BWT-Transformation

- What is BWT-transformation?
- BWT string compression
- BWT string matching
  - RankAll
  - BWT array construction

BWT transformation

- We use *s* to denote a string that we would like to transform.
- Assume that *s* terminates with a special character \$, which does not appear elsewhere in *s* and is alphabetically prior to all other characters.
- In the case of DNA sequences, we have $\$ < A < C < G < T$.

BWT transformation

- As an example, consider $s = acagaca\$$. We can rotate s consecutively to create eight different strings as shown in Fig. 1(a).

First column *Last column*

	F	L	
$a\ c\ a\ g\ a\ c\ a\ \$$	a_1	$\$$	$\$\ a\ c\ a\ g\ a\ c\ a$
$c\ a\ g\ a\ c\ a\ \$\ a$	c_1	a_1	$a\ \$\ a\ c\ a\ g\ a\ c$
$a\ g\ a\ c\ a\ \$\ a\ c$	a_2	c_1	$a\ c\ a\ \$\ a\ c\ a\ g$
$g\ a\ c\ a\ \$\ a\ c\ a$	g_1	a_2	$a\ c\ a\ g\ a\ c\ a\ \$$
$a\ c\ a\ \$\ a\ c\ a\ g$	a_3	g_1	$a\ g\ a\ c\ a\ \$\ a\ c$
$c\ a\ \$\ a\ c\ a\ g\ a$	c_2	a_3	$c\ a\ \$\ a\ c\ a\ g\ a$
$a\ \$\ a\ c\ a\ g\ a\ c$	a_4	c_2	$c\ a\ g\ a\ c\ a\ \$\ a$
$\$\ a\ c\ a\ g\ a\ c\ a$	$\$$	a_4	$g\ a\ c\ a\ \$\ a\ c\ a$

Fig. 1: Rotation of a string

BWT transformation

- By writing all these strings stacked vertically, we generate an $n \times n$ matrix, where $n = |s|$ (see Fig. 1(a).) Here, special attention should be paid to the first column, denoted as F, and the last column, denoted as L. For them, the following equation, called the *LF mapping*, can be immediately observed:

$$F[i] = L[i]\text{'s predecessor}, \qquad (1)$$

where $F[i]$ ($L[i]$) is the i^{th} element of F (resp. L).

BWT transformation

- Now we sort the rows of the matrix alphabetically. We will get another matrix, called the *Burrow-Wheeler Matrix* and denoted as *BWM*(*s*).
- Especially, the last column of *BWM*(*s*), read from top to bottom, is called the *BWT*-transformation (or the *BWT*-array) and denoted as *BWT*(*s*). So for *s* = *acagaca*$, we have *BWT*(*s*) = *acg*$*caaa*.

$	*a*	*c*	*a*	*g*	*a*	*c*	*a*
a	$	*a*	*c*	*a*	*g*	*a*	*c*
a	*c*	*a*	$	*a*	*c*	*a*	*g*
a	*c*	*a*	*g*	*a*	*c*	*a*	$
a	*g*	*a*	*c*	*a*	$	*a*	*c*
c	*a*	$	*a*	*c*	*a*	*g*	*a*
c	*a*	*g*	*a*	*c*	*a*	$	*a*
g	*a*	*c*	*a*	$	*a*	*c*	*a*

Fig. 2: Sorting the rows of the matrix

BWT transformation

- By the *BWM* matrix, the *LF*-mapping is obviously not changed.
- Surprisingly, the rank correspondence also remains. Even though the ranks of different appearances of a certain character (in *F* or in *L*) may be different from before, their rank correspondences are not changed as shown in Fig. 2(b), in which a_2 now appears in both *F* and *L* as the third element among all the *a*-characters, and c_1 the second element among all the *c*-characters.

rk_F	F	L	rk_L
–	\$	a_4	1
1	a_4	c_2	1
2	a_3	g_1	1
3	a_1	\$	–
4	a_2	c_1	2
1	c_2	a_3	2
2	c_1	a_1	3
1	g_1	a_2	4

By ranking the elements in *F*, each element in *L* is also ranked with the same number.

$F_\$ = <\$; 1, 1>$

$F_a = <a; 2, 5>$

$F_c = <c; 6, 7>$

$F_g = <g; 8, 8>$

Fig. 3: *LF*-mapping and tank-correspondence

BWT string compression

- The first purpose of *BWT*(*s*) is for the string compression since same characters with similar *right-contexts* in *s* tend to be clustered together in *BWT*(*s*), as shown by the following example:

  BWT(tomorrow and tomorrow and tomorrow)

  = wwwdd nnoooaatttmmmrrrrrrooo $ooo

BWT string matching

- For the purpose of the string search, the character clustering in F has to be used. Especially, for any DNA sequence, the whole F can be divided into five or less segments: \$-segment, A-segment, C-segment, G-segment, and T-segment, denoted as $F_{\$}$, F_A, F_C, F_G, F_T, respectively.
- In addition, for each segment in F, we will rank all its elements from top to bottom, as illustrated in Fig. 2(a). \$ is not ranked since it appears only once.

$F_{\$} = \langle \$; 1, 1 \rangle$

$F_a = \langle a; 2, 5 \rangle$

$F_c = \langle c; 6, 7 \rangle$

$F_g = \langle g; 8, 8 \rangle$

BWT string matching

- From Fig. 2(a), we can see that the rank of a_4, denoted as $rk_F(a_4)$, is 1 since it is the first element in F_A. For the same reason, we have $rk_F(a_3) = 2$, $rk_F(a_1) = 3$, $rk_F(a_2) = 4$, $rk_F(c_2) = 1$, $rk_F(c_1) = 2$, and $rk_F(g_1) = 1$.
- It can also be seen that each segment in F can be effectively represented as a triplet of the form: $<\alpha; x_\alpha, y_\alpha>$, where $\alpha \in \Sigma \cup \{\$\}$, and x_α, y_α are the positions of the first and last appearance of α in F, respectively.
- Now we consider α_j (the j^{th} appearance of α in s). Assume that $rk_F(\alpha_j) = i$. Then, the position where α_j appears in F can be easily determined:

$$F[x_\alpha + i - 1] = \alpha_j. \qquad (2)$$

BWT string matching

- In addition, if we rank all the elements in L top-down in such a way that an α_j is assigned i if it is the i^{th} appearance among all the appearances of α in L. Then, we will have

$$rk_F(\alpha_j) = rk_L(\alpha_j), \qquad (3)$$

  where $rk_L(\alpha_j)$ is the rank assigned to α_j in L.

- With the ranks established, a string matching can be very efficiently conducted by using the formulas (2) and (3).

BWT string matching

- To see this, let's consider a pattern string $p = aca$ and try to find all its occurrences in $s = acagaca\$$.
- First, we check $p[3] = a$ in the pattern string p, and then figure out a segment in L, denoted as L', corresponding to $F_a = <a; 2, 5>$. So $L' = L[2 .. 5]$, as illustrated in Fig. 3(a), where we still use the non-compact F for ease of explanation.
- In the second step, we check $p[2] = c$, and then search within L' to find the first and last c in L'. We will find $rk_L(c_2) = 1$ and $rk_L(c_1) = 2$. By using (3), we will get $rk_F(c_2) = 1$ and $rk_F(c_1) = 2$. Then, by using (2), we will figure out a sub-segment F' in F: $F[x_c + 1 - 1 .. x_c + 2 - 1] = F[6 + 1 - 1 .. 6 + 2 - 1] = F[6 .. 7]$. (Note that $x_c = 6$. See Fig. 3(b).)

BWT string matching

- In the third step, we check $p[1] = a$, and find $L'' = L[6 .. 7]$ corresponding to $F' = F[6 .. 7]$. Repeating the above operation, we will find $rk_L(a_3) = 2$ and $rk_L(a_1) = 3$. See Fig. 3(c).
- Since now we have exhausted all the characters in p and $F[x_a + 2 - 1, x_a + 3 - 1] = F[3, 4]$ contains only two elements, two occurrences of p in s are found, corresponding to a_1 and a_3 in s, respectively.

BWT string matching

F	L	rk_L
\$	a_4	1
a_4	c_2	1
a_3	g_1	1
a_1	\$	–
a_2	c_1	2
c_2	a_3	2
c_1	a_1	3
g_1	a_2	4

To find the first *c*

to find the last *c*

F	L	rk_L
\$	a_4	1
a_4	c_2	1
a_3	g_1	1
a_1	\$	–
a_2	c_1	2
c_2	a_3	2
c_1	a_1	3
g_1	a_2	4

to find the first *a*

to find the last *a*

F	L	rk_L
\$	a_4	1
a_4	c_2	1
a_3	g_1	1
a_1	\$	–
a_2	c_1	2
c_2	a_3	2
c_1	a_1	3
g_1	a_2	4

Fig. 3: Sample trace

RankAll

- The dominant cost of the above process is the searching of *L* in each step.
- However, this can be dramatically reduced by arranging |Σ| arrays each for a character $\alpha \in \Sigma$ such that $\alpha[i]$ (the i^{th} entry in the array for α) is the number of appearances of α within $L[1 .. i]$. See Fig. 4(a) for illustration.respectively.

j	F	L	\$	a	c	g	t
1	\$	a_4	0	1	0	0	0
2	a_4	c_2	0	1	1	0	0
3	a_3	g_1	0	1	1	1	0
4	a_1	\$	1	1	1	1	0
5	a_2	c_1	1	1	2	1	0
6	c_2	a_3	1	2	2	1	0
7	c_1	a_1	1	3	2	1	0
8	g_1	a_2	1	4	2	1	0

i	A_a	A_c	A_g	A_t
0	0	0	0	0
1	1	1	1	0
2	4	2	1	0

For each β = 4 values in *L*, a *rankAll* value is stored.

Fig. 4: *LF*-mapping and rank-correspondence

RankAll

- Now, instead of scanning a certain segment $L[x .. y]$ ($x \leq y$) to find a subrange for a certain $\alpha \in \Sigma$, we can simply look up the array for α to see whether $\alpha[x - 1] = \alpha[y]$. If it is the case, then α does not occur in $L[x .. y]$.
- Otherwise, $[\alpha[x - 1] + 1, \alpha[y]]$ should be the found range. For example, to find the first and the last appearance of c in $L[2 .. 5]$, we only need to find $c[2 – 1] = c[1] = 0$ and $c[5] = 2$. So the corresponding range is $[c[2 - 1] + 1, c[4]] = [1, 2]$.

RankAll

- The problem of this method is the high space requirement, which can be mitigated by storing a compact array A_α for each $\alpha \in \Sigma$, in which, rather than for each $L[i]$, only for some elements in L the number of their appearances will be stored.
- For example, we can divide L into a set of buckets of the same size and only for each bucket a value will be stored in A_α.

i	A_a	A_c	A_g	A_t
0	0	0	0	0
1	1	1	1	0
2	4	2	1	0

For each $\beta = 4$ values in L, a *rankAll* value is stored.

RankAll

- Obviously, doing so, more search will be required. In practice, the size β of a bucket (referred to as a *compact factor*) can be set to different values.
- For example, we can set $\beta = 4$, indicating that for each four contiguous elements in L a group of $|\Sigma|$ integers (each in an A_α) will be stored. However, each $\alpha[j]$ for $\alpha \in \Sigma$ can be easily derived from A_α by using the following formulas:

  $$\alpha[j] = A_\alpha[i] + r, \qquad (4)$$

  where $i = \lfloor j/\beta \rfloor$ and r is the number of α's appearances within $L[i \cdot \beta \,..\, j]$, and

  $$\alpha[j] = A_\alpha[i'] - r', \qquad (5)$$

  where $i' = \lceil j/\beta \rceil$ and r' is the number of α's appearances within $L[j \,..\, i' \cdot \beta]$.

Construction of Arrays

- As mentioned above, a string $s = c_0c_1 \dots c_{n-1}$ is always ended with \$ (i.e., $c_i \in \Sigma$ for $i = 0, \dots, n-2$, and $c_{n-1} = \$$). Let $s[i] = c_i$, $i = 0, 1, \dots, n-1$, be the i^{th} character of s, $s[i..j] = c_i \dots c_j$ substring and $s_i = s[i .. n-1]$ a suffix of s. Suffix array A of s is a permutation of the integers $0, \dots, n-1$ such that $A[i]$ is the start position of the i^{th} smallest suffix. The relationship between A and the BWT array L can be determined by the following formulas:

$$L[i] = \begin{cases} \$, & \text{if } A[i] = 0; \\ s[A[i]-1], & \text{otherwise.} \end{cases} \qquad (6)$$

- Once L is determined. F can be created immediately by using formula (1).