Techniques algorithmique pour la comparaison de séquences biologiques

Module MADG

Eric Tannier, Master 1 Bioinfo Univ Lyon 1, 2022-2023
Eric.Tannier@univ-lyon1.fr

Exam à la fin

Support (ce document) disponible à partir de ma page internet

Buts de la comparaison de séquences:

En général, tout ce qui peut concerner la comparaison de textes, ou plus généralement de séquences linéaires
- Fouille de données (de grep à Google)
- Comparaison de fichiers (diff)
- Linguistique comparée
- Études littéraires
- Analyse de séquences musicales
- Détection de plagiat

En bioinformatique, comparaisons de séquences protéiques ou nucléiques pour la détection de l'homologie
- prédiction de fonction
- phylogénie, évolution
- biodiversité
- annotation
- assemblage

Activité quotidienne de milliers de bioinformaticiens

- utilisation de méthodes et logiciels standards

- domaine de recherche en informatique toujours actif, toujours demandeur de nouvelles fonctionnalités et performances, publications, conférences…

- phase cruciale des méthodes bioinformatiques, en termes de rapidité de calcul, consommation de ressources en temps, en électricité, en mémoire

Spécificités bioinformatiques de la comparaison de séquences

- les séquences sont très grandes, très nombreuses, fixes, sur des alphabets petits
- on recherche ou compare des séquences qui se ressemblent mais peuvent différer jusqu'à extinction de la ressemblance
- le processus de différenciation des séquences est issu de mécanismes biologiques

Les méthodes, générales par certains aspects, sont adaptées à ces spécificités

Plan du cours

I/ Recherche de séquences exactes

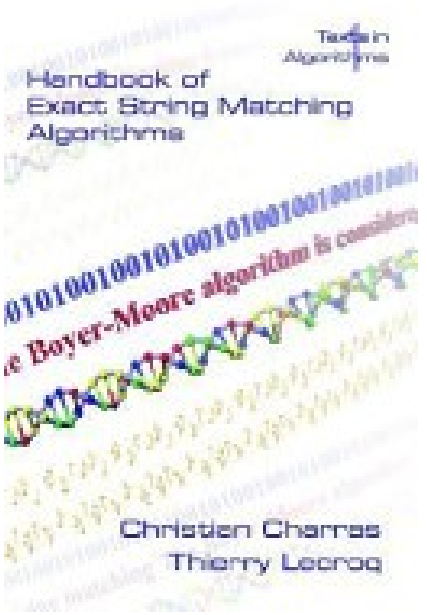
II/ Recherche de séquences approchées, alignement

III/ Méthodes et logiciels actuels

IV/ Limites et effets rebond

Recherches de mots exacts dans un texte

C'est un sujet classique en informatique, qui répond aux besoins de nombreuses applications. Il y a toujours aujourd'hui des articles, livres, des conférences, des sites internet sur le sujet.



Par exemple,

http://monge.univ-mlv.fr/~lecroq/string/

Recense plus de 85 algorithmes différents pour cette même tâche, les plus célèbres datant des années 1970, mais plus de la moitié ont moins de 10 ans.

Définitions

A alphabet
a élément de A caractère, lettre
S séquence de lettres
|S|=n taille d'une séquence de lettres
M mot, séquence courte
T texte, séquence longue

S[i] caractère de la ième position (départ à 1)
S[i:j] sous-chaîne entre les positions i et j
S[1:i] préfixe
S[j:n] suffixe

Définition du problème :
M un mot, T un texte, M est-il une sous-chaîne de T ? Déterminer toutes les cous-chaînes de T égales à M.

Exemple :
M=CGATGCGTAC

T=
ACGTAGTCAGCTAGCTGACTAGCTAGCTGATCGACTGAGTCAGCGAGTCA
GCTAGCTGACTGACTGACTGACTGACTGAGACTCTGACTGACTGACTGAG
CTGGCTGACTGGATCGTAGCAGTCGACGATGCGTACGTAGCTAGCTGTGT
CTAGCAGAAGCGAACGCTGAGCTGTCGCTGGACGAGCGCTTGACGAGCAT
GACGTACTA

Le mot est-il dans le texte?

Définition du problème :
M un mot, T un texte, M est-il une sous-chaîne de T ? Déterminer toutes les cous-chaînes de T égales à M.

Exemple :
M=CGATGCGTAC

T=
ACGTAGTCAGCTAGCTGACTAGCTAGCTGATCGACTGAGTCAGCGAGTCA
GCTAGCTGACTGACTGACTGACTGACTGAGACTCTGACTGACTGACTGAG
CTGGCTGACTGGATCGTAGCAGTCGACGATGCGTACGTAGCTAGCTGTGT
CTAGCAGAAGCGAACGCTGAGCTGTCGCTGGACGAGCGCTTGACGAGCAT
GACGTACTA

Occurrence i = 127

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M|+1)
    j ← 1
    Tant que j ≤ |M| et que T[i+j-1] = M[j]
        j ← j + 1
    Si j = |M|+1 alors afficher i

En Python

```
for i in range(len(T)-len(M)):
   j = 0
   while j < len(M) and T[i+j] == M[j]:
      j = j + 1
  if j == len(M):
     print i
```

Comment évaluer la performance de cet algorithme?

Calculer le nombre d'opérations effectuées pour connaître un ordre de grandeur du temps mis par une machine pour l'exécuter.

Ce nombre dépend
- de la taille de T
- de la taille de M
- du langage de programmation
- du compilateur ou de l'interpréteur
- de l'architecture de l'ordinateur

Le nombre précis est impossible à calculer. Il faut une approximation, un ordre de grandeur qui ne dépende que
- de la taille de T
- de la taille de M

Notations pour le calcul de la complexité des algorithmes :

f et g sont deux fonctions (de la taille des données)
f = O(g) s'il existe un nombre k tel que quel que soit x (comportement asymptotique),

$$f(x) <= k * g(x)$$

ce qui veut dire que f ne croît pas beaucoup plus vite que g

Propriétés :
O(k*f) = O(f) (k est une constante et f une fonction)
O(f+g) = O(f) + O(g) (f et g deux fonctions)
O(f*g) = O(f) * O(g) (f et g deux fonctions)

1 est utilisé pour noter une fonction constante qui vaut 1 partout

Par exemple, O(1) veut dire une fonction qui est bornée par une constante. O(n) est une fonction bornée par une constante fois une variable n (taille des données)

Notations pour le calcul de la complexité des algorithmes :

Petits exemples :

i = 3

Notations pour le calcul de la complexité des algorithmes :

Petits exemples :

i = 3 $\rightarrow O(1)$

Notations pour le calcul de la complexité des algorithmes :

Petits exemples :

```
pour i de 1 à n :
    écrire i
```

Notations pour le calcul de la complexité des algorithmes :

Petits exemples :

pour i de 1 à n :
    écrire i → O(n)

Notations pour le calcul de la complexité des algorithmes :

Petits exemples :

La notation permet de simplifier les expressions :

$O(2n) \Rightarrow O(n)$
$O(n^2 + n) \Rightarrow O(n^2)$

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M+1|)

    j ← 1
    Tant que j ≤ |M| et que T[i+j] = M[j]
        j ← j + 1
    Si j = |M|+1 alors afficher i

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M|+1)

j ← 1	O(1)
Tant que j ≤ |M| et que T[i+j] = M[j]	O(1)
j ← j + 1	O(1)
Si j = |M|+1 alors afficher i	O(1)

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M|+1)

j ← 1	O(1)	
Tant que j ≤ |M| et que T[i+j] = M[j]	O(1)	O(|M|)
j ← j + 1	O(1)	
Si j = |M|+1 alors afficher i	O(1)	

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M|+1)

j ← 1	O(1)	
Tant que j ≤ |M| et que T[i+j] = M[j]	O(1)	O(|M|)
j ← j + 1	O(1)	
Si j = |M|+1 alors afficher i	O(1)	

O(|M|*|T|)

Algorithme :

Pour un indice i parcourant le texte T (de 1 à |T|-|M|+1)

j ← 1	O(1)	
Tant que j ≤ |M| et que T[i+j] = M[j]	O(1)	
j ← j + 1	O(1)	O(|M|)
Si j = |M|+1 alors afficher i	O(1)	
		O(|M|*|T|)

Notation : n = |T|, m = |M|

O(n*m)

Combien de temps pour rechercher les similarités entre deux génomes complets de mammifères avec cet algorithme ?

1/ Découper le premier génome en morceaux de taille 50
2/ Rechercher chaque morceau dans le deuxième génome

Combien de temps pour rechercher les similarités entre deux génomes complets de mammifères avec cet algorithme ?

1/ Découper le premier génome en morceaux de taille 50
2/ Rechercher chaque morceau dans le deuxième génome

Il y a 3.10^9 / 50 = 6 millions de mots à rechercher.
Chaque mot est recherché en environ 50 * 3.10^9 opérations

En tout, il faut de l'ordre de 9.10^{18} opérations.

En combien de temps un ordinateur faisant 10^{12} opérations par seconde pourra-t-il finir le calcul ?

Cet algorithme est juste mais impraticable si on veut traiter des données génomiques

Solutions
→ améliorer la puissance des ordinateurs
→ diminuer la taille des données
→ améliorer l'algorithme de recherche

Idée de la recherche dans un **dictionnaire**
Quelle est la complexité de la recherche d'un mot dans un dictionnaire? De la construction d'un dictionnaire si on a la liste des mots?

Exemple de question posée une autre année à un examen

1. Quelles sont les contraintes sous lesquelles on conçoit habituellement des algorithmes pour la bioinformatique ?

- ¬ Le réalisme biologique des processus modélisés
- ¬ La complexité des problèmes algorithmiques
- ¬ Les capacités des ordinateurs
- ¬ Les ressources naturelles
- ¬ Le carbone rejeté dans l'atmosphère
- ¬ Le temps d'une vie humaine
- ¬ La taille et la quantité des données biologiques disponibles

Exercice ; Supposez que vous avez accès à un dictionnaire T (une liste de mots triés par ordre alphabétique), que vous voulez vérifier si un mot M est dedans, et que vous disposiez seulement d'une fonction de comparaison entre mots "<=". Ecrivez l'algorithme et calculez sa complexité.

Algorithme de recherche dans un dictionnaire

début = 1, fin = |T|
Tant que fin-début > 1
    m = int((début+fin)/2)
    Si le mot T[m] < M (T[m] est avant M par ordre alphabétique)
        début = m
    Si le mot T[m] > M:
        fin = m

Algorithme de recherche dans un dictionnaire

début = 1, fin = |T|
Tant que fin-début > 1
    m = int((début+fin)/2)
    Si le mot T[m] < M (T[m] est avant M par ordre alphabétique)
        début = m
    Si le mot T[m] > M:
        fin = m

Combien de mots examine-t-on?
C(n) = 1 + C(n/2)

Algorithme de recherche dans un dictionnaire

début = 1, fin = |T|
Tant que fin-début > 1
    m = int((début+fin)/2)
    Si le mot T[m] < M (T[m] est avant M par ordre alphabétique)
        début = m
    Si le mot T[m] > M:
        fin = m

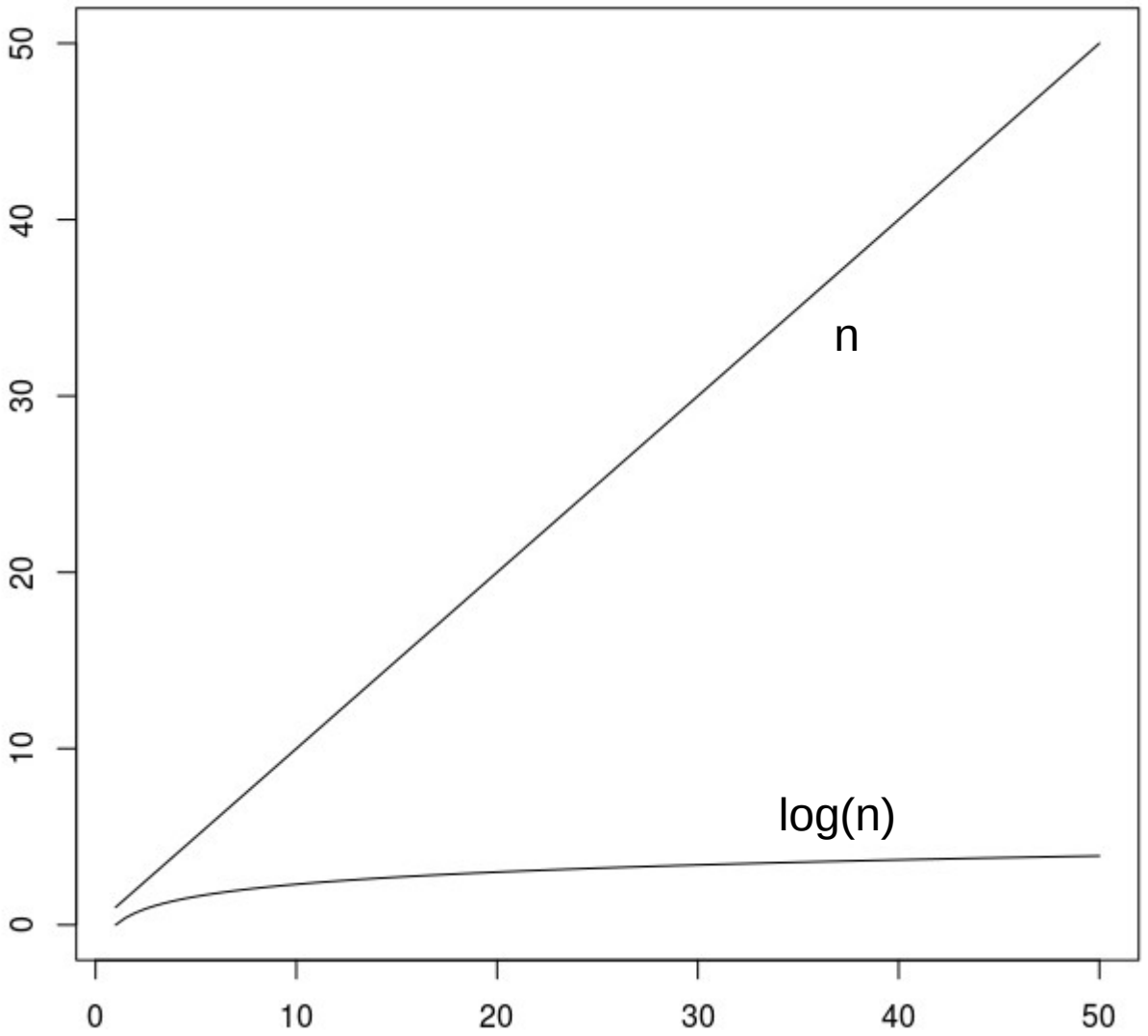
Combien de mots examine-t-on?
C(n) = 1 + C(n/2)
C(n) = 2 + C(n/4)
C(n) = 3 + C(n/8)
…
C(n) = k + C(n/2^k)
...

Algorithme de recherche dans un dictionnaire (recherche dichotomique)

début = 1, fin = |T|
Tant que fin-début > 1
    milieu = int((début+fin)/2)
    Si le mot T[milieu] < M (T[milieu] est avant M par ordre alphabétique)
        début = milieu + 1
    Si le mot T[milieu] > M:
        fin = milieu – 1
    Si le mot T[milieu] == M :
        afficher milieu

Combien de mots examine-t-on?
$C(n) = \log_2(n)$

Complexité de l'algorithme = coût d'une comparaison * $\log_2(n)$



La croissance des fonctions n ou log(n) montrent pourquoi la recherche dans un dictionnaire est possible

Dictionnaires de séquences génomiques

Il faut stocker tous les mots d’un texte de façon organisée

Combien y a-t-il de mots dans un texte de taille n?

Dictionnaires de séquences génomiques

Il faut stocker tous les mots d'un texte de façon organisée

Combien y a-t-il de mots dans un texte de taille n?

n*(n+1)/2

Mais on n'a pas besoin de stocker tous les mots, par exemple s'il y a GAC et GACC, le fait qu'il y ait GACC nous renseigne sur la présence de GAC.

On peut ne stocker que les suffixes de T

Combien y en a-t-il?

Dictionnaires de séquences génomiques

Il faut stocker tous les mots d'un texte de façon organisée

Combien y a-t-il de mots dans un texte de taille n?

n*(n+1)/2

Mais on n'a pas besoin de stocker tous les mots, par exemple s'il y a GAC et GACC, le fait qu'il y ait GACC nous renseigne sur la présence de GAC.

On peut ne stocker que les suffixes de T

Combien y en a-t-il?

n

Dictionnaires de séquences génomiques

Par exemple, si T = `CACGTACGTACTA`,
On stocke la liste

```
CACGTACGTACTA
ACGTACGTACTA
CGTACGTACTA
GTACGTACTA
TACGTACTA
ACGTACTA
CGTACTA
GTACTA
TACTA
ACTA
CTA
TA
A
```

Dictionnaires de séquences génomiques

Par exemple, si T = `CACGTACGTACTA`,

On stocke la liste, triée par ordre alphabétique

```
CACGTACGTACTA
ACGTACGTACTA
CGTACGTACTA
GTACGTACTA
TACGTACTA
ACGTACTA
CGTACTA
GTACTA
TACTA
ACTA
CTA
TA
A
```

```
A
ACGTACGTACTA
ACGTACTA
ACTA
CACGTACGTACTA
CGTACGTACTA
CGTACTA
CTA
GTACGTACTA
GTACTA
TA
TACGTACTA
TACTA
```

Algorithmes de tri

Un ensemble d’éléments (mots, nombres) qu’on peut comparer deux à deux

Écrire ces éléments dans l’ordre

Algorithmes de tri

Un ensemble d'éléments (mots, nombres) qu'on peut comparer deux à deux

Ecrire ces éléments dans l'ordre

Algorithme 1:

Tant que la liste a des éléments
    Chercher le minimum, l'enlever de la liste et l'afficher

Complexité?

Algorithmes de tri

Un ensemble d'éléments (mots, nombres) qu'on peut comparer deux à deux

Écrire ces éléments dans l'ordre

Algorithme 1:

Tant que la liste a des éléments
    Chercher le minimum, l'enlever de la liste et l'afficher

Complexité?

$O(n^2)$

Algorithmes de tri

Un ensemble d'éléments (mots, nombres) qu'on peut comparer deux à deux

Sortir ces éléments dans l'ordre

Algorithme 2:

Tri(liste):

Couper la liste en deux parties égales L1 et L2
Tri(L1) et Tri(L2)
Fusionner les listes triées

Fusion(L1,L2):

Tant que les deux listes contiennent des éléments, comparer le premier élément de L1 et de L2, enlever et écrire le plus petit. Compléter par la liste qui reste.

Algorithmes de tri

Complexité :

Fusion(L1,L2):
Tant que les deux listes contiennent des éléments, comparer le premier élément de L1 et de L2, enlever et écrire le plus petit. Compléter par la liste qui reste. → O(n)

Tri(liste): C(n)

Couper la liste en deux parties égales L1 et L2	O(1)
Tri(L1) et Tri(L2)	2C(n/2)
Fusionner les listes triées	O(n)

C(n) = O(n) + O(1) + 2C(n/2)
C(n) = O(n * log(n))

Attention : si la comparaison n'est pas O(1), il faut multiplier par le coût de la comparaison!

Algorithmes de tri

Comme pour la recherche de mots dans un texte, le tri se fait souvent avec des algorithmes (QuickSort) dont la complexité au pire est moins bonne, mais qui sont meilleurs en pratique.

Recherche dans un dictionnaire de suffixes

Construction du dictionnaire:

$O(n^2 \log(n))$ (tri * coût de la comparaison)

Recherche dans le dictionnaire

$O(m \log(n))$ (recherche dichotomique * coût de la comparaison)

La recherche (tâche la plus fréquente) a une complexité faible, mais il faut construire le dictionnaire et la composante n^2 est beaucoup trop élevée.

D'autre part stocker les suffixes prend aussi un espace $O(n^2)$.

Table de suffixes (1990)

Par exemple, si T = `CACGTACGTACTA`,

On stocke non pas la liste des suffixes, mais la liste des indices

```
CACGTACGTACTA
ACGTACGTACTA
CGTACGTACTA
GTACGTACTA
TACGTACTA
ACGTACTA
CGTACTA
GTACTA
TACTA
ACTA
CTA
TA
A
```

```
13 A
 2 ACGTACGTACTA
 6 ACGTACTA
10 ACTA
 1 CACGTACGTACTA
 3 CGTACGTACTA
 7 CGTACTA
11 CTA
 4 GTACGTACTA
 8 GTACTA
12 TA
 5 TACGTACTA
 9 TACTA
```

Table de suffixes

Par exemple, si T = `CACGTACGTACTA`,

On peut faire la recherche dichotomique avec uniquement l'information

```
13  2   6 10 1   3   7 11   4   8 12   5   9
```

Construction et stockage O(n)

Table de suffixes

T = CACGTACGTACTA,

TABLE = 13 2 6 10 1 3 7 11 4 8 12 5 9

Recherche O(m log(n))

début = 1, fin = |T|
Tant que fin > début
    milieu = (début+fin)/2
    S = T[TABLE[milieu],n]
    Si M > S (M est après S par ordre alphabétique) :
        début = milieu+1
    Si M < S :
        fin = milieu-1

Quelques astuces de programmation permettent une recherche en O(m+log(n))

Table de suffixes

Construction de la table :

Trier les positions selon la première lettre, assigner à chaque position son rang

Pour i de 1 à $\log_2(n)$:

Trier les positions selon les 2^i premières lettres,

à partir des positions triées selon les 2^{i-1} premières lettres

---

Passage de i-1 à i :

Pour chaque ensemble de positions avec le même rang :

Pour chaque position de cet ensemble :

Écrire le rang du mot de 2^{i-1} lettres suivant

Trier cet ensemble selon les rangs écrits

Assigner à chaque position son nouveau rang

T = CACGTACGTACTA

Positions des suffixes	1 2 3 4 5 6 7 8 9 10 11 12 13	
Positions triées selon la 1e lettre	2 6 10 13 1 3 7 11 4 8 5 9 12	Seul accès au texte de toute la procédure
Rangs	1 1 1 1 2 2 2 2 3 3 4 4 4	
Position de la deuxième lettre	3 7 11 0 2 4 8 12 5 9 6 10 13	
Rang de la deuxième lettre	2 2 2 0 1 3 3 4 4 4 1 1 1	
Positions triées selon 2 lettres	13 2 6 10 1 3 7 11 4 8 5 9 12	
Rangs	1 2 2 2 3 4 4 5 6 6 7 7 7	
Position des 2 lettres suivantes	0 4 8 12 3 5 9 13 6 10 7 11 0	
Rang des 2 lettres suivantes	0 6 6 7 4 7 7 1 2 2 4 5 0	
Positions triées selon 4 lettres	13 2 6 10 1 3 7 11 4 8 12 5 9	
Rangs	1 2 2 3 4 5 5 6 7 7 8 9 10	
Positions des 4 lettres suivantes	0 6 10 0 4 7 11 0 8 12 0 9 13	
Rang des 4 lettres suivantes	0 2 3 0 7 5 6 0 7 8 0 10 1	
Positions triées selon 8 lettres	13 2 6 10 1 3 7 11 4 8 12 5 9	
Rangs	1 2 3 4 5 6 7 8 9 10 11 12 13	

Complexité en O(nlog n) 1990
Construction à partir d'un arbre de suffixes O(n) 1990
Construction directe en O(n) 2016

Table de suffixes

Structure utilisée dans quelques logiciels bioinformatiques.

Toujours un peu coûteuse en stockage.

15Gb et quelques secondes pour le génome humain

Transformation de Burrows Wheeler (1994,2007)

T=acatacagatg$

$acatacagatg
acagatg$acat
acatacagatg$
agatg$acatac
atacagatg$ac
atg$acatacag
cagatg$acata
catacagatg$a
g$acatacagat
gatg$acataca
tacagatg$aca
tg$acatacaga

Écrire n+1 fois le texte avec un décalage circulaire, et classer les n+1 occurrences par ordre alphabétique

Transformation de Burrows Wheeler

T=acatacagatg$

$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a

Isoler la dernière lettre de chaque occurrence

(Remarque : la première lettre est la première lettre des suffixes rangés selon la table des suffixes)

Transformation de Burrows Wheeler

T=acatacagatg$

	T[SA[i]]		BWT
1	$	acatacagat	g
2	a	cagatg$aca	t
3	a	catacagatg	$
4	a	gatg$acata	c
5	a	tacagatg$a	c
6	a	tg$acataca	g
7	c	agatg$acat	a
8	c	atacagatg$	a
9	g	$acatacaga	t
10	g	atg$acatac	a
11	t	acagatg$ac	a
12	t	g$acatacag	a
	F		L

BWT(ACATACAGATG$) = GT$CCGAATAAA

Transformation de Burrows Wheeler

BWT(ACATACAGATG$) = GT$CCGAATAAA

La structure d’indexation du texte a la taille du texte, et permet une compression optimale. C’est le texte dans un ordre différent, au lieu d’une série de chiffres. Chaque caractère prend 4 bits au lieu de 64 pour un nombre.

Souvent, même si ce n’est pas systématique, les caractères identiques sont regroupés dans BWT, ce qui permet une meilleure compression que le texte lui-même.

Transformation de Burrows Wheeler

Décodage : on connaît
GT$CCGAATAAA et on cherche à retrouver le texte
T=ACATACAGATG$

G		$		G$		$A		G$A	
T		A		TA		AC		TAC	
$		A		$A		AC		$AC	
C		A		CA		AG		CAG	
C		A		CA		AT		CAT	
G	Tri	A	Collage	GA	Tri	AT	Collage	GAT	Tri...Collage...
A		C		AC		CA		ACA	
A		C		AC		CA		ACA	
T		G		TG		G$		TG$	
A		G		AG		GA		AGA	
A		T		AT		TA		ATA	
A		T		AT		TG		ATG	

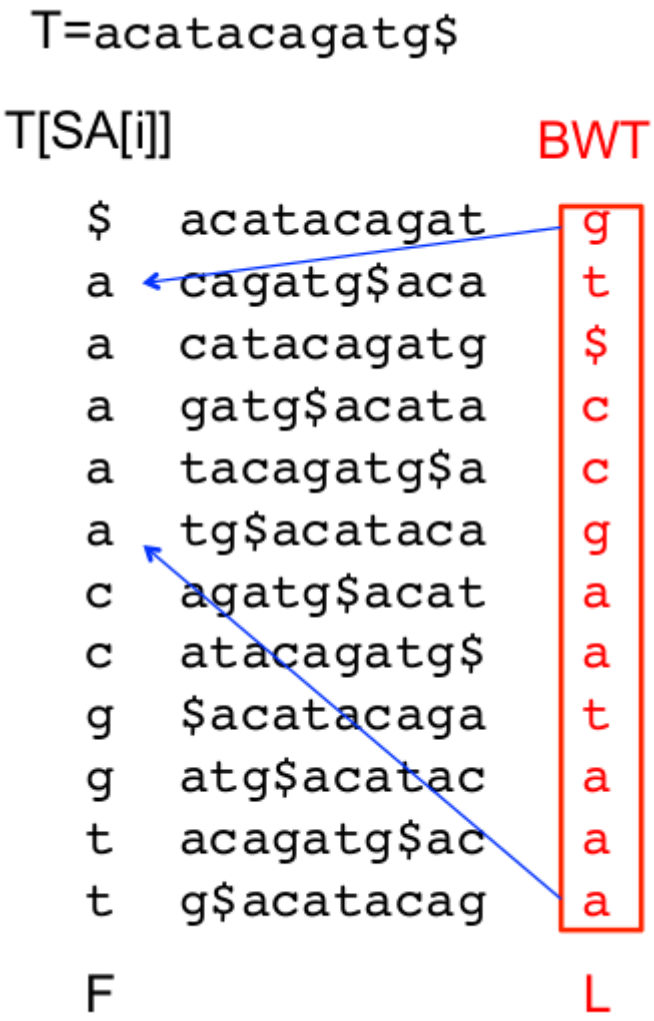
Transformation de Burrows Wheeler
Recherche d'un mot P = taca

T=acatacagatg$

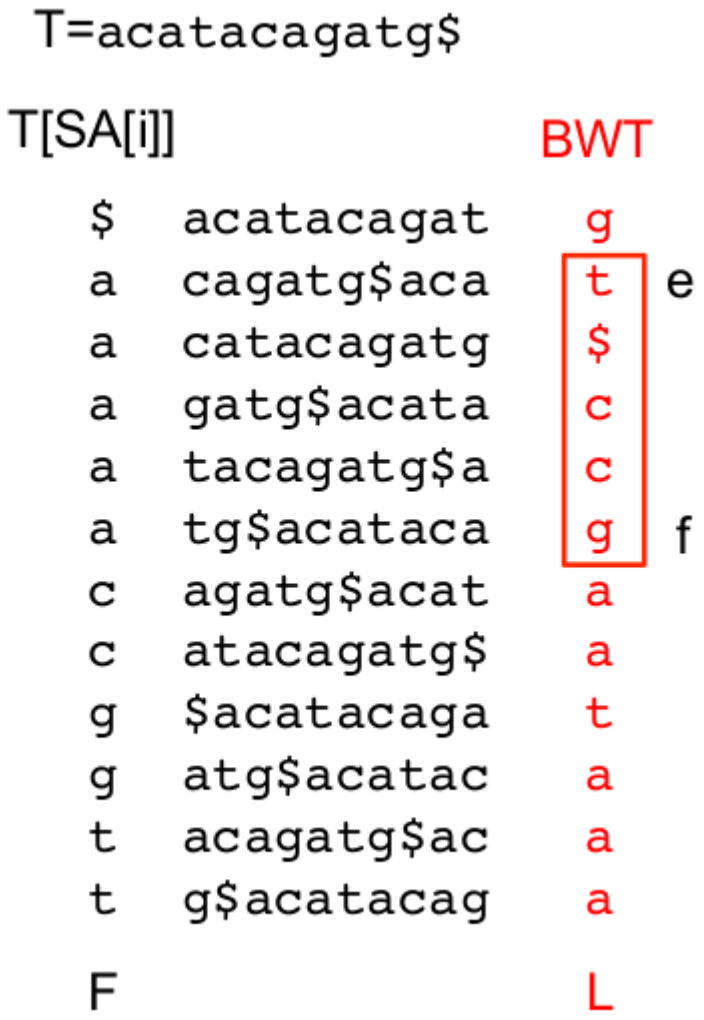
P=taca

T[SA[i]]		BWT
$	acatacagat	g
a	cagatg$aca	t
a	catacagatg	$
a	gatg$acata	c
a	tacagatg$a	c
a	tg$acataca	g
c	agatg$acat	a
c	atacagatg$	a
g	$acatacaga	t
g	atg$acatac	a
t	acagatg$ac	a
t	g$acatacag	a
F		L

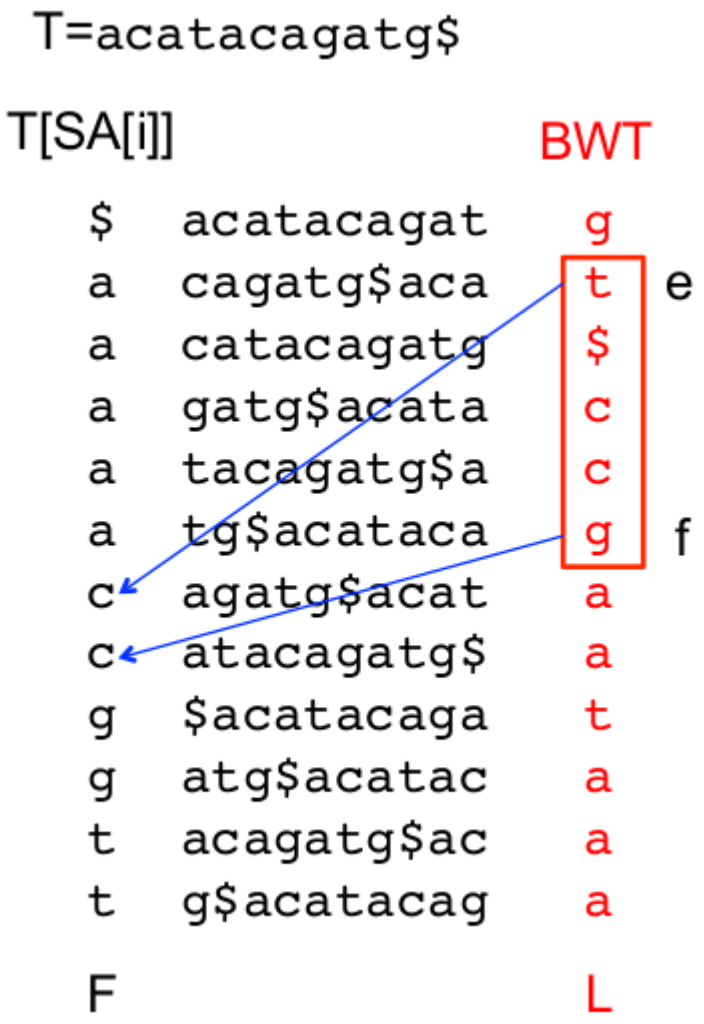
Transformation de Burrows Wheeler
Recherche d'un mot P = taca



Transformation de Burrows Wheeler
Recherche d'un mot P = taca



Transformation de Burrows Wheeler
Recherche d'un mot P = taca

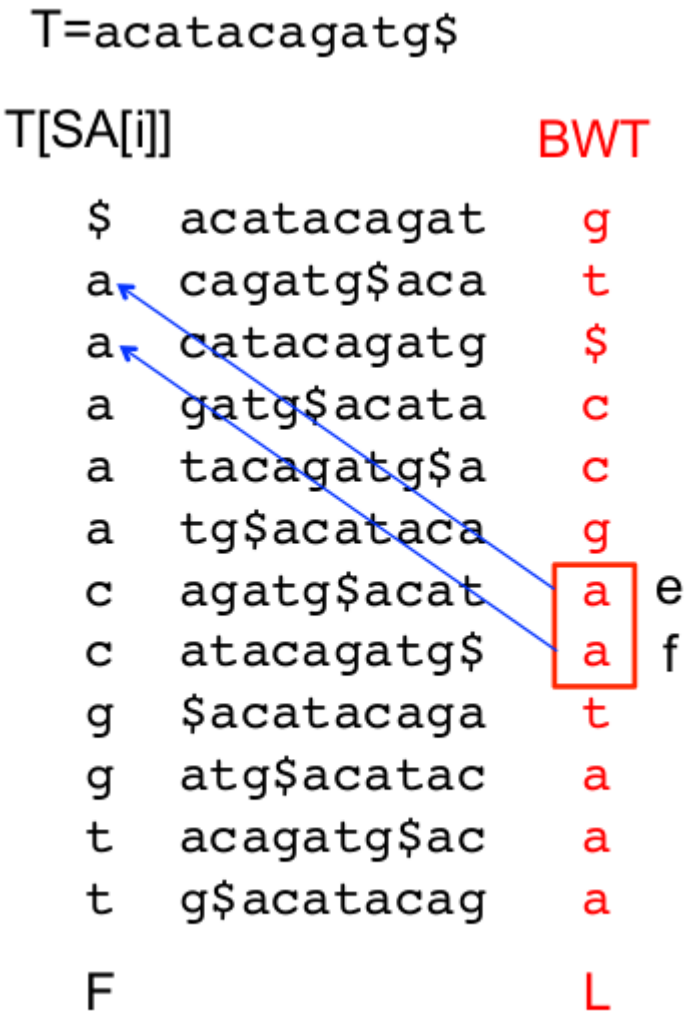


Transformation de Burrows Wheeler
Recherche d'un mot P = taca

T=acatacagatg$

	T[SA[i]]	BWT	
$	acatacagat	g	
a	cagatg$aca	t	
a	catacagatg	$	
a	gatg$acata	c	
a	tacagatg$a	c	
a	tg$acataca	g	
c	agatg$acat	a	e
c	atacagatg$	a	f
g	$acatacaga	t	
g	atg$acatac	a	
t	acagatg$ac	a	
t	g$acatacag	a	
F		L	

Transformation de Burrows Wheeler
Recherche d'un mot P = taca



Transformation de Burrows Wheeler
Recherche d'un mot P = taca

T=acatacagatg$

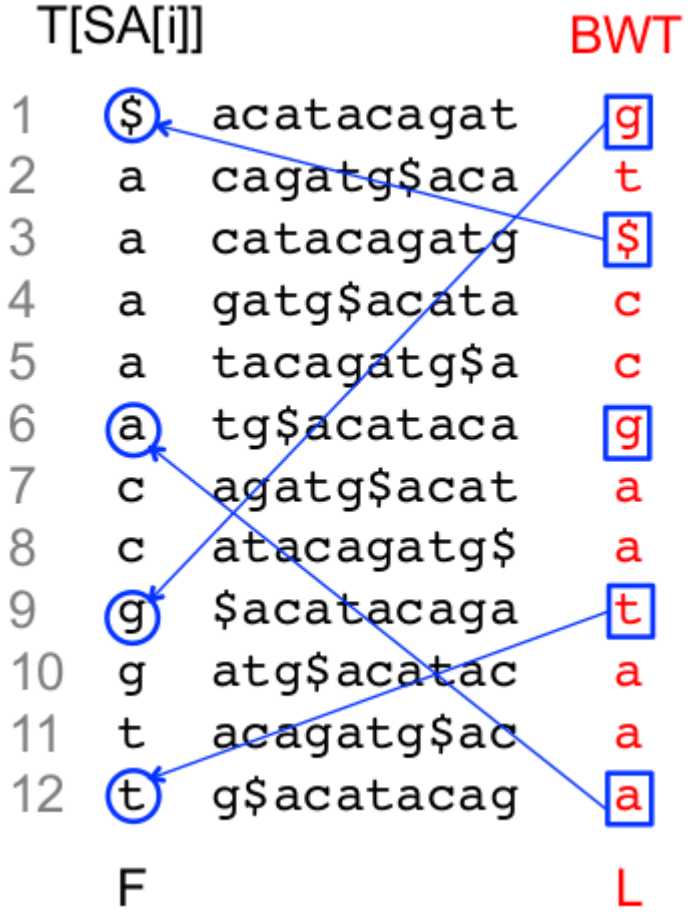
T[SA[i]]		BWT	
$	acatacagat	g	
a	cagatg$aca	t	e
a	catacagatg	$	f
a	gatg$acata	c	
a	tacagatg$a	c	
a	tg$acataca	g	
c	agatg$acat	a	
c	atacagatg$	a	
g	$acatacaga	t	
g	atg$acatac	a	
t	acagatg$ac	a	
t	g$acatacag	a	
F		L	

Transformation de Burrows Wheeler
Recherche d'un mot

e ← 1
f ← n
j ← m
Tant que e <= f et j >= 1:
    r ← numéro de la première occurrence de M[j] dans L[e,f]
    s ← numéro de la dernière occurrence de M[j] dans L[e,f]
    e ← indice de la r ième occurrence de M[j] dans F
    f ← indice de la s ième occurrence de M[j] dans F
    j ← j - 1

Pour un calcul en temps linéaire, il faut avoir accès en temps constant aux numéros et aux indices. C'est possible par un précalcul.

Transformation de Burrows Wheeler
Recherche d'un mot

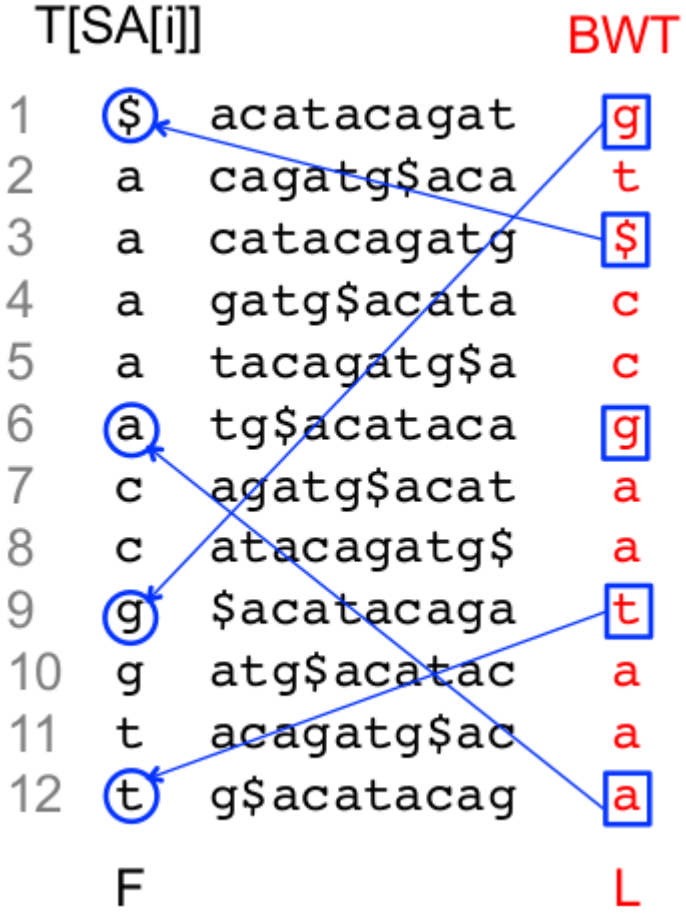


Pointeurs à stocker

Indice du r ième x dans F
Nombre de x dans L[1..i]
Nombre de x dans T

Implémentations compactes de ces fonctions possibles

Transformation de Burrows Wheeler
Recherche d'un mot



Pour localiser l'indice d'une lettre dans T à partir de son indice dans F, stocker à l'avance n/k indices, et poursuivre l'avancée dans le texte jusqu'au prochain. Complexité O(k), taille en mémoire O(n/k).

Transformation de Burrows Wheeler

Structure couramment utilisée dans les logiciels de bioinformatique les plus performants et les plus utilisés (Bowtie, BWA).

Complexité O(n) en ayant stocké les pointeurs appropriés.

Indexation du génome humain 2.2GB (contre 10 à 15Gb pour une table de suffixes)

Mais dans des version de recherche approchée.

Reportez-vous au deuxième document.

Exercices

1/ Combien un mot a-t-il de préfixes? De suffixes? De sous-mots? De sous-séquences? (un sous-mot est une suite de lettres consécutives dans le mot, une sous-séquence est une suite de lettres non nécessairement consécutives)

2/ Quelle est la complexité d'un algorithme dont le nombre d'opérations peut s'exprimer par
f(1) = a
f(n) = b*n + 2*f(n/2)

Exercices

3/ Pour le texte T=TROTTINETTE, construisez
- une table de suffixes
- une transformée de Burrows-Wheeler

4/ Calculez la complexité de l'algorithme de construction et décodage BWT vus en cours

5/ Proposez un algorithme de construction de la BWT de complexité O(n) à partir de la table des suffixes

Exercices

6/ Le décodage BWT vu dans le cours est très coûteux. Comment l'optimiser? Construisez un algorithme à partir de la séquence transformée (vecteur L), de la permutation de ses caractères en les triant par ordre lexicographique (vecteur F), et de la relation entre les deux vecteurs (la lettre L[i] précède la lette F[i] dans le texte).

7/ Étant donnée la transformée BWT = e$elplepa, reconstituer le texte. Qu'indique cet exemple sur les capacités de compression de BWT?

8/ Avec BWT, comment, à partir de l'algorithme de recherche d'un mot, peut-on facilement obtenir le nombre d'occurrences de ce mot?

Exercices

9/ Un k-mer est un mot de taille k dans un alphabet de taille a.

9.1- Compter le nombre de k-mers possibles.

9.2- On a maintenant un texte T, et pour chaque k-mer possible, on stocke dans un tableau toutes les positions de T où le k-mer apparaît. Par exemple, si T=ATTT et k=2, les positions sont pos(AT) ={1}, pos(TT)={2,3} et une liste vide pour tous les autres k-mers. Quelle est la complexité de la recherche d'un mot de taille k dans un texte de taille n si on dispose d'une telle structure?

9.3- Quelle est la complexité en temps et en mémoire de la construction de cette structure?

À quoi sert l'algorithme suivant :

```
Entree : une liste L de nombres, de taille n
variable booleenne paire_inversee = Vrai
tant que paire_inversee == Vrai:
  paire_inversee = Faux
  pour i de 1 a n-1:
    Si L[i] > L[i+1]:
      variable nombre temp = L[i]
      L[i] = L[i+1]
      L[i+1] = temp
      paire_inversee = Vrai
    Fin_si
  Fin_pour
Fin_tant_que
Sortie : liste L
```

Quelle est sa complexité ?

Et celui-là?

Entrée : une liste de nombres L, de taille n

Soit L2 une liste vide

Pour i variant de 1 à n :
    début = 1, fin = taille de L2,
    Tant que fin-début > 0
        milieu = int((début+fin)/2)
        Si L2[milieu] < L[i]
            début = milieu + 1
        Si L2[milieu] >= L[i]:
            fin = milieu

    ajouter L[i] à L2 en l'insérant à la position début