de l'Université de Nantes (avec l'aimable autorisation de **Guillaume Fertin**) qui a fourni une grande partie du contenu de ce cours en licence CC-by
- Quelques références sur le Web :
  - Recherche de motifs dans des chaînes de symboles : KMP, Boyer Moore (Bad Character Heuristic), Boyer Moore (Good Suffix Heuristic)
  - Pour aller plus loin : les automates à état fini, ou la programmation d'algorithmes basés sur les expressions régulières

## The good suffix rule [ edit ]

### Description [ edit ]

The good suffix rule is markedly more complex in both concept and implementation than the bad character rule. It is the reason comparisons begin at the end of the pattern rather than the start, and is formally stated thus:[5]

> Suppose for a given alignment of **P** and **T**, a substring **t** of **T** matches a suffix of **P**, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy **t'** of **t** in **P** such that **t'** is not a suffix of **P** and the character to the left of **t'** in **P** differs from the character to the left of **t** in **P**. Shift **P** to the right so that substring **t'** in **P** aligns with substring **t** in **T**. If **t'** does not exist, then shift the left end of **P** past the left end of **t** in **T** by the least amount so that a prefix of the shifted pattern matches a suffix of **t** in **T**. If no such shift is possible, then shift **P** by **n** places to the right. If an occurrence of **P** is found, then shift **P** by the least amount so that a *proper* prefix of the shifted **P** matches a suffix of the occurrence of **P** in **T**. If no such shift is possible, then shift **P** by **n** places, that is, shift **P** past **t**.



Demonstration of good suffix rule with pattern **ANAMPNAM**.

### Preprocessing [ edit ]

The good suffix rule requires two tables: one for use in the general case, and another for use when either the general case returns no meaningful result or a match occurs. These tables will be designated $L$ and $H$ respectively. Their definitions are as follows:[5]

> For each $i$, $L[i]$ is the largest position less than $n$ such that string $P[i..n]$ matches a suffix of $P[1..L[i]]$ and such that the character preceding that suffix is not equal to $P[i-1]$. $L[i]$ is defined to be zero if there is no position satisfying the condition.

> Let $H[i]$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of $P$, if one exists. If none exists, let $H[i]$ be zero.

Both of these tables are constructible in $O(n)$ time and use $O(n)$ space. The alignment shift for index $i$ in $P$ is given by $n - L[i]$ or $n - H[i]$. $H$ should only be used if $L[i]$ is zero or a match has been found.