Burrows - Wheeler Transform (BWT)

巡回ソート (block sort):

(\$) A T G　　目印として置くものを番兵 (sentinel) と呼ぶ
A T G \$
T G \$ A
G \$ A T

接尾辞配列 (Suffix Array):
その接尾辞が元の配列の何番目かを表す

接尾辞 (suffix) は、文字列 seqのi番目から末尾までを並べた文字列.

よって、ただ1つ巡回してない文字列の先頭は、そこから始まる接尾辞がくる。
巡回ソートの並び順は、接尾辞のソートの並び順と同一になる

位置	巡回	接尾辞	巡回ソート	接尾辞ソート	SA
0	\$GCTTACGTAT	\$GCTTACGTAT	\$G ..... T	———	0
1	GCTTACGTAT\$	GCTTACGTAT	A	—	5
2	CTTACGTAT\$G	CTTACGTAT		———	9
3	TTACGTAT\$GC	TTACGTAT		———	6
4	TACGTAT\$GCT	TACGTAT		———	2
⋮	⋮	⋮		⋮	⋮
10	T\$GCTTACGTA	T		———	3

⇒ (F) (L)

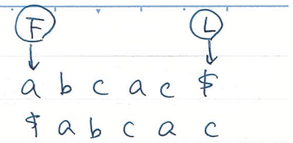
巡回ソートの最後の文字を並べた文字列を Burrows-Wheeler Transform と呼ぶ.

☆「Lをアルファベット順に並び替えるとFになる。」　☆「Lの次がF」

文字列の長さをnとしたとき、BWTにかかる時間はおよそ $O(n^2)$

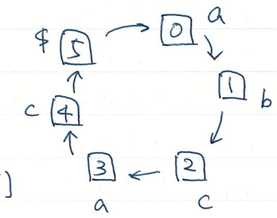
ただし、BWTを $O(n)$ または $O(n\log n)$ で行うアルゴリズムも存在する.

FM index：BWTを用いて文字列の検索を行うためのアルゴリズム



Burrows-Wheeler Transformの配列Lに対して、次の2つを定義する:

配列O: その位置まで(含む)に、各々の文字が登場する回数
配列C: その位置にある文字が、F上で初めて出現する位置(インデックス) - 1



i	L	O[\$,i]	O[A,i]	O[C,i]	O[G,i]	O[T,i]	C[i]	LF[i]
0	T	0	0	0	0	1		
1	T	0	0	0	0	2		
2	T	0	0	0	0	3		
3	A	0	1	0	0	3		
4	G	0	1	0	1	3		
⋮	⋮	⋮	⋮	⋮	⋮	⋮		
10								

C[i]: Fに戻ってくる
LF[i]: (あとで分かる)

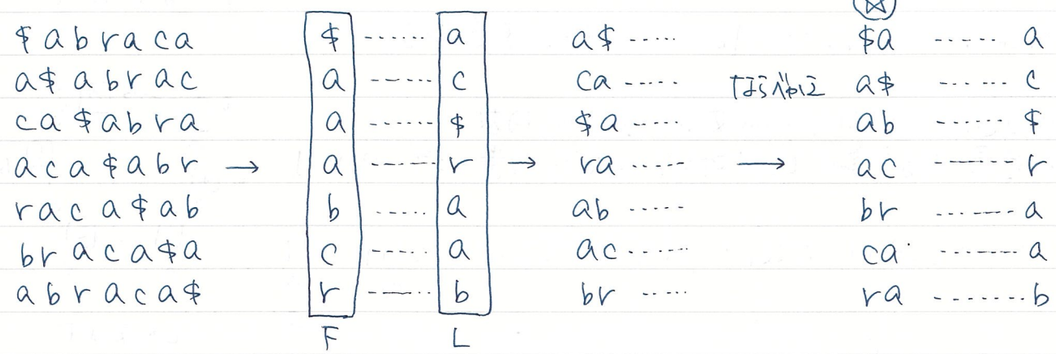
Lの「その文字」が、Fの中で何番目に現れるかを求めた対応関係を「LF mapping」と呼び
「FはLの次」なので、Fの中での「k番目」がわかれば、その行のLが Fの1つ前にあたる.
→ 以下より 逆に元の配列を後ろから復元できる

FM indexの作成は、長さnの配列をただ読みだけなので $O(n)$ でできる

No.

Date

BWTを用いたもとの配列の復元



☆のところで同じものが複数あったとしても，あくまで最初の配列の復元過程であるので，アルファベット順に並べておきさえすれば復元は問題なく進む．

これで元の文字列が復元できるが，n文字の配列に対してメモリを O(n²) 使うので非効率。

BWTを用いたもとの配列の復元 (with LF mapping)　　$abraca

Ⓕ　Ⓛ

$　a
a　c
a　$
a　r
b　a
c　a
r　b

「Lの文字が下」後ろから上へのぼって配列を決める．

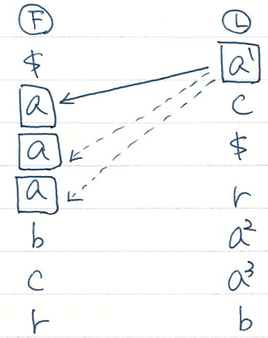
cが下の中で何番目にあるかは，「アルファベット順でcより前の文字数」つまり

$(1コ) + a(3コ) + b(1コ) = 5

よって．cは下の5番目になる（インデックスが0からなので）

ここで複数ある文字をどのように識別するか，ということを考える

ex. a'の1つ前はc，$，rのどれか？



↓ 1ステップすすめると

Ⓕ	Ⓛ
a'$	$a ..... a'
ca	a'$ c
$a	a²b ..... $
ra	a³c ..... r
a²b	br ..... a²
a³c	ca ..... a³
br	ra ..... b

ソートすると →

$　---- a'　　a'$ ...
b　---- a²　→　a²b ---
c　---- a³　　a³c ....

↑　　↑

ここがすでにアルファベット順に並んでいるので，aをいくつずらしても並び順は同じ。

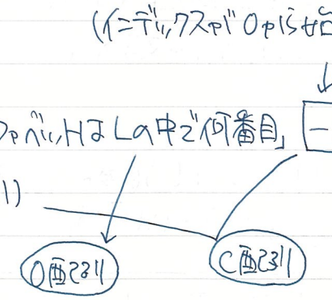
よって，aの前を知りたければ，下の中で「aの1コ目」に相当する行の右端を見ればよい。

これを一般化すると，LF mappingを行う際に，LF[i]は

（インデックスが0から始まるので）

「そのアルファベットよりも小さいアルファベットの総数」＋「そのアルファベットはLの中で何番目」 -1 番目

= 下上でそのアルファベットが初めて出現するインデックス（：0始まり）



よって．L[i] = X とすると

$$LF[i] = C[X] + O[X, i] \quad (= C[L[i]] + O[L[i], i])$$

FM indexの記法を用いて，これをもう一度整理する（Lに対するFは，$aaaabcr）

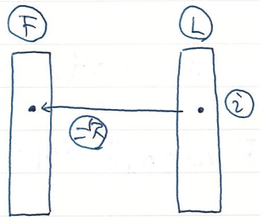
L	O[$,i]	O[a,i]	O[b,i]	O[c,i]	O[r,i]	C[i]	LF[i]
a	0	(1)	0	0	0	0	1
c	0	1	0	(1)	0	4	5
$	(1)	1	0	1	0	-1	0
r	1	1	0	1	(1)	5	6
a	1	(2)	0	1	1	0	2
a	1	(3)	0	1	1	0	3
b	1	3	(1)	1	1	3	4

LF[i]は「i番目の値は，Fの中では何番目？」（同じLの値についても，Fが辞書順に並んでいるときの，その中での順序）

→ 要はLをアルファベット順に並べ替えたときのその文字のインデックスに一致する。

LF[i]と接尾辞配列との関係

SA[i]



辞書順ソートと接尾辞ソートの順番は一致するので
SA[i]は，L[i]の次の文字の，もとの文字列における位置を表す。

一方で，LF[i]は，L[i]自身が，Fの中で何番目か？を表すので，SA[LF[i]]は，L[i]自身がもとの文字列において何番目かを表す。

これより，$SA[LF[i]] \equiv SA[i]-1 \pmod n$ が成立する

FM indexを用いた文字列検索

- 直観的にはこれまでと同じように後ろから文字のぼって範囲を絞ることで探索できる

ex. TTA　③②①　最初の配列でどのように検索していく？

F	C[F]	L	O[T]
$	-1	T[1]	1
① A	0	T[2] sp	2
① A	0	T[3] ep	3
C	2	A	3
C	2	G	3
G	4	$	3
G	4	C	3
T	6	A	3
② T	6	T[4]	4
③ T	6	G	4
T	6	C	4

①→②について；
最初と最後をsp, epとすると
sp = C[①の文字] + 1
ep = C[アルファベット順で①の次の文字]
として，sp≦i≦epの範囲のL[i]を探索する。

②→③について；
②の文字(T)は，spより前には，O[②の文字, sp-1]個ある。
よって，[sp, ep]の間に存在するTは，O[②の文字, sp-1]+1番目から
O[②の文字, ep]番目までになる。

よって，②→③における検索範囲は，
sp = C[②の文字] + O[②の文字, sp-1] + 1
ep = C[②の文字] + O[②の文字, ep]

このように順次sp, epを更新しながら検索を進める．sp≦i≦epの範囲（のL[i]の中）に該当の文字がない場合は

O[②の文字, sp-1] = O[②の文字, ep]

ゆえ，sp > epとなるので，ここで検索終了．

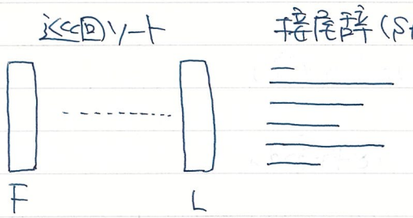
最後まで行ってもsp≦epなら，クエリは対象の文字列内に含まれる。

接尾辞配列(SA)

FM indexを用いると，対象の文字列内のクエリの有無がわかる。さらに逆BWT変換を行うことで，クエリと一致した配列の場所がわかる。

しかし，元のゲノム上のどこに位置するクエリかを毎回逆BWT変換を行うのは計算に時間がかかる。

ここで接尾辞配列を利用する（巡回ソートと接尾辞ソートの並びは同じ）



接尾辞配列はLの次の文字のもとの配列における位置を表す。よって，後ろから順番に探索にいって，先頭の文字Lにつくまで行ったとして，SAにつくを見れば，それが「先頭の次の文字」の位置になるので，これの1つ前が先頭の位置。

ただし SA[LF[i]] = SA[i]-1 であるので，実際には接尾辞をソートしてSA[ ]を求めるのではなく，逆BWT変換をしながらSAを求める方が速い。

実際にはSAをデフォルトで記憶させるとデータサイズを食うので，10文字おき，100文字おきなど間引いた状態でSA[ ]を記憶させておいて，実際のクエリの先頭から逆BWT変換で，位置が分かるところまで戻る。

iに対してLF[i]をすると，SA[LF[i]] = SA[i]-1より，L[i]の1つ前の文字のもとの文字列における位置を表す。これを繰り返して，(LF^k[i], SA[LF^k[i]])のペアが発見できれば，

SA[i] = SA[LF^k[i]]+k

とできる。