

# Alignment of short/long-read sequencing data

2019 Dragon Star Bioinformatics Course (Day 2)

# Sequence similarity

- For any two sequences
  - Mutation: mismatch
  - Insertion
  - Deletion

*DNA-sequence-1*

tcctc tgc t c gat gcc at cat -- ca acc a ca agt  
||||| ||| | ||| ||| ||| ||| ||| ||| |||  
tcctg tgc a tc -- tg ca at cat gg ca acc a ga agt

*DNA-sequence-2*

# Sequence alignment

- Sequence alignment
  - Consider matches, mismatches, insertions and deletions (indels are gaps)
  - Find an optimal alignment between sequences
- Applications
  - Reference-based read mapping
  - Genome assembly
  - Gene finding

# Pairwise alignment

- Sequence alignment between two sequences
  - Find an optimal alignment between two sequences
- Dynamic programming
  - Input
    - Two sequences
    - Score matrix

# Dynamic programming - score matrix

- Simple scoring matrix
  - Assume:  $a, b$  are two bases

- $\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -2 & \text{if } b = \text{gap} \end{cases} \quad \text{or} \quad \phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -2 & \text{for open gaps} \\ -1 & \text{for extended gaps} \end{cases}$

	A	C	G	T
A	2	-3	-3	-1
C	-2	3	-1	-2
G	-2	-1	4	-3
T	-1	-1	-2	1

# Dynamic programming

- Purpose:
  - To find an alignment between two sequences with best matching scores
- Three components
  - Recursive calculation
  - Tabular arrangement
  - Traceback
- Three common types of pairwise alignments
  - Global alignment: Needleman-Wunsch
  - Local alignment: Smith-Waterman
  - Semi-global alignment

# Global alignment: Needleman-Wunsch

- Best global alignment
  - Have maximal alignment score with all bases in two sequences
  - Assume we have two sequences: P and Q
    - P = TCATGGC
    - Q = TCATC
  - Score functions

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

# Global alignment: Needleman-Wunsch

- 1. Tabular arrangement
  - $C(0, j) = \sum_{1 \leq k \leq \|P\|} \phi(-, P(k))$
  - $C(i, 0) = \sum_{1 \leq k \leq \|Q\|} \phi(Q(k), -)$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
	-	0	-1	-2	-3	-4	-5	-6	-7
	T	-1							
	C	-2							
	A	-3							
	T	-4							
	C	-5							



# Global alignment: Needleman-Wunsch

- 2. Recursive calculation  $C(i, j) = \max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \end{cases}$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
	-	0	-1	-2	-3	-4	-5	-6	-7
	T	-1							
	C	-2							
	A	-3							
	T	-4							
	C	-5							

# Global alignment: Needleman-Wunsch

- 2. Recursive calculation  $C(i, j) = \max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \end{cases}$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
	-	0	-1	-2	-3	-4	-5	-6	-7
	T	-1							
	C	-2							
	A	-3							
	T	-4							
	C	-5							

# Global alignment: Needleman-Wunsch

- 2. Recursive calculation  $C(i, j) = \max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \end{cases}$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
	-	0	-1	-2	-3	-4	-5	-6	-7
	T	-1							
	C	-2							
	A	-3							
	T	-4							
	C	-5							

# Global alignment: Needleman-Wunsch

- 2. Recursive calculation  $C(i, j) = \max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \end{cases}$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

Q

<u>P</u>	-	T	C	A	T	G	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
T	-1	1						
C	-2							
A	-3							
T	-4							
C	-5							

# Global alignment: Needleman-Wunsch

- 2. Recursive calculation  $C(i, j) = \max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \end{cases}$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

Q

<u>P</u>	-	T	C	A	T	G	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
T	-1	1	-1	-1	-2	-3	-4	-5
C	-2	0	2	-1	0	-1	-2	3
A	-3	-1	-1	3	-1	1	-1	-1
T	-4	-2	-2	0	4	-1	2	1
C	-5	-3	-1	-1	3	3	2	3

# Global alignment: Needleman-Wunsch

- 3. Traceback
  - Check which operation obtained the current alignment score.

Q

P

	-	T	C	A	T	G	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
T	-1	1	0	-1	-2	-3	-4	-5
C	-2	0	2	1	0	-1	-2	-3
A	-3	-1	1	3	2	1	0	-1
T	-4	-2	-2	0	4	3	2	1
C	-5	-3	-1	-1	3	3	2	3

```
TCATGGC
||| |
TCAT--C
```

# Local alignment: Smith-Waterman

- Best local alignment
  - Have maximal alignment score with a subset of bases in two sequences
- Assume we have two sequences: P and Q
  - P = TCATGGC
  - Q = TCATC
- Score functions

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

# Local alignment: Smith-Waterman

- 1. Tabular arrangement

- $C(0, j) = 0$

- $C(i, 0) = 0$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
	-	0	0	0	0	0	0	0	0
	T	0							
	C	0							
	A	0							
	T	0							
	C	0							



# Local alignment: Smith-Waterman

- 2. Tabular arrangement  $(i, j) =$

$$\max \begin{cases} C(i-1, j-1) + \phi(Q(i), P(j)) \\ C(i-1, j) + \phi(Q(i), -) \\ C(i, j-1) + \phi(-, P(j)) \\ 0 \end{cases}$$

Q  
P

$$\phi(a, b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a \neq b \\ -1 & \text{if } a = \text{gap} \\ -1 & \text{if } b = \text{gap} \end{cases}$$

	-	T	C	A	T	G	G	C
-	0	0	0	0	0	0	0	0
T	0	1	-1	-1	0	-1	-1	-1
C	0	-1	2	-1	-1	0	-1	1
A	0	-1	-1	3	-1	-1	-1	-1
T	0	1	0	0	4	3	2	1
C	0	-1	0	-1	-1	3	2	3

# Local alignment: Smith-Waterman

T C A T  
| | | |  
T C A T

- 3. Traceback
  - 1. Find a best score recursively
  - 2. Check which operation is used to obtain the current alignment score.
  - 3. Stop when an alignment is 0

		<u>Q</u>							
<u>P</u>		-	T	C	A	T	G	G	C
-	-	0	0	0	0	0	0	0	0
T	T	0	1	0	0	1	0	0	0
C	C	0	0	2	1	0	0	0	1
A	A	0	0	1	3	2	1	0	0
T	T	0	0	0	0	4	3	2	1
C	C	0	0	0	0	3	3	2	3

# Local alignment: Smith-Waterman

---

	-	T	A	T	C	T	T	A	A	C	G	C	C
-	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	1	0	0
A	0	0	1	0	0	0	0	1	1	0	0	0	0
T	0	1	0	2	1	1	1	0	0	0	0	0	0
C	0	0	0	1	3	2	1	0	0	1	0	1	1
A	0	0	1	0	2	2	1	2	1	0	0	0	0
A	0	0	1	0	1	1	1	2	3	2	1	0	0
T	0	1	0	2	1	2	2	1	2	2	1	0	0
T	0	1	0	1	1	2	3	2	1	1	1	0	0
C	0	0	0	0	2	1	2	2	1	2	1	2	1
G	0	0	0	0	1	1	1	1	1	1	3	2	1
C	0	0	0	0	1	0	0	0	0	2	2	4	3
A	0	0	1	0	0	0	0	1	1	1	1	3	3

---

# BLAST

- BLAST is used for sequence similarity search, and much faster than Smith-Waterman method
- Compare a query sequence against a database of sequences.
- BLAST is a collection of algorithms
  - BLASTN: nucleotide sequence against nucleotide database
  - BLASTP: protein sequence against protein database
  - BLASTX: translated nucleotide sequence against protein database
  - TBLASTX: six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database
  - TBLASTN: protein sequence against translated nucleotide databases

# BLAST

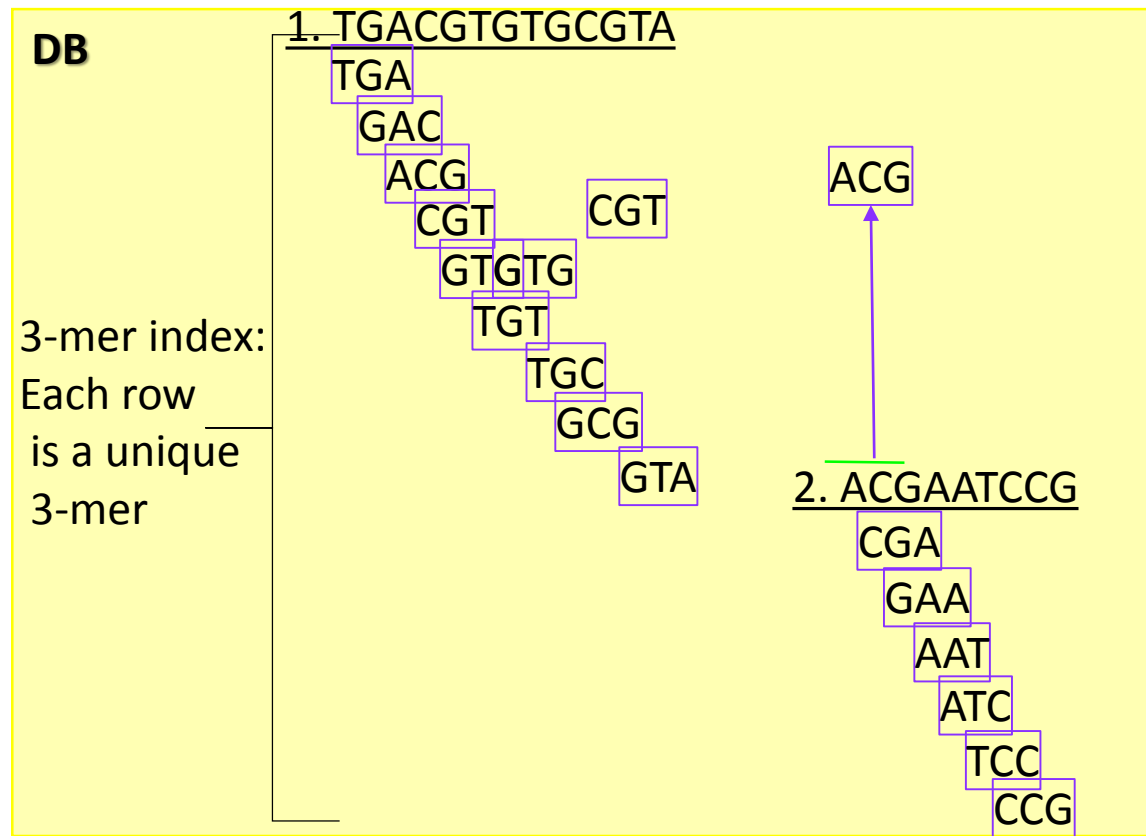
Assume all 3-mer are high-scoring words

- Process: The seed-index-map-extend-merge strategy

- Seed and index:

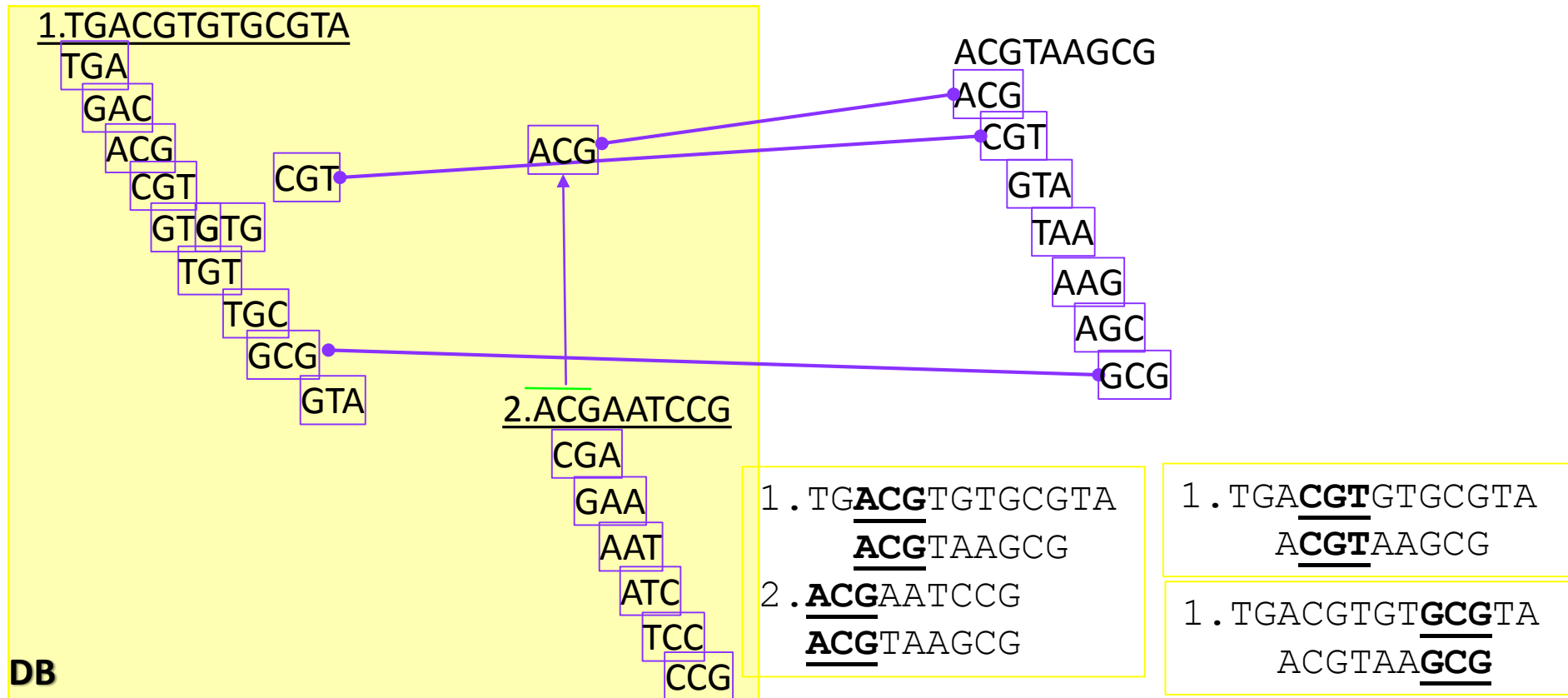
- Construct common words (k-mers) for sequences in a database
- Assume the database has two sequences and k=3
  - 1. TGACGTGTGCGTA
  - 2. ACGAATCCG

- Assume all 3-mers are high-score words
  - BLAST needs a threshold to determine high-score words.



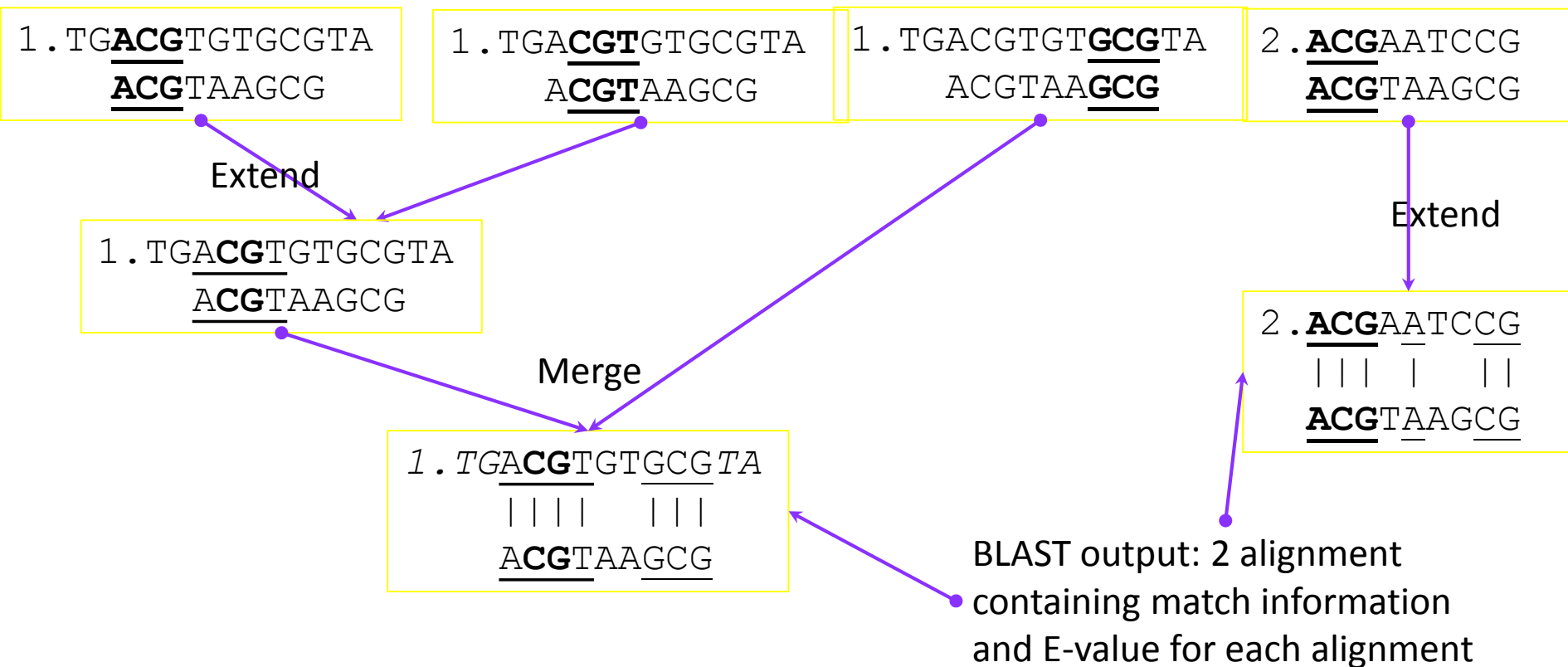
# BLAST

- Process: The seed-index-map-extend-merge strategy
  - Seed-map with high-score words
    - Obtain words from a query sequence



# BLAST

- Process: The seed-index-map-extend-merge strategy
  - Extend-merge
    - Extend: high-scoring segment



# Statistical significance of alignment

- How to assess the significance of a high-scoring hit to the database?
  - $E$  is the number of alignments expected by chance during a database search
  - $E$  is a function of the size of the search space ( $mn$ ), the normalized score ( $\lambda S$ ), and a constant ( $K$ ).
- Karlin-Altschul equation:  $E = Kmn e^{-\lambda S}$ 
  - Score ( $S$ ) from scoring matrix
  - $K$  and  $\lambda$  are two constants
  - $m$ : the length of a query sequence
  - $n$ : the sum of bases of all sequences in database
  - The lower  $E$  value indicates more significant alignment.



# Statistical significance of alignment

- How to assess the P-value of finding a high-scoring pair (HSP)?
  - The number of random HSPs with score  $\geq S$  is described by a Poisson distribution
  - the probability of finding exactly  $k$  HSPs with score  $\geq S$  is given by  $e^{-E} * E^k/k!$
  - Specifically the chance of finding zero HSPs with score  $\geq S$  is  $e^{-E}$ , so the probability of finding at least one such HSP is  $P=1-e^{-E}$
  - Conversely,  $E = -\ln(1 - P)$
  - When  $E$  and  $P$  are very small, they are almost identical

# Bowtie/BWA

- Ultrafast, memory-efficient alignment programs for aligning short DNA sequence reads to large genome
- Burrows–Wheeler transform (BWT)
  - Invented by Burrows and Wheeler, 1994
  - Is a block-sorting compression algorithm for many repeated characters
  - BWA and Bowtie are the most famous implementations of BWT in sequence alignment

# Burrows–Wheeler transform (BWT)

- BWT is an invertible transformation of a string  $S$  of length  $n$  into another string  $S'$  of length  $n$
- BWT can organize the genome string into a sequence of suffixes of the original genome string
- Given a string  $S$ , BWT
  - Add  $\$$  and assume  $\$$  < any alphabet
  - Obtains a suffix array of all cyclic rotations
  - Transform into a new string  $S'$

# BWT procedures

- Given a string  $S = \text{"TCATC"}$ , BWT
  - Adds \$ and assume  $\$ < \text{any alphabet}$
  - Obtains a suffix array of all cyclic rotations

Transformation				
1. Input	2. Cyclic rotations	3. Sorting	4. Last column	5. BWT output
TCATC\$				

# BWT procedures

- Cyclic rotation:
  - Obtains a suffix array of all cyclic rotations
  - Keeps an index of the rotated strings in the array
  - Creates Circular Permutation Table (CPT)

Transformation				
1. Input	2. Cyclic rotations	3. Sorting	4. Last column	5. BWT output
TCATC\$	0 TCATC\$ 1 CATC\$T 2 ATC\$TC 3 TC\$TCA 4 C\$TCAT 5 \$TCATC			

# BWT procedures

- **Sorting:**
  - Sorts the Circular Permutation Table (CPT) alphabetically
  - Keep the index with the strings

Transformation				
1. Input	2. Cyclic rotations	3. Sorting	4. Last column	5. BWT output
TCATC\$	0 TCATC\$ 1 CATC\$T 2 ATC\$TC 3 TC\$TCA 4 C\$TCAT 5 \$TCATC	5 \$TCATC 2 ATC\$TC 4 C\$TCAT 1 CATC\$T 3 TC\$TCA 0 TCATC\$		

# BWT procedures

- Taking the last column from the sorted array

Transformation				
1. Input	2. Cyclic rotations	3. Sorting	4. Last column	5. BWT output
TCATC\$	0 TCATC\$ 1 CATC\$T 2 ATC\$TC 3 TC\$TCA 4 C\$TCAT 5 \$TCATC	5 \$TCATC 2 ATC\$TC 4 C\$TCAT 1 CATC\$T 3 TC\$TCA 0 TCATC\$	5 \$TCATC 2 ATC\$TC 4 C\$TCAT 1 CATC\$T 3 TC\$TCA 0 TCATC\$	

# BWT procedures

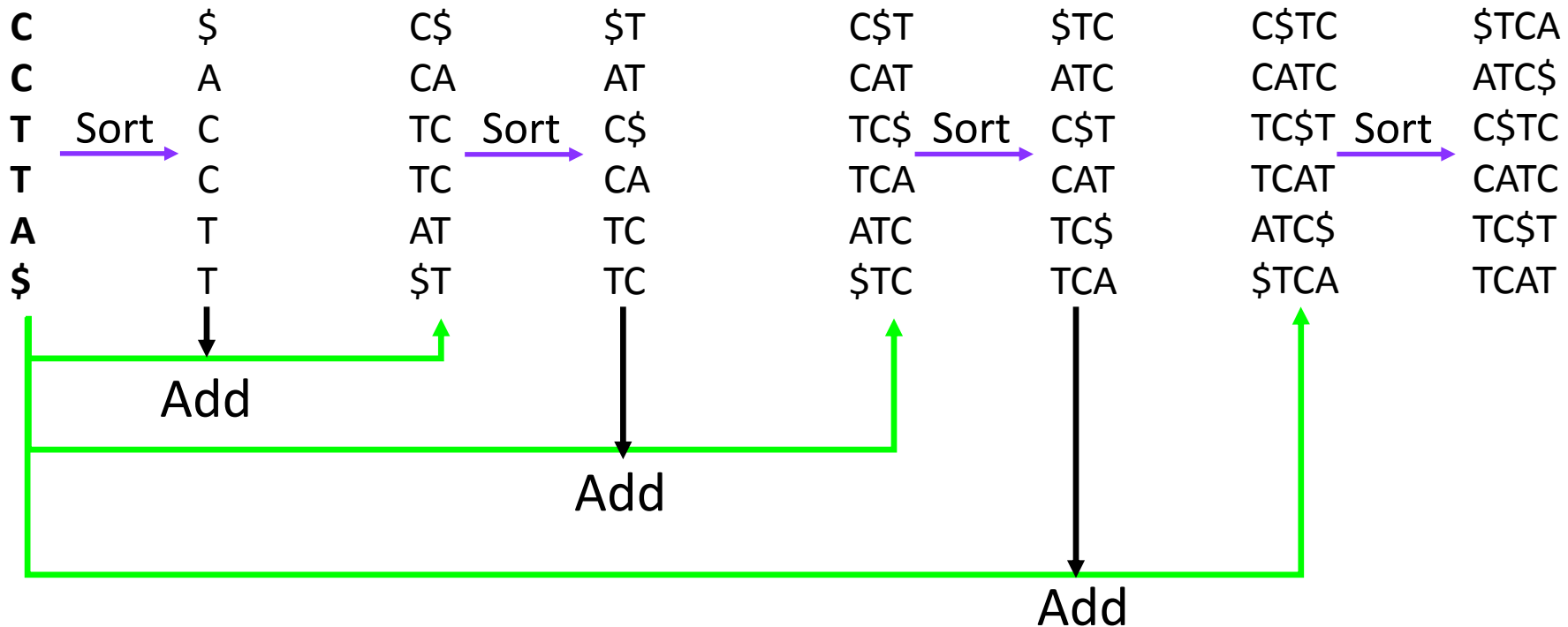
- BWT output:
  - The last column represents the transformed string

Transformation				
1. Input	2. Cyclic rotations	3. Sorting	4. Last column	5. BWT output
TCATC\$	0 TCATC\$ 1 CATC\$T 2 ATC\$TC 3 TC\$TCA 4 C\$TCAT 5 \$TCATC	5 \$TCATC 2 ATC\$TC 4 C\$TCAT 1 CATC\$T 3 TC\$TCA 0 TCATC\$	5 \$TCATC 2 ATC\$TC 4 C\$TCAT 1 CATC\$T 3 TC\$TCA 0 TCATC\$	CCTTA\$

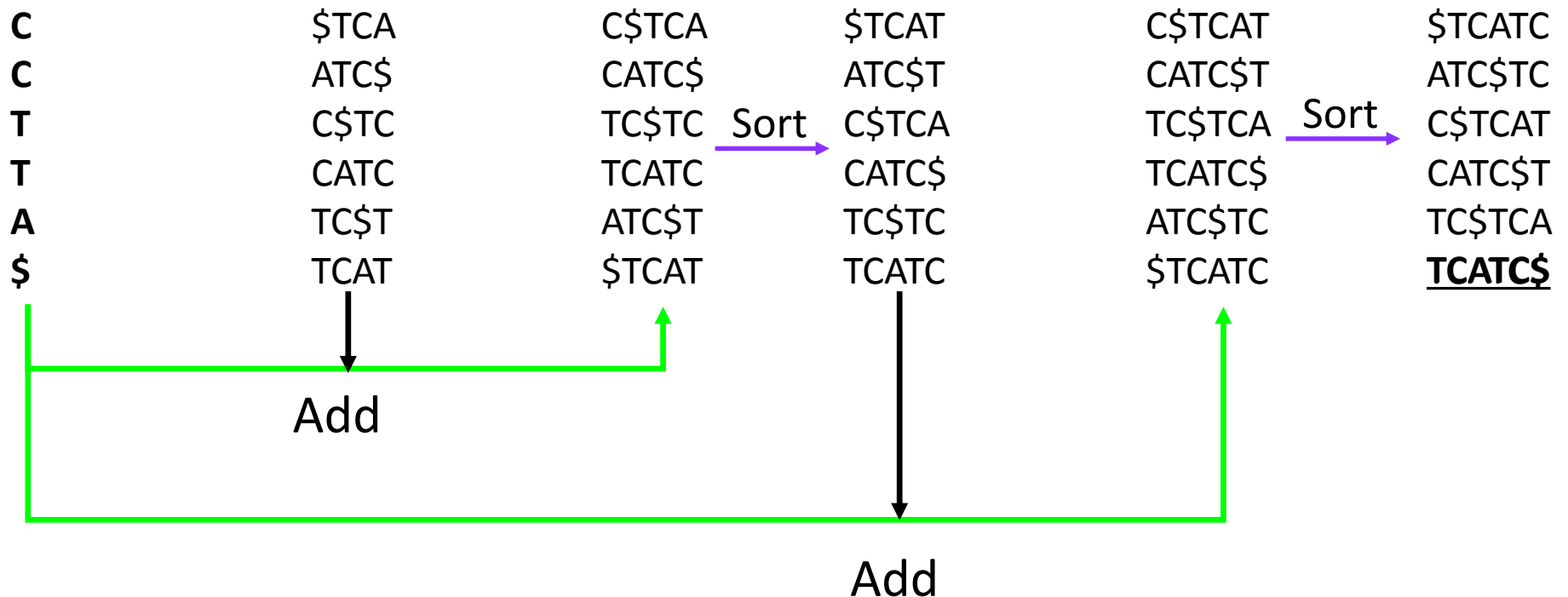


# BWT reverse transformation

- Transform back to the original string
  - With a suffix array, it is easy to recover the original string S
  - Input: CCTTA\$ for TCATC\$

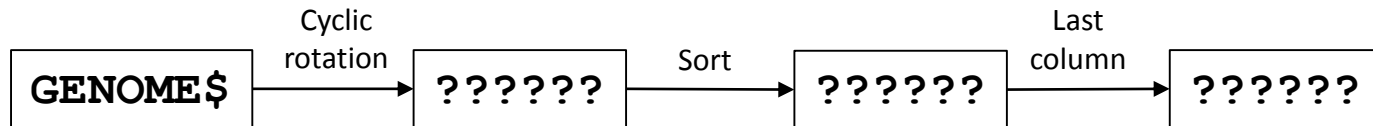


# BWT reverse transformation



# What's the BWT for GENOME?

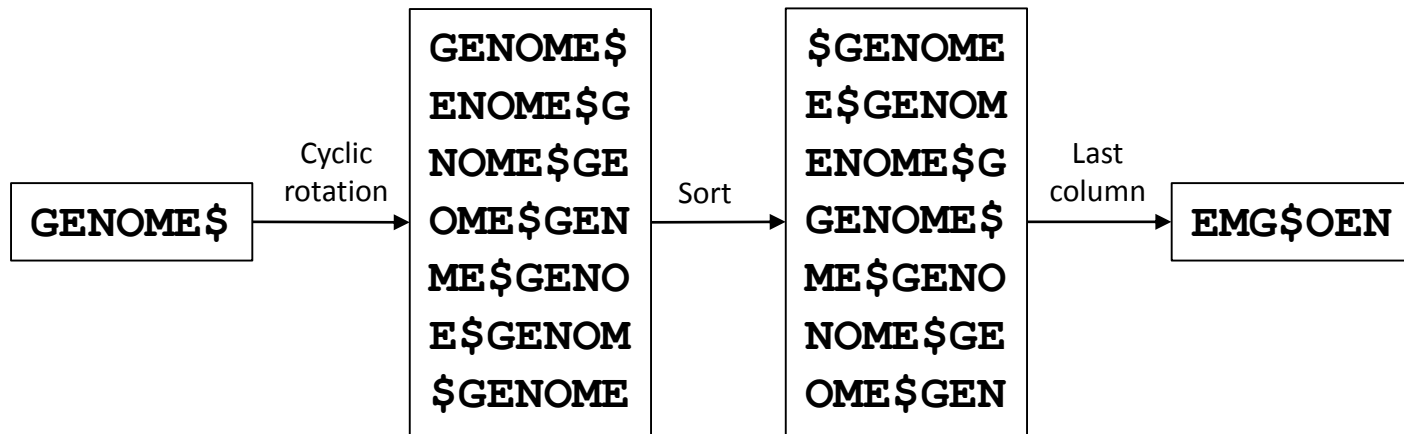
**BWT (GENOME\$) = ?**



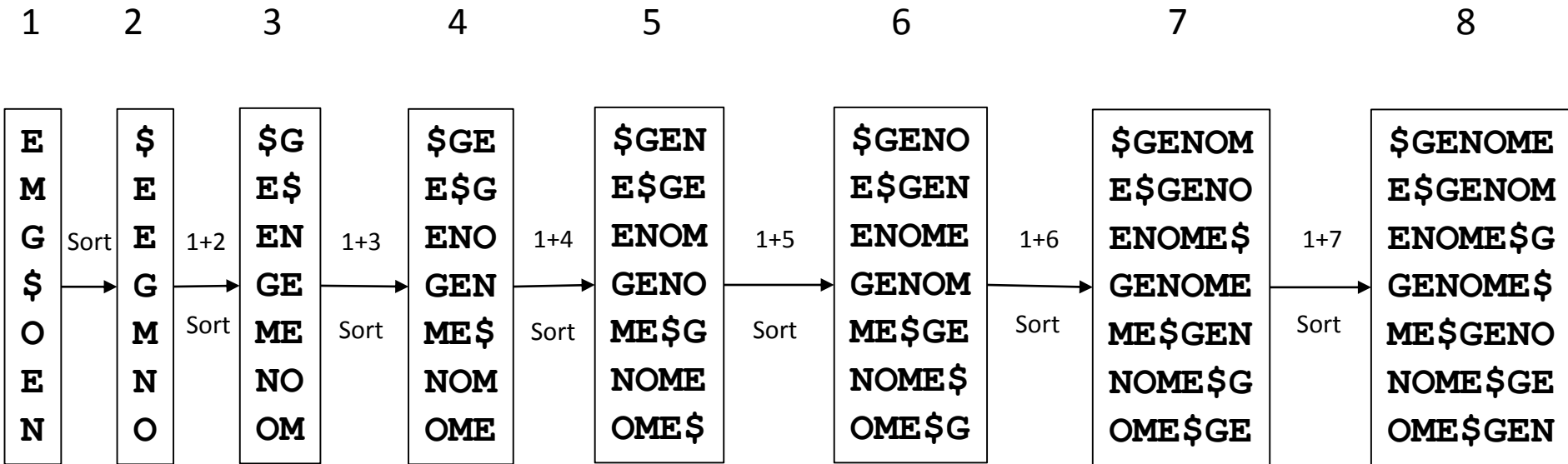
After class exercise: reproduce the BWT transformation above

# What's the BWT for GENOME\$?

**BWT (GENOME\$) = EMG\$OEN**



# BWT reverse transformation



After class exercise: reproduce the reverse transformation above

# BWT search

- Match process:
  - Given a query string, we want to find where is the matched sub-string in original string: TCATC\$
  - We do this search through the use of  $\mathcal{LF}$  functions on the BWT of the original string
  - $\mathcal{LF}$  (last to first) functions:
    - Purpose: find the same base in the first column corresponding to a specific base in the last column of BWT matrix

# LF property

- $\mathcal{LF}$  is a function mapping from the last column of the BWT matrix to the first column
- The  $i$ th occurrence of the character  $X$  in the last column of BWT matrix corresponds to the  $i$ th occurrence of  $X$  in the first column.
- Examples:
  - The 1<sup>st</sup> occurrence of C in the last column is the same C as the 1<sup>st</sup> occurrence of C in the first column
  - The 2<sup>nd</sup> occurrence of T in the last column is the same T as the 2<sup>nd</sup> occurrence of T in the first column

```
$TCATC
ATC$TC
C$TCAT
CATC$T
TC$TCA
TCATC$
```

# LF calculation

- $LF(pos, na) = C(na) + pre\_na(pos, na)$ 
  - pos: position in the output CCTTA\$
  - na: a base
  - $C(na)$ : the number of bases smaller than na in BWT first column
  - $pre\_na(pos, na)$ : the number of base na before the position of pos (not included).
- For TCATC (see right side), the  $C()$  for \$, A, C, T are 0, 1, 2, 4, respectively. This is pre-computed.
- For example,  $LF(3, 'T')$ 
  - $C('T') = 4$ ,  $pre\_na(3, 'T') = 1$
  - Then:  $LF(3, 'T') = 4+1 = 5$
  - The two underlined T in bold (one in the last column and one in the first column) as the same T in the original string.

0	\$TCATC
1	ATC\$TC
2	C\$TCAT
3	CATC\$ <b><u>T</u></b>
4	TC\$TCA
5	<b><u>T</u></b> CATC\$

T in the last column pos =3 is the same T in first column pos =5

Now, we can do the exercise on the C in the last column pos =1



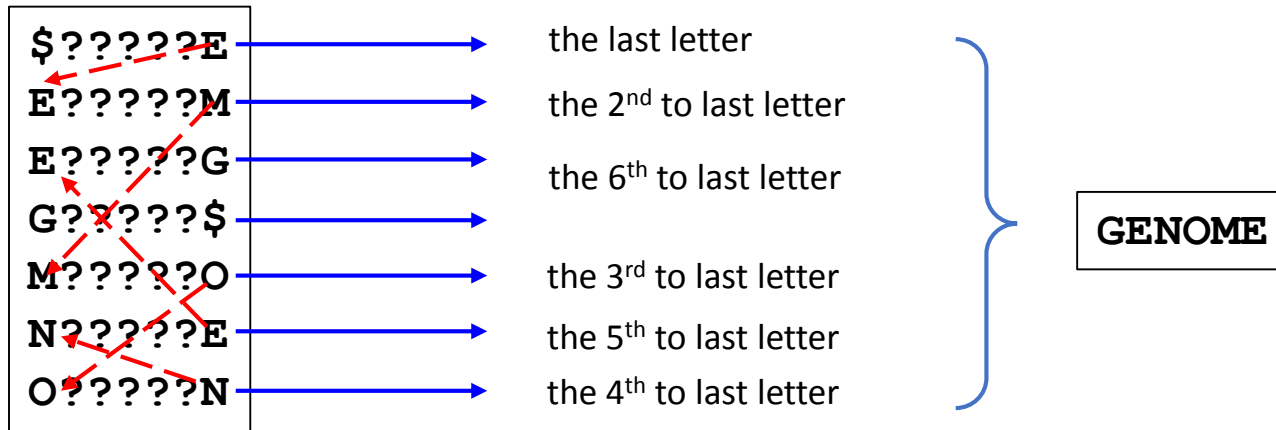
# How LF recovers original string

- Suppose we only know first/last column of BWT, what is the original string?

\$	?	?	?	?	?	E
E	?	?	?	?	?	M
E	?	?	?	?	?	G
G	?	?	?	?	?	\$
M	?	?	?	?	?	O
N	?	?	?	?	?	E
O	?	?	?	?	?	N

# How LF recovers original string

- Suppose we only know first/last column of BWT, what is the original string?



# BWT search

- Purpose: search a specific pattern (query) in the BWT matrix, and recover the index in original database sequence

- Example: search “AT” within the database sequence “TCATC” (the index of “AT” is 2)

0	1	2	3	4	5
T	C	A	T	C	\$

- Next we define the lower and upper bound of search recursively (from last to first base of query)
  - $L(W)$ : lowest index in BWT matrix where  $W$  is prefix
  - $U(W)$ : highest index in BWT matrix where  $W$  is prefix
  - The  $L()$  and  $U()$  for A, C, G, T can be pre-calculated from the BWT matrix
  - For a new prefix  $p$  in front of  $W$ , we will have:
    - $L(pW) = \text{LF}(L(W), p)$
    - $U(pW) = \text{LF}(U(W)+1, p)-1$

# BWT search

- Goal:
  - Original string: TCATC\$; BWT: CCTTA\$
  - Query: AT (find it in original string using BWT matrix)

- Search procedure:

- 1. for 'T' in "AT"

- $L(T) = 4$
    - $U(T) = 5$

- 2. for 'A' in "AT"

- $L(AT) = \text{LF}(L(T), 'A') = \text{LF}(4, 'A') = 1 + 0 = 1$
    - $U(AT) = \text{LF}(U(T) + 1, 'A') - 1 = \text{LF}(6, 'A') - 1 = 1 + 1 - 1 = 1$

- The matched row is: 1

- The index in original string is 2

0	1	2	3	4	5
T	C	A	T	C	\$

0	5	\$TCATC
1	2	ATC\$TC
2	4	C\$TCAT
3	1	CATC\$T
4	3	TC\$TCA
5	0	TCATC\$

# BWT search

- Goal:
  - Original string: TCATC\$; BWT: CCTTA\$
  - Query: TC (find it in original string using BWT matrix)


- Search procedure:


- 1. for 'C' in "TC"

- $L(C) = 2$  

- $U(C) = 3$  

- 2. for 'T' in "TC"

- $L(TC) = \text{LF}(L(C), 'T') = \text{LF}(2, 'T') = 4 + 0 = 4$  

- $U(TC) = \text{LF}(U(C)+1, 'T') - 1 = \text{LF}(4, 'T') - 1 = 4 + 2 - 1 = 5$  

- The matched row is: 4 and 5.

- The index in original string is 0 and 3.

0	1	2	3	4	5
T	C	A	T	C	\$

0	5	\$TCATC
1	2	ATC\$TC
2	4	C\$TCAT
3	1	CATC\$T
4	3	TC\$TCA
5	0	TCATC\$

# BWT search

- Goal:
  - Original string: TCATC\$; BWT: CCTTA\$
  - Query: AC (find it in original string using BWT matrix)

- Search procedure:

- 1. for 'C' in "AC"

- $L(C) = 2$

- $U(C) = 3$

- 2. for 'A' in "AC"

- $L(AC) = \text{LF}(L(C), 'A') = \text{LF}(2, 'A') = 1 + 0 = 1$

- $U(AC) = \text{LF}(U(C)+1, 'A') - 1 = \text{LF}(4, 'A') - 1 = 1 + 0 - 1 = 0$

- The matched row is None.

- "AC" is not in "TCATC"

0	5	\$TCATC
1	2	ATC\$TC
2	4	C\$TCAT
3	1	CATC\$T
4	3	TC\$TCA
5	0	TCATC\$

# BWT search

- Goal:

- Original string: TCATC\$; BWT: CCTTA\$
- Query: ATC (find it in original string using BWT matrix)

- Search procedure:

- 1. for 'C' in "ATC"

- $L(C) = 2$

- $U(C) = 3$

- 2. for 'T' in "ATC"

- $L(TC) = \text{LF}(L(C), 'T') = \text{LF}(2, 'T') = 4 + 0 = 4$

- $U(TC) = \text{LF}(U(C)+1, 'T') - 1 = \text{LF}(4, 'T') - 1 = 4 + 2 - 1 = 5$

- 3. for "A" in "ATC"

- $L(ATC) = \text{LF}(4, 'A') = 1 + 0 = 1$

- $U(ATC) = \text{LF}(6, 'A') - 1 = 1 + 1 - 1 = 1$

- The matched row is 1

- The original index is 2

0	1	2	3	4	5
T	C	A	T	C	\$

0	5	\$TCATC
1	2	ATC\$TC
2	4	C\$TCAT
3	1	CATC\$T
4	3	TC\$TCA
5	0	TCATC\$

# Minimap2

- Fast and accurate aligner for whole genome sequencing data against a reference genome
  - Can work with both short reads and noisy long reads
  - A typical seed-chain-alignment strategy
    - 1. Collect minimizers of reference sequence
    - 2. Index in a hash table
    - 3. Get minimizers from query sequences
    - 4. Find exact match as anchors
    - 5. Find collinear anchors
    - 6. Extend or close gaps by dynamic programming



# Minimap2

- K-mer based sequence similarity
  - BLAST:
    - Get k-mer, generate hash value and store in a hash table
    - Find k-mer match between reference sequence and query sequence
  - DALIGNER(Myers. WABI 2014; 8701:52-67):
    - Generate k-mer for each of two sets of reads
    - Sort k-mers and merge them for potential match
  - MHAP(Berlin et al. Nat. Biotechnol. 2015;33:623-630):
    - $m$  k-mer hash functions
    - For each hash functions, find minimum hash value for all k-mers in a sequence
    - Two sequences are similar to each other if they have many overlaps of minimum hash value
- Minimap used hash functions, hash table, and sorting

# The concept of minimizer

- Chooses a representative k-mer from a group of adjacent k-mers, so different strings  $T_i$  and  $T_j$  choose the same representative if they share a long enough subsequence.
- Only a small fraction of k-mers, called ‘minimizers’, needs to be stored.

Position	1	2	3	4	5	6	7	1	2	3	4	5	6	7	8	9	10	11	12
Sequence	2	3	1	0	3	4	3	4	2	6	4	7	2	8	1	4	7	5	1
<i>k</i> -mers	2	3	1					4	2	6	4	7	2	8					
with		3	1	0					<b>2</b>	<b>6</b>	<b>4</b>	<b>7</b>	<b>2</b>	<b>8</b>	<b>1</b>				
minimizer			1	0	3					6	4	7	2	8	1	4			
in				<b>0</b>	<b>3</b>	<b>4</b>					4	7	2	8	1	4	7		
<b>bold</b>					3	4	3					7	2	8	1	4	7	5	
	(a)							(b)					2	8	1	4	7	5	1

# Minimap2

- 1. Collect minimizers of reference sequence

- Minimizer:

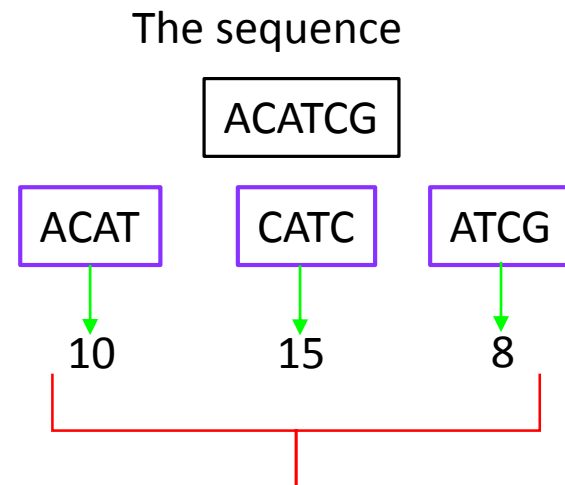
- A smallest  $k$ -mer of  $w$  consecutive  $k$ -mers in a sequence with  $w+k$  bases
    - Minimap2:  $k=15$  and  $w=5$

- Example:

- Assume  $w=3$  and  $k=4$

three  $k$ -mer (4-mer)

hash value



The minimizer of ACATCG is 8

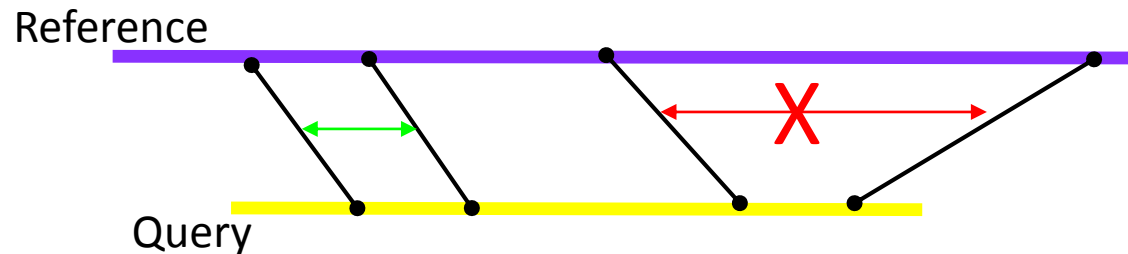
- Given a sequence with  $>w+k$  bases
    - A set of minimizers were generated

# Minimap2

- 2. Index in a hash table
  - Key: a hash value of a minimizer
  - Values: list of locations of minimizer in the reference genome
- 3. Get minimizers from query sequences
  - Used the same hash function to obtain a set of minimizers for query sequence
- 4. Find exact match as anchors
  - Match minimizers between reference and query sequence
  - The matched sub-sequence of reference and sub-sequence of query: anchors

# Minimap2

- 5. Find collinear anchors
  - Two anchors are collinear
    - Their distance in a sequence is less than a threshold.
    - Forward and reverse strands are considered individually
    - Cluster close anchors.



- 6. Extend or close gaps between anchors by dynamic programming