

DIU EIL 2019 / Bloc 5

Séance 4 – Recherche Textuelle



Sommaire

Recherche Exacte de Motif dans un Texte

- Recherche Naïve
- Algorithme de Boyer-Moore

Alignement de Séquences

- Distance de Hamming
- Distance d'édition

Programmation Dynamique – Un détour

Recherche Exacte de Motif

Qu'est-ce qu'un « Texte » ?

- Texte = suite de **caractères**, pris dans un **alphabet** donné
- Œuvre littéraire, page Web, tout document écrit en langue dite « naturelle »
 - « Scarlett O'Hara n'était pas d'une beauté classique, mais les hommes ne s'en apercevaient guère quand, à l'exemple des jumeaux Tarleton, ils étaient captifs de son charme. » (Margaret Mitchell - « Autant en emporte le vent »)
 - Alphabet = {A,B...Z,a,b...Z,0,1...9, etc.¹}
- Séquence biologique comme:
 - Séquence d'ADN :
GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG
Alphabet = {A,C,G,T}
 - Séquence protéique :
EGDRNTAYFHKLINFRSSGAVQGILIDGVVHQPDLVKKAVFNFFAERFTEQN
Alphabet = {A,R,N,D,C,Q,E,J,H,I,L,K,M,F,P,O,U,S,T,W,Y,V}

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

Recherche (Exacte) de Motifs

- **Motif** : un texte aussi, mais de « petite » taille
- Motif appelé **M**, de longueur **m**
- **Texte T** de longueur **n**
- Recherche d'un motif M dans un texte T :
 - recherche de **toutes les occurrences** de M dans T
 - recherche exacte \Rightarrow on n'autorise pas d'erreur
- Exemple : recherche de M dans T pour

M = GCAG et

T = GGCAGCCGAACCGCAGGCAGCAC

M apparaît dans T aux positions 1, 12 et 15

Recherche de Motifs

- Applications fréquentes :
 - Recherche d'un mot dans un document (ctrl-f)
 - Recherche d'une sous-séquence d'intérêt dans une séquence biologique
- \Rightarrow on considérera dans la suite que $m \ll n$
- Exemples :
 - chercher $M = \text{algorithm}$ dans le programme NSI de Terminale
 - $n = 201.097$, $m = 10$; 9 occurrences
 - chercher $M = \text{TTGACA}$ (promoteur de gène) dans le chromosome 1 de l'humain
 - $n = 249.250.621$, $m = 6$; plus de 1000 occurrences

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
- Exemple

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 A C G

A G C A

A G C A

A G C A

Précisions et Vocabulaire

Dans toute la suite :

- T et M 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 $M[0..j]$)... et on l'a déjà fait
- **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 $T[i+j]$?**)
 - $M[j]=T[i+j] \Rightarrow$ **match**
 - $M[j]\neq T[i+j] \Rightarrow$ **mismatch**
- Alphabet Σ , de taille σ (ex : $\Sigma=\{A,C,G,T\}$, de taille $\sigma=4$)

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...

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

AGCA

AGCA

AGCA

etc.

Recherche naïve – Exercices (20')

- 1) Quelle est la dernière valeur de i à tester ?
- 2) Écrire l'algorithme de Recherche Naïve

Recherche naïve – Réponses

Quelle est la dernière valeur de i à tester ?



$$n=22, m=4 \Rightarrow i = 18 = 22-4$$

En général, $i=n-m$

Dernière possibilité d'apparition de M dans T

Recherche naïve – Réponses

Écrire l'algorithme de Recherche Naïve

```
def recherche_naive(T, M):  
  
    n = len(T)  
  
    m = len(M)  
  
    for i in range(n-m+1):  
  
        j = 0  
  
        while j < m and T[i+j] == M[j]:  
  
            j +=1  
  
        if j == m:  
  
            print("Motif trouvé à la position ",i)
```

Recherche naïve – Exercices (30')

- 1) Quelle est la complexité temporelle au mieux de la Recherche Naïve ? Pour quelle forme des données ?
- 2) Quelle est la complexité temporelle au pire de la Recherche Naïve ? Pour quelle forme des données ?
- 3) 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.

Recherche naïve – Réponses

- 1) Quelle est la complexité temporelle au mieux de la Recherche Naïve ? Pour quelle forme des données ?

On effectue le moins d'opérations lorsque, pour tout $0 \leq i \leq n-m+1$, $T[i] \neq M[0]$ (mismatch dès le premier caractère testé)

Complexité en $\Omega(n-m)$

Forme des données : $T=AAA..AAA$ et $M=BB..B$

- 2) Quelle est la complexité temporelle au pire de la Recherche Naïve ? Pour quelle forme des données ?

On effectue le plus d'opérations lorsque, pour tous $0 \leq i \leq n-m+1$ et $0 \leq j \leq m-1$, $T[i+j]=M[j]$ (M est présent à chaque position de T)

Complexité en $O((n-m)*m)$

Forme des données : $T=AAA..AAA$ et $M=AA..A$

Recherche naïve – Réponses

Supposons que le motif M ne contient pas deux fois la même lettre.

Exemple :

T=ACTGTTACGTAGAACCTT et M=ACGT

ACGT → ?

T=ACTGTTACGTAGAACCTT et M=ACGT

ACGT → ?

Idée : décaler M du nombre de positions qui « matchent »
(décaler de 1 si $T[i] \neq M[0]$)

Recherche naïve – Réponses

Écrire un algorithme de recherche exacte de motif qui exploite cette information.

```
def recherche_tous_différents(T,M):  
    n = len(T)  
    m = len(M)  
    i = 0  
    while i < n-m+1:  
        j = 0  
        while j < m and T[i+j] == M[j]:  
            j +=1  
        if j == m:  
            print("Motif trouvé à la position ",i)  
        i+=max(1,j) ## pour traiter le cas j=0 (⇒ décalage de 1)
```

Recherche naïve – Réponses

Discuter de ses complexités temporelles au mieux et au pire.

Pour l'analyse de la complexité, mieux vaut compter le nombre de comparaisons $M[j]=T[i+j]$ (idée que l'on gardera pour la suite)

On peut voir que pour tout $0 \leq i \leq n-m+1$, chaque $T[i]$ n'est consulté qu'une fois.

- Au mieux, les $m-1$ derniers caractères de $T[i]$ ne sont pas lus (car $M[0] \neq T[n-m]$) $\Rightarrow \Omega(n-m)$
- Au pire, ces $m-1$ derniers caractères sont tous lus (M apparaît en position $i=n-m$) $\Rightarrow O(n)$

Si on considère $m \ll n$, la complexité est en $\Theta(n)$

Retour sur la Recherche Naïve

- Au pire : $O((n-m)*m) \Rightarrow$ pas souhaitable (surtout pour des grandes valeurs de n , voir transparent 6)

Remarques :

- on peut montrer que si T et M sont choisis uniformément au hasard dans l'alphabet, alors la recherche naïve est en $\Theta(n-m)$
- mais ce n'est jamais vrai en pratique !
- Idée : la **comparaison $M[0..m-1]$ vs $T[i..i+m-1]$** donne des informations pouvant mener à un **décalage plus important** (que 1)
- Exemple :

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

M = A C T T A C G A

←→ A C T T A C G A

\Rightarrow décaler M de 4 positions dans T (répétition de « AC » dans M)

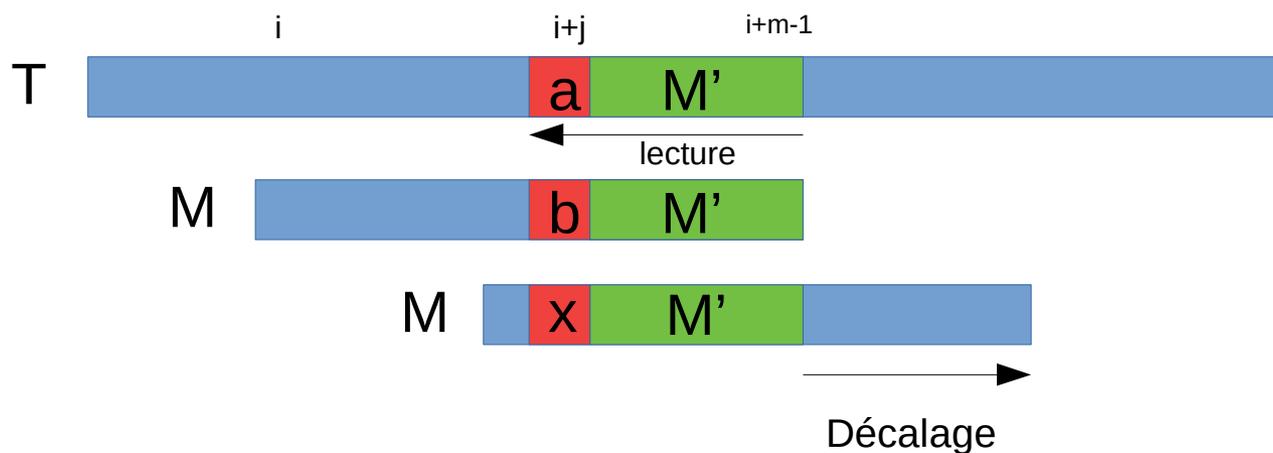
- Une solution possible : **l'Algorithme de Boyer-Moore**

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**

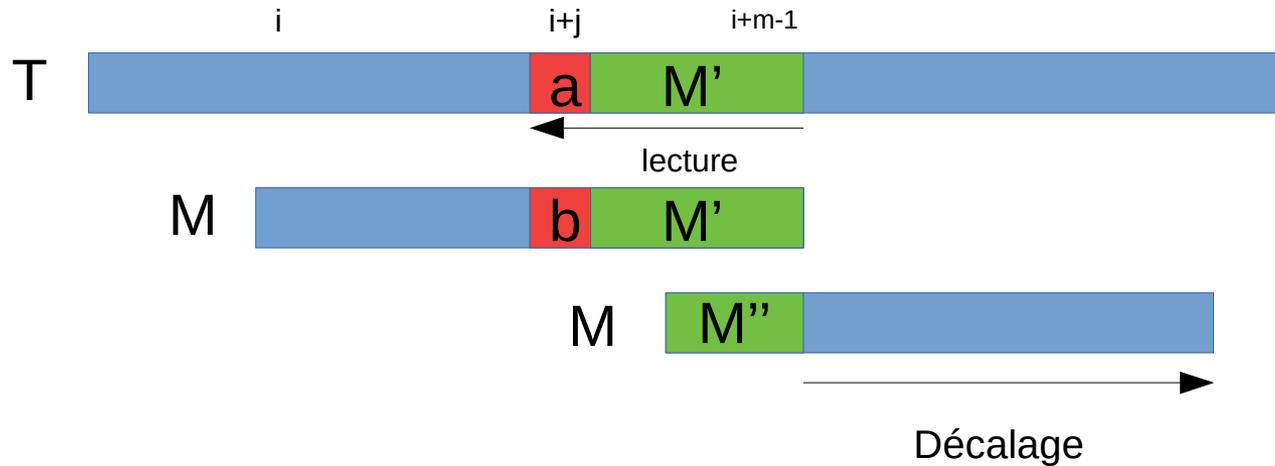
Règle du Bon Suffixe (1)

- S'il existe un $x.M'$ à gauche de $b.M'$ dans M , avec $x \neq b...$
- ...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



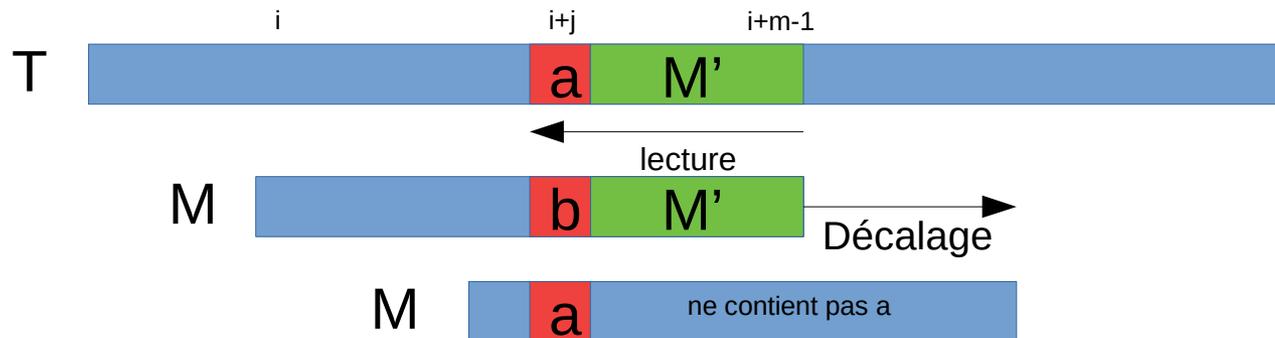
Règle du Bon Suffixe (2)

- 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



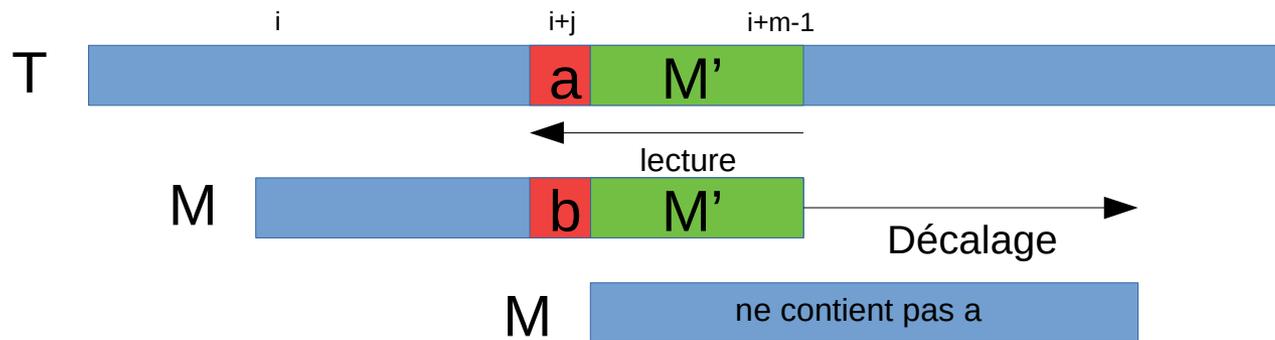
Règle du Mauvais Caractère (1)

- 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



Règle du Mauvais Caractère (2)

- Si $T[i+j]=a$ n'est pas dans M , la prochaine position à tester pour M est la position $i+j+1$
 - a, b : caractères
 - M' :suffixe de M



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
 - $MC[0..\sigma-1]$ (rappel: $\sigma =$ 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 (5')** : Donner le contenu de MC pour $M=GCAGAGAG$ avec $\Sigma=\{A,C,G,T\}$

Tableau « Mauvais Caractère » – Exercice

Donner le contenu de MC pour $M=GCAGAGAG$ avec $\Sigma=\{A,C,G,T\}$

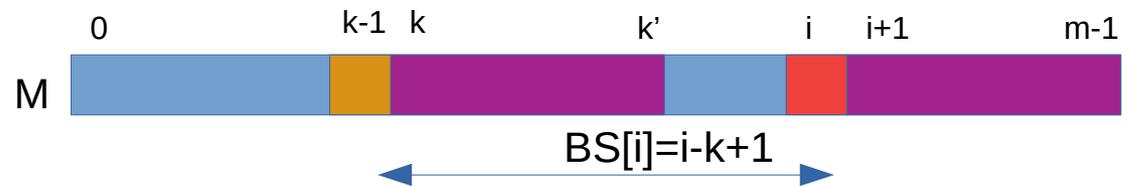
Caractère c	A	C	G	T
MC[c]	1	6	2	8

Mise en œuvre des règles BS et MC (2)

- **Tableau BS** : $BS[0..m-1]$

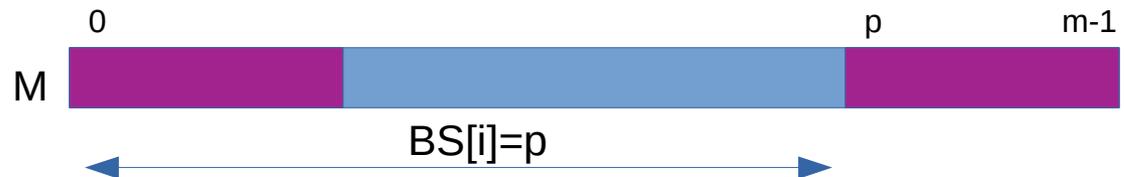
- $BS[i] = i - k + 1$, où k est la plus grande position dans M telle que :

- $M[k..k'] = M[i+1..m-1]$ (avec $k' = k + (m - i - 2)$)
- $M[k-1] \neq M[i]$



- 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 (10')** : Donner le contenu de BS pour $M = GCAGAGAG$

Tableau « Bon Suffixe » – Exercice

Donner le contenu de BS pour M=GCAGAGAG

i	0	1	2	3	4	5	6	7
M[i]	G	C	A	G	A	G	A	G
BS[i]	7	7	7	2	7	4	7	1

Algorithme/Complexité du Pré-traitement de M

On parle ici de 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 $O(m+\sigma)$**

Exercice (5') : discuter de la complexité **spatiale du pré-traitement de M**

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 ## Attention ! 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 à la position »,i)

 i ← i + BS[0] ## décalage du motif

 Sinon

 i ← max(BS[j],MC[T[i+j]]+j-m+1) ## décalage du motif

 FinSi

FinTantQue

Algorithme de Boyer-Moore – 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** $BS[7]=1$; $MC[A]+7-8+1=1 \Rightarrow$ 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** $BS[5]=4$; $MC[C]+5-8+1=4 \Rightarrow$ 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 \Rightarrow 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

$BS[5]=4$; $MC[C]+5-8+1=4 \Rightarrow$ 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**

\Rightarrow 17 comparaisons de caractères, là où la recherche naïve en fait 30 G C A G A G **A** **G**

Algorithme de Boyer-Moore – Exercices (10')

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

Algorithme de Boyer-Moore – Réponses

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

⇒ « peu » de comparaisons $M[j]$ vs $T[i+j]$ ⇒ plutôt règle « Mauvais Caractère »

⇒ « grands » décalages ⇒ caractère absent (décalage de m)

Forme des données : $T = AAA..AAA$ et $M = BB..BB$

A chaque fois :

1 comparaison $M[m-1]$ vs $T[i+m-1]$ suivie

d'un décalage de m positions

⇒ Complexité au mieux en $\Omega(n/m)$

Algorithme de Boyer-Moore – Réponses

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

Complexité au pire :

⇒ « beaucoup » de comparaisons $M[j]=T[i+j]$

⇒ le motif apparaît très fréquemment

⇒ des « petits » décalages

⇒ ni $MC[]$ ni $BS[]$ ne sont utiles

⇒ Même cas que dans la recherche naïve : une forme des données possible est : $T=AAA..AAA$ et $M=AA..AA$

Complexité en $O((n-m)*m)$

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" T: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s	17 matches $ T = 5.59 \text{ M}$
P: 50 nt string from Alu repeat* T: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s	336 matches $ T = 249 \text{ M}$

* GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

Recherche Exacte de Motif – En Résumé

- Recherche par fenêtre glissante
 - Algorithme naïf en $O((n-m)*m)$ au pire
 - Algorithme de Boyer-Moore :
 - Au pire, comme naïf
 - Au mieux, sous-linéaire $\Omega(n/m)$
 - En pratique, très rapide et très utilisé
 - Rem : amélioration par Galil (1979) $\Rightarrow O(n+m)$ au pire
- D'autres algorithmes existent (vaste littérature!)
- Variante « naturelle » : recherche approchée (à ϵ erreurs près)... pas si évident !

Exercice (non traité en séance) : écrire un algorithme qui calcule $MC[]$ et dont la complexité temporelle est en $O(m+\sigma)$.

Alignement de Séquences

Alignement de Séquences – Introduction

- Séquence ou texte ?

l'alignement a plus d'applications dans les séquences biologiques

⇒ mais texte ou séquence, c'est la même chose !

- Objectif : comparer deux textes/séquences T_1 et T_2
- Ex: $T_1 = \text{SUCCES}$, $T_2 = \text{ECHECS}$
- Décider de leur degré de similarité/dissimilarité
 - dissimilarité → **minimiser** une **distance** $d(T_1, T_2)$
 - similarité → **maximiser** un **score** (de ressemblance) $s(T_1, T_2)$
- Applications : si 2 séquences d'ADN représentant des gènes sont similaires, la fonction du gène est probablement la même
- Dans la suite : étude de deux distances : Distance de Hamming et Distance d'édition
- Ce qui nous permettra de présenter une (nouvelle?) catégorie d'algorithmes : les **algorithmes de Programmation Dynamique**

Distance de Hamming (1)

- Avant-propos :
 - T_1 de longueur n_1 ; T_2 de longueur n_2
 - $T_1 = T_1[0..n_1-1]$; $T_2 = T_2[0..n_2-1]$
 - $d(T_1, T_2)$ doit être une **distance** (au sens mathématique) :
 - $d(T_1, T_1) = 0$ pour tout T_1
 - $d(T_1, T_2) = d(T_2, T_1)$ pour tous T_1, T_2
 - $d(T_1, T_3) \leq d(T_1, T_2) + d(T_2, T_3)$ pour tous T_1, T_2, T_3
- Distance de Hamming :
 - Définie seulement **si $n_1 = n_2$**
 - Notée ici $dH(T_1, T_2)$
 - $dH(T_1, T_2) =$ nombre de positions i pour lesquelles $T_1[i] \neq T_2[i]$

Distance de Hamming (2)

- Exemple : $T_1 = \text{PLATINE}$, $T_2 = \text{DIAMANT}$

PLATINE
DIAMANT

- $d_H(\text{PLATINE}, \text{DIAMANT}) = 5$

Hamming – Algorithme et Discussion

Entrée : Textes T_1 et T_2

$dH \leftarrow 0$

Pour i de 0 à n_1-1 faire

 Si $T_1[i] \neq T_2[i]$ alors

$dH \leftarrow dH + 1$

 FinSi

FinPour

return dH

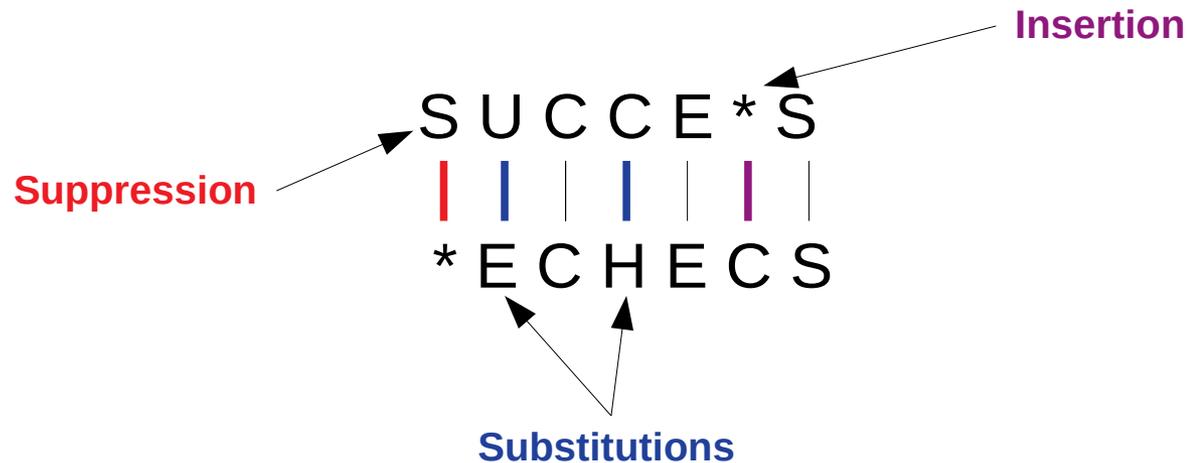
- Complexité : $\Theta(n_1)$
- Remarques : distance « contrainte » (exige que $n_1=n_2$) et peu informative
 - $dH(\text{TERMINALE}, \text{TEERMINAL})=7$
 - $dH(\text{EXEMPLES}, \text{EXERCICE})=dH(\text{EXEMPLES}, \text{ZZZZZLES})=5$
 - dH utile pour les petits alphabets, notamment binaires (ex : correction d'erreurs)

Distance d'édition (1)

- Définie pour **tous textes** T_1 et T_2
- Autorise insertion et suppression d'éléments
- Plus précisément, 3 opérations :
 - Insertion (d'un caractère c dans T_1)
 - Suppression (d'un caractère c dans T_1)
 - Substitution (d'un caractère c par un autre c' dans T_1)
- Coûts associés : respectivement, C_i , C_d (d pour « deletion »), C_s
- Distance : **minimiser la somme des coûts** entre T_1 et T_2 , notée ici **$dE(T_1, T_2)$**
- Rappel : $dE(T_1, T_2)$ doit être une **distance**, ce qui est vrai si $C_i = C_d$
- Remarque : aussi appelée **Distance de Levenshtein**

Distance d'édition (2)

- Le calcul de $dE(T_1, T_2)$ permet de déterminer un **alignement** entre T_1 et T_2
- Exemple (ici avec $C_i=C_d=C_s=1$)



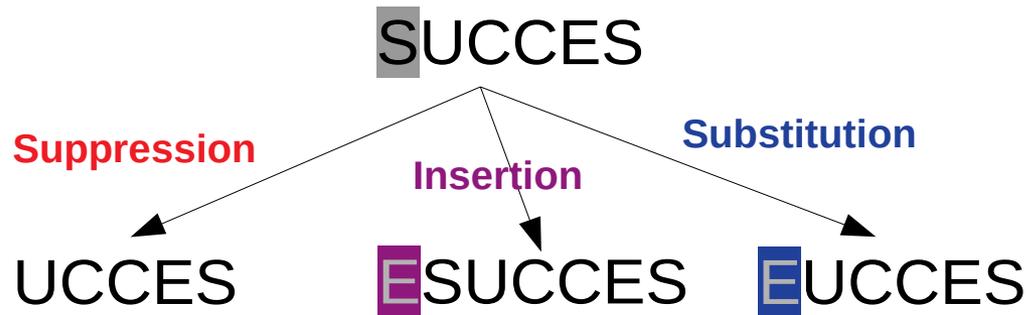
- $dE(\text{SUCCE}^*\text{S}, \text{*ECHECS})=4$
- Remarques :
 - il existe un autre alignement optimal. **Lequel ?**
 - si passer des SUCCE^{*}S aux ECHECS vous déplaît, il suffit de lire du bas vers le haut : la distance est (par définition) symétrique !

Distance d'édition – Algorithme

- Première (mauvaise) idée : tout tester

(rappel : cela s'appelle generate-and-test, ou analyse exhaustive, ou brute force)

Exemple : SUCCES vs ECHECS



- Pourquoi est-ce une mauvaise idée ?

Combien de cas à tester pour $n_1=2$? Pour $n_1=3$? En général ?

Distance d'édition – Algorithme

- Deuxième (bonne) idée :
 - Appelons $T_1[0..i]$ le préfixe de T_1 jusqu'à la position i (incluse)
 - Appelons $T_2[0..j]$ le préfixe de T_2 jusqu'à la position j (incluse)
- Peut-on calculer $dE(T_1[0..i], T_2[0..j])$ si on connaît $dE()$ pour des « bouts de texte » plus courts ?
- Réponse : oui !

Distance d'édition – Algorithme

- Argument : supposons avoir aligné $T_1[0..i]$ et $T_2[0..j]$ et regardons l'alignement le plus à droite. Quatre cas se présentent :
 - Insertion : $T_1[i]$ s'aligne sur « * »
 - ⇒ demande à avoir aligné $T_1[0..i-1]$ et $T_2[0..j]$ + coût C_d
 - Suppression : « * » s'aligne sur $T_2[j]$
 - ⇒ demande à avoir aligné $T_1[0..i]$ et $T_2[0..j-1]$ + coût C_i
 - Substitution : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] \neq T_2[j]$
 - ⇒ demande à avoir aligné $T_1[0..i-1]$ et $T_2[0..j-1]$ + coût C_s
 - « Match » : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] = T_2[j]$
 - ⇒ demande à avoir aligné $T_1[0..i-1]$ et $T_2[0..j-1]$ sans surcoût

Distance d'édition – Algorithme

- Comme on cherche à **minimiser** la distance, on a la relation de récurrence suivante :
- pour tous $0 \leq i \leq n_1 - 1$ et $0 \leq j \leq n_2 - 1$

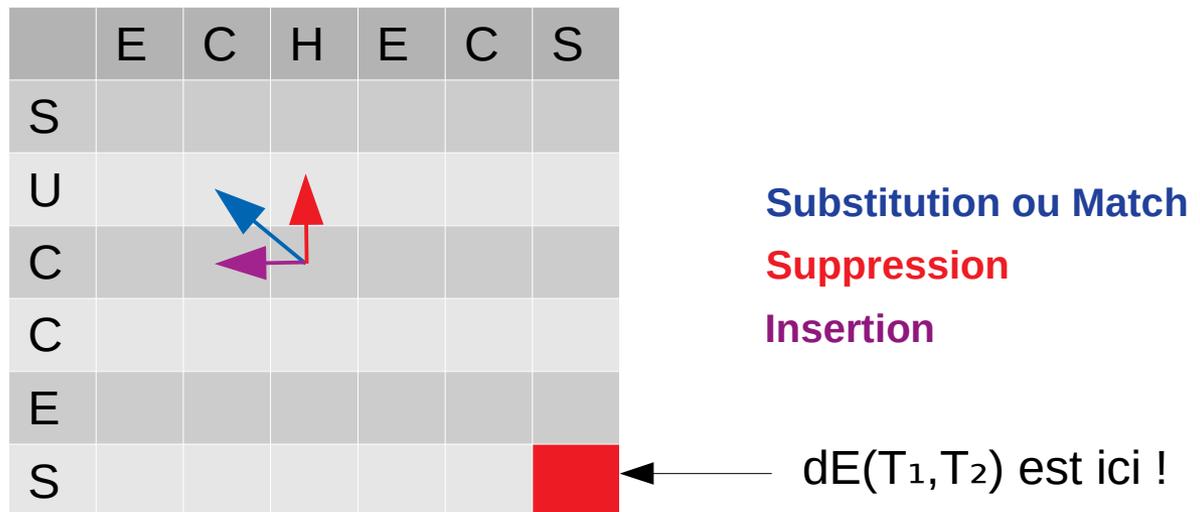
$$dE(T_1[0..i], T_2[0..j]) = \min \begin{cases} dE(T_1[0..i-1], T_2[0..j]) + C_d \\ dE(T_1[0..i], T_2[0..j-1]) + C_i \\ dE(T_1[0..i-1], T_2[0..j-1]) + \text{sub}(T_1[i], T_2[j]) \end{cases}$$

$$\text{où } \text{sub}(a, b) = \begin{cases} 0 \text{ si } a = b \text{ (Match)} \\ C_s \text{ si } a \neq b \text{ (Substitution)} \end{cases}$$

- Problème : **où s'arrête-t-on** ? (dit autrement, que veut dire par exemple $T_1[0..-1]$?) \Rightarrow on en reparle dans 2 transparents

Distance d'édition – Algorithme

- Il suffit donc de calculer tous les $dE(T[0..i], T[0..j])$ en utilisant cette formule de récurrence
- Algorithmiquement :
 - Tableau $D[i,j]$, pour tous $0 \leq i \leq n_1-1$ et $0 \leq j \leq n_2-1$
 - Rempli dans un ordre « intelligent », suivant la formule (voir tsp suivant)
 - **La solution $dE(T_1, T_2)$ se trouve en $D[n_1-1, n_2-1]$**



Distance d'édition – Remplir le tableau D[]

	-1	0	1	2	3	4	5	
	ε	E	C	H	E	C	S	
-1	ε	0	1	2	3	4	5	6
0	S	1	→					▶
1	U	2	→					▶
2	C	3	→					▶
3	C	4						
4	E	5						
5	S	6	→					▶

i : indice des lignes
 j : indice des colonnes
 ε = mot vide

- On va supposer que D[] possède une ligne et une colonne d'indice -1
- C'est plus simple :
 - $D[-1,-1]=0$
 - $D[-1,j]=(j+1)*C_i$ pour tout $1 \leq i \leq n_2-1$; $D[i,-1]=(i+1)*C_d$ pour tout $1 \leq i \leq n_1-1$
- Sur l'exemple, $C_i=C_d=1$
- Puis remplissage de D ligne par ligne

Distance d'édition – Algorithme (enfin!)

```
Calcul_Distance_d_edition(T1, T2 : textes)
```

```
D[-1,-1]=0 ## dE(ε,ε)=0
```

```
Pour i de 0 à n1-1 faire
```

```
    D[i,-1]=D[i-1,-1]+Cd ## Colonne -1
```

```
FinPour
```

```
Pour j de 0 à n2-1 faire
```

```
    D[-1,j]=D[-1,j-1]+Ci ## Ligne -1
```

```
FinPour
```

```
Pour i de 0 à n1-1 faire
```

```
    Pour j de 0 à n2-1 faire
```

```
        D[i,j]=min(D[i-1,j]+Cd ; D[i,j-1]+Ci ; D[i,j]+sub(T1[i],T2[j]))
```

```
    FinPour
```

```
FinPour
```

```
Renvoyer D[n1-1,n2-1]
```

$$\text{sub}(T_1[i], T_2[j]) = \begin{cases} 0 & \text{si } T_1[i] = T_2[j] \\ C_s & \text{si } T_1[i] \neq T_2[j] \end{cases}$$



Programmation Dynamique

- Cet algorithme est un algorithme de **Programmation Dynamique**
- Technique due à Richard Bellman dans les années 1950
- Idée : Déterminer un résultat sur la base de calculs précédents
 - ⇒ formule de récurrence (avec un nombre limité de cas, ici 3 (ou 4 si on distingue Substitution et Match))
 - ⇒ stockage des calculs intermédiaires dans une « table de programmation dynamique » (ici, le tableau $D[]$)
- Une catégorie de plus... après les algorithmes **gloutons** et ceux de type **diviser-pour-régner**

Distance d'édition – Exercices (5')

- 1) Quelle est la complexité spatiale de cet algorithme ?
- 2) Quelle est la complexité temporelle de cet algorithme ?

Distance d'édition – Réponses

1) Quelle est la complexité spatiale de cet algorithme ?

$D[]$ = tableau de valeurs à n_1+1 lignes et n_2+1 colonnes

⇒ complexité spatiale en $\Theta(n_1*n_2)$

2) Quelle est la complexité temporelle de cet algorithme ?

Dans l'algorithme, c'est la double boucle « Pour i/Pour j » qui est la plus coûteuse. Il y a un nombre constant d'opérations pour chaque couple de valeurs i, j

⇒ complexité temporelle en $\Theta(n_1*n_2)$

Remarque : boucles « Pour », donc mieux = pire

Distance d'édition – Exercice (15')

Remplir le tableau $D[]$ ci-dessous, en supposant $C_i=C_d=C_s=1$

	-1	0	1	2	3	4	5
	ϵ	E	C	H	E	C	S
-1	ϵ						
0	S						
1	U						
2	C						
3	C						
4	E						
5	S						

Distance d'édition – Réponse

Remplir le tableau $D[]$ ci-dessous, en supposant $C_i=C_d=C_s=1$

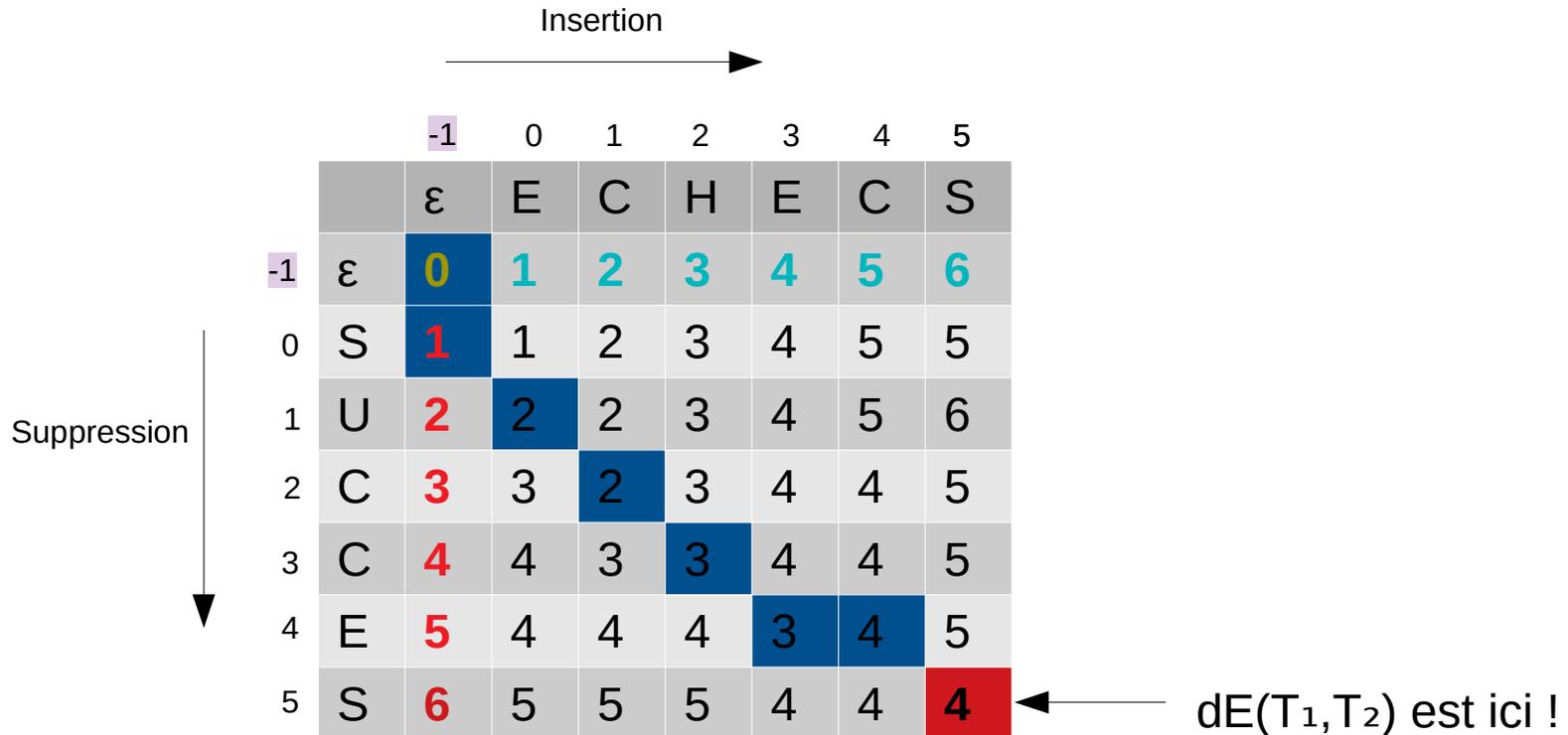
	-1	0	1	2	3	4	5	
	ϵ	E	C	H	E	C	S	
-1	ϵ	0	1	2	3	4	5	6
0	S	1	1	2	3	4	5	5
1	U	2	2	2	3	4	5	6
2	C	3	3	2	3	4	4	5
3	C	4	4	3	3	4	4	5
4	E	5	4	4	4	3	4	5
5	S	6	5	5	5	4	4	4

← $dE(T_1, T_2)$ est ici !

Distance d'édition et Alignement

- Le nombre $dE(T_1, T_2)$ peut donc se calculer en temps $\Theta(n_1 * n_2)$
- Mais comment retrouver la façon de passer de T_1 à T_2 (c'est-à-dire l'**alignement**) ?
- Remarque : il peut y avoir **plusieurs** alignements
 - En veut-on un ?
 - Les veut-on tous ?
- Contentons-nous d'**un seul alignement**
- **Idée** : « remonter » depuis $D[n_1-1, n_2-1]$ vers $D[-1, -1]$ en considérant le « meilleur » des 3 « voisins »

Distance d'édition et Alignement



Question : où se trouve le 2^e alignement induisant une distance de 4 ?

Exercice (non traité en séance):

- écrire un algorithme qui renvoie un alignement (c'est-à-dire les textes T_1 et T_2 , possiblement agrémentés de « * »)
- discuter sa complexité en temps

Programmation Dynamique – Un détour

Prog. Dynamique pour le Rendu de Monnaie

- Rendu de monnaie (rappel, cf. Bloc 2)

Obtenir un certain entier N en additionnant un **minimum** d'entiers pris (avec répétition possible) parmi un ensemble fini $P = \{p_1, \dots, p_k\}$

- N : la monnaie à rendre
- p_1, p_2, \dots, p_k : les valeurs faciales des pièces
- Ici, on suppose que:
 - $p_1 = 1$ (permet d'éviter le cas « impossible d'obtenir la valeur N », puisque $N * p_1 = N$)
 - $p_1 < p_2 < p_3 < \dots < p_k$

Prog. Dynamique pour le Rendu de Monnaie

- Idée : calculer $R[S,i]$, où :
 - S est un entier entre 0 et N
 - i est un entier entre 0 et k
 - $R[S,i]$ est le **nombre minimum** de pièces nécessaire pour **obtenir S** en utilisant uniquement les **pièces de valeurs p_1, p_2, \dots, p_i**
- Intérêt : $R[S,i]$ peut se calculer « par récurrence » :
 - Si la pièce de valeur p_i est utilisée, alors **$R[S,i]=R[S-p_i,i]+1$**
(il reste à atteindre **$S-p_i$** , et on a toujours droit à **p_1, p_2, \dots, p_i**)
 - Sinon, **$R[S,i]=R[S,i-1]$** (on n'utilise que **p_1, p_2, \dots, p_{i-1}** pour atteindre **S**)

Prog. Dynamique pour le Rendu de Monnaie

- Donc, pour tous $0 \leq S \leq N$ et $1 \leq i \leq k$,

$$R[S,i] = \min \begin{cases} R[S-p_i,i] + 1 & \text{si } S-p_i \geq 0 \\ R[S,i-1] & \text{si } i \geq 1 \end{cases}$$

- Cas de base (premières colonne et ligne de $R[\]$) :
 - Pour tout $0 \leq i \leq k$, $R[0,i] = 0$ (pas besoin de pièces pour faire 0)
 - Pour tout $1 \leq S \leq N$, $R[S,0] = \infty$ ($S \geq 1$ inatteignable sans pièce)
- Solution (nb min. de pièces pour atteindre N): dans la case $R[N,k]$

Prog. Dyn. et Rendu de Monnaie – Exercices

(Non traités en séance)

- 1) Écrire l'algorithme qui permet de calculer $R[N,k]$
- 2) Indiquer ses complexités spatiales et temporelles
- 3) Écrire l'algorithme qui, partant de $R[]$, retrouve le nombre de pièces de chaque valeur à utiliser pour obtenir N