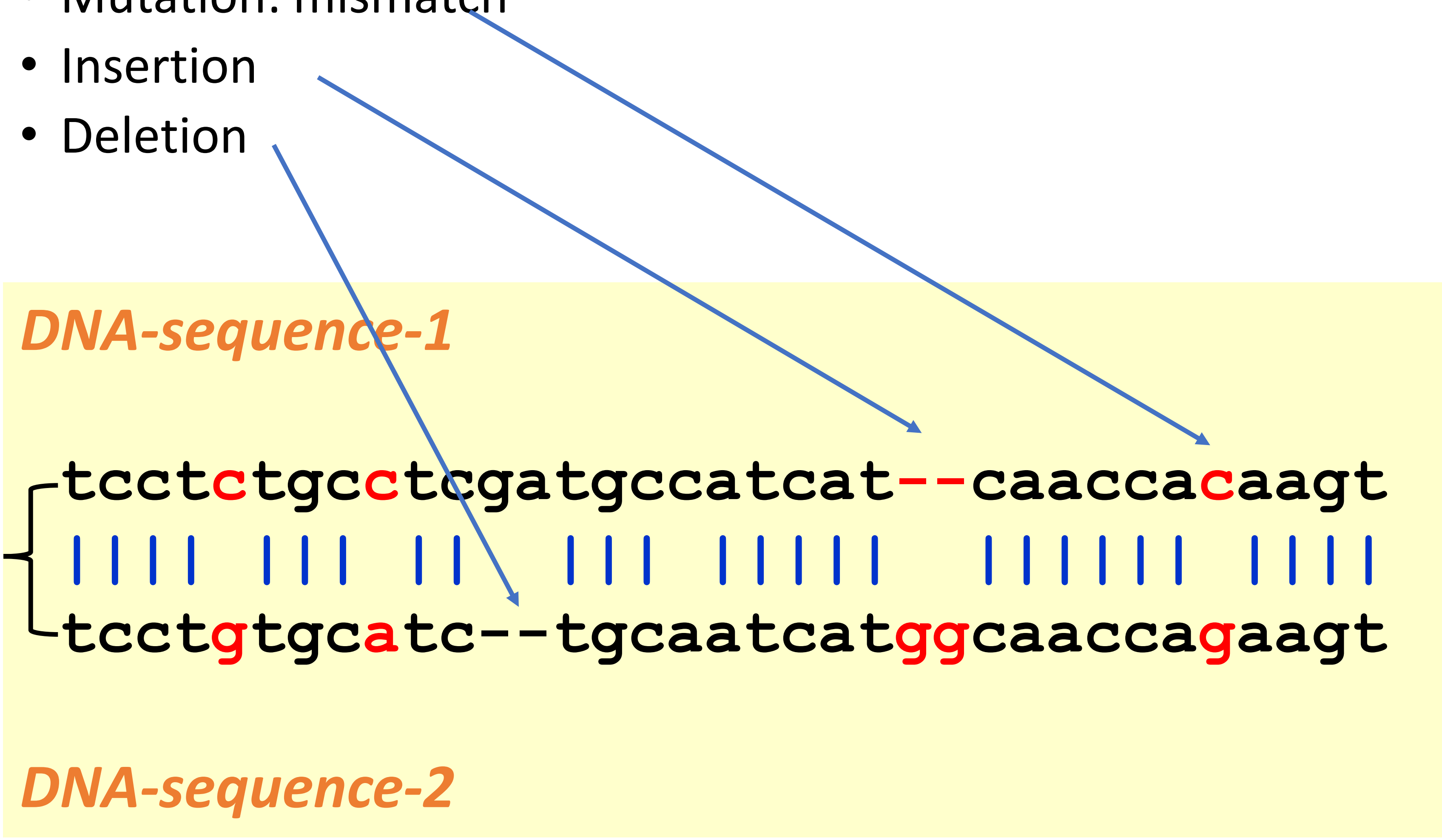
Alignment of short/long-read sequencing data

2019 Dragon Star Bioinformatics Course (Day 2)

Sequence similarity

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



DNA-sequence-1

```
tcctctgcctcgatgccatcat--caaccacaagt
|||| ||| ||  ||| |||||  |||||| ||||
tcctgtgcatc--tgcaatcatggcaaccagaagt
```

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

	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

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

Needelman & Wunsch, J Mol Biol, 1970

Global alignment: Needleman-Wunsch

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

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

Q

P

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

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

Q

P

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

Q

P

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

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

Global alignment: Needleman-Wunsch

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

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

Q

P

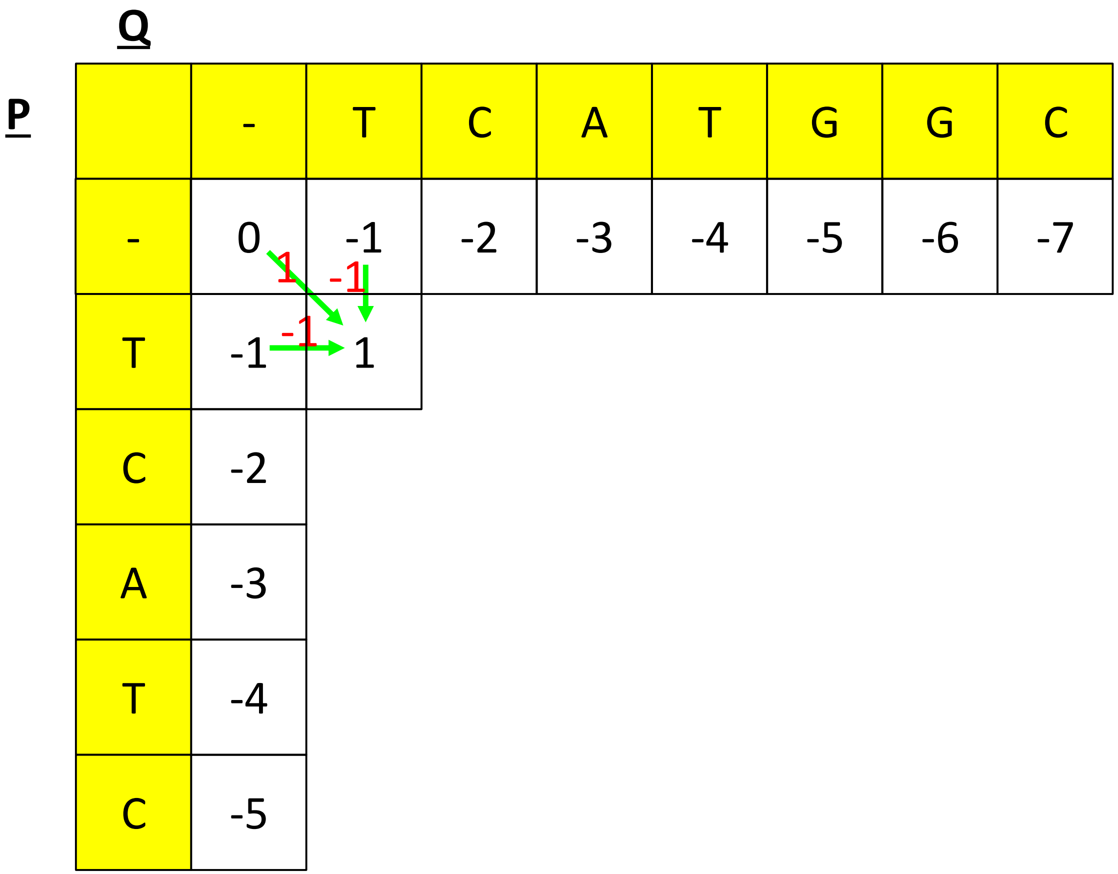
	-	T	C	A	T	G	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
T	-1							
C	-2							
A	-3							
T	-4							
C	-5							

1

-1

Global alignment: Needleman-Wunsch

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



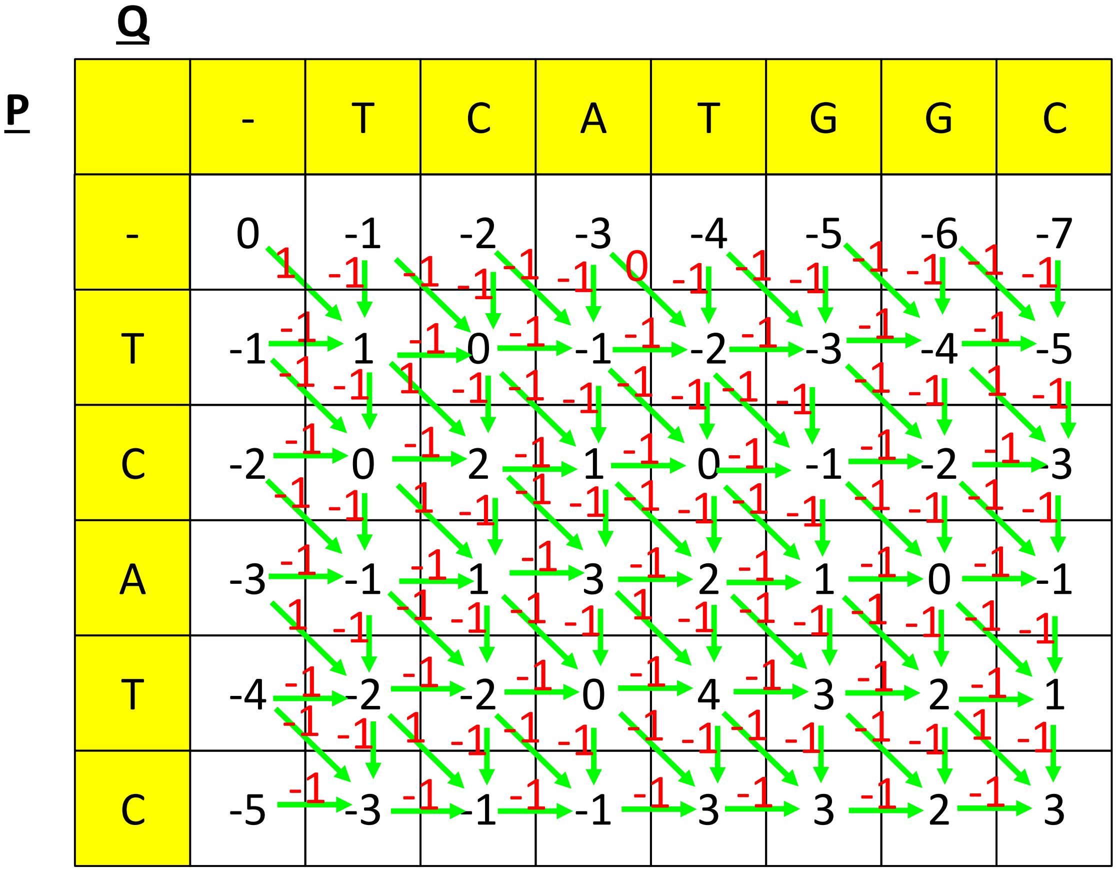
	-	T	C	A	T	G	G	C
-	0	-1	-2	-3	-4	-5	-6	-7
T	-1	1						
C	-2							
A	-3							
T	-4							
C	-5							

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

Global alignment: Needleman-Wunsch

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

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



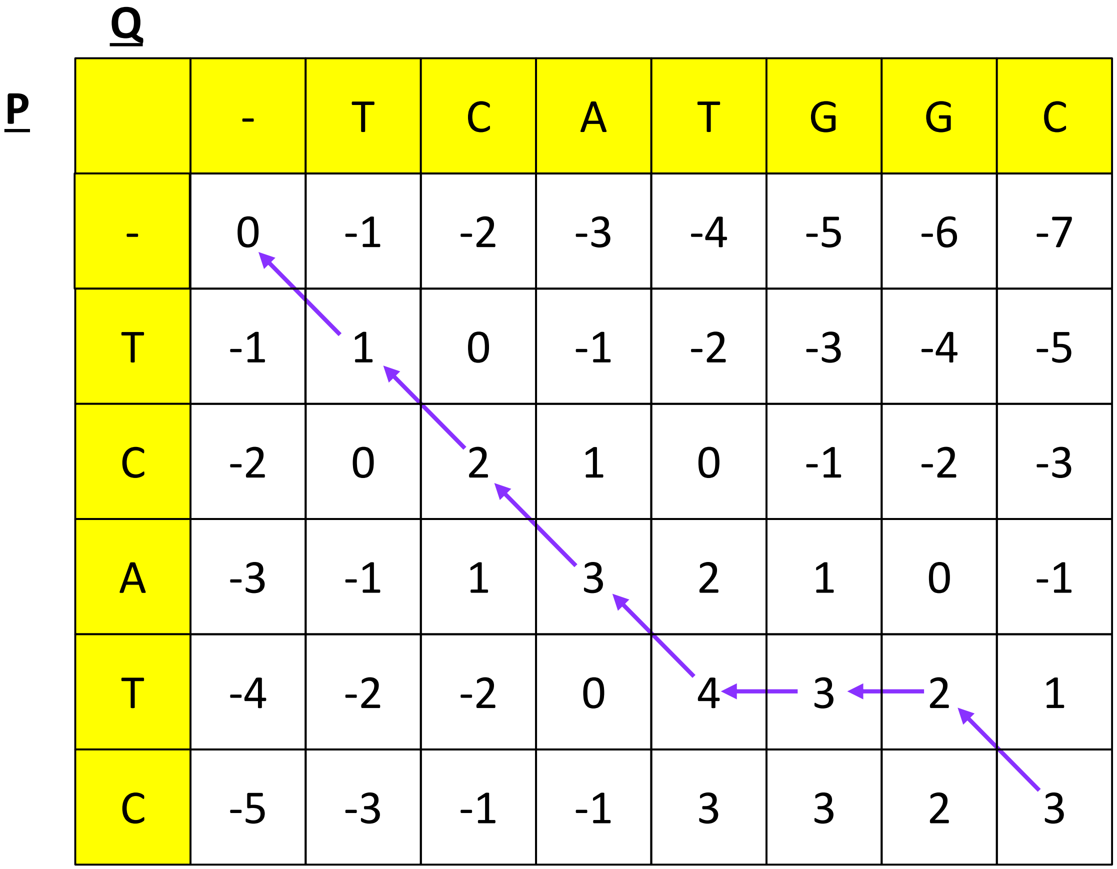
Q

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

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

```
T C A T  G G C
| | | |      |
T C A T  - - 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

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

Smith & Waterman, J Mol Biol, 1981

Local alignment: Smith-Waterman

- 1. Tabular arrangement
  - $C(0, j) = 0$
  - $C(i, 0) = 0$

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

Q

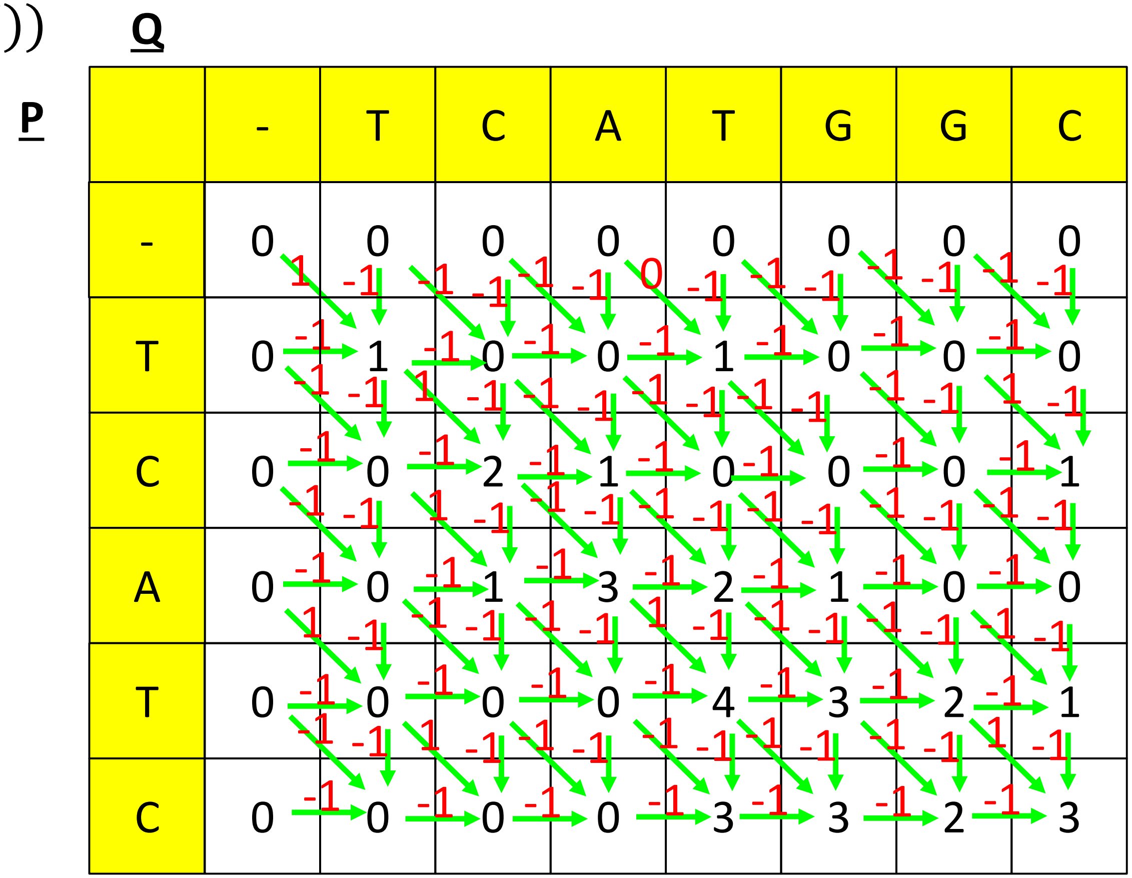
P

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

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



Q

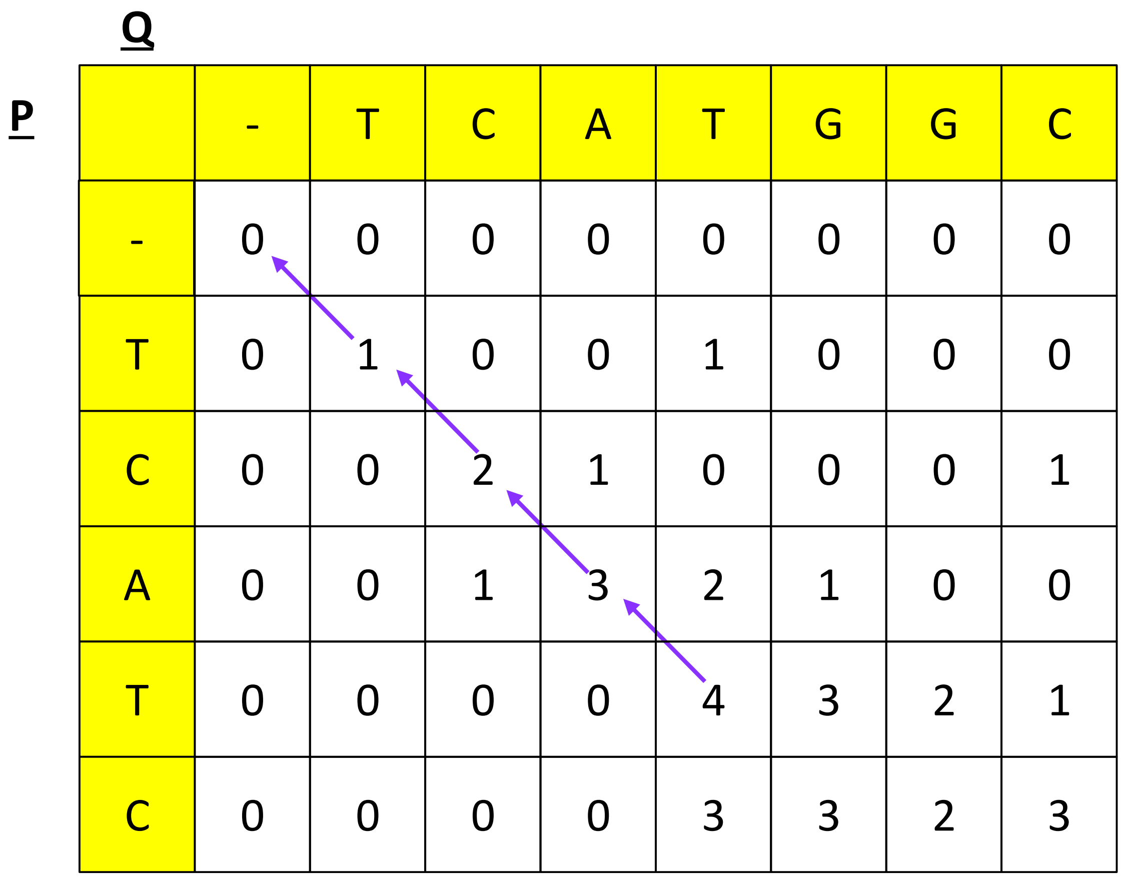
P

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

Local alignment: Smith-Waterman

- 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

```
TCAT
||||
TCAT
```

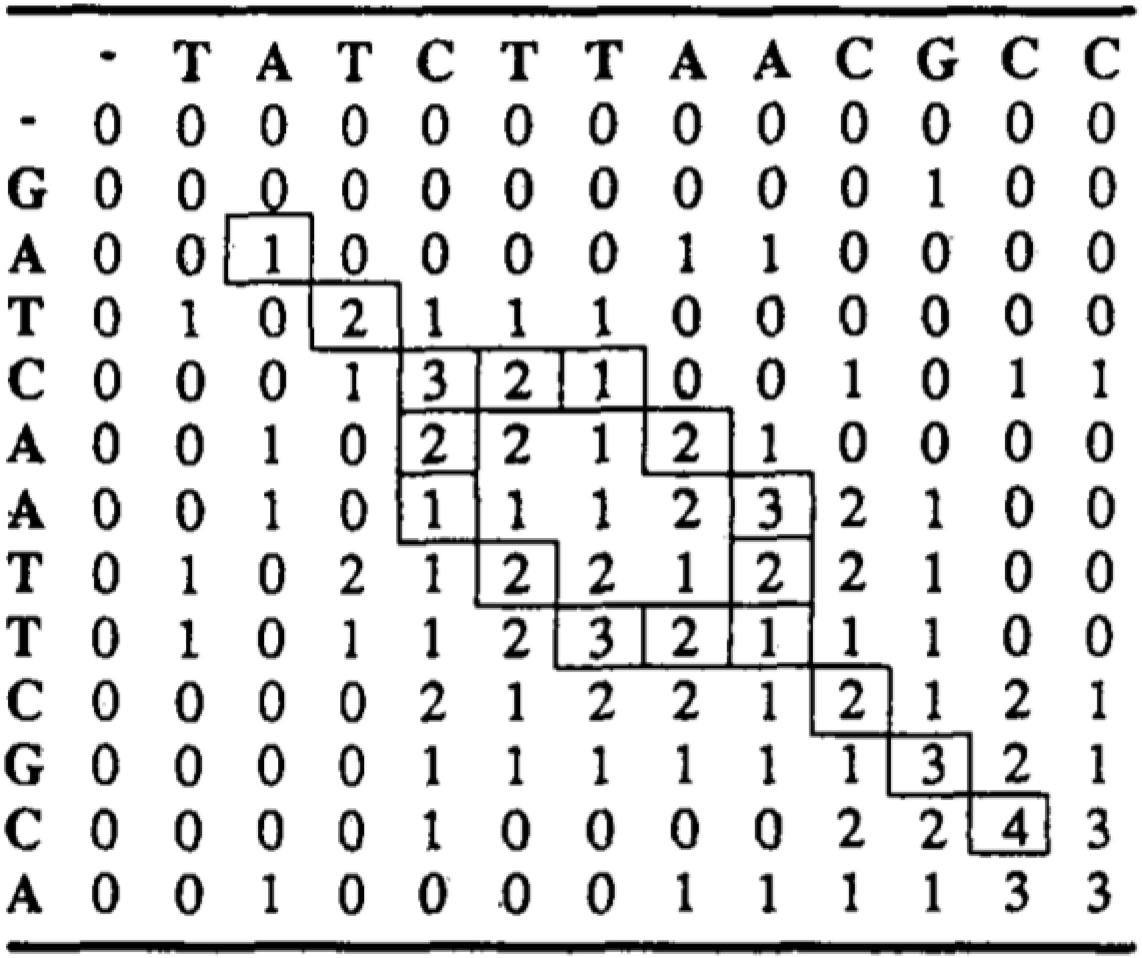


Q

P

P \ Q	-	T	C	A	T	G	G	C
-	0	0	0	0	0	0	0	0
T	0	1	0	0	1	0	0	0
C	0	0	2	1	0	0	0	1
A	0	0	1	3	2	1	0	0
T	0	0	0	0	4	3	2	1
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

Waterman & Vingron. Statistical science 1994:9(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

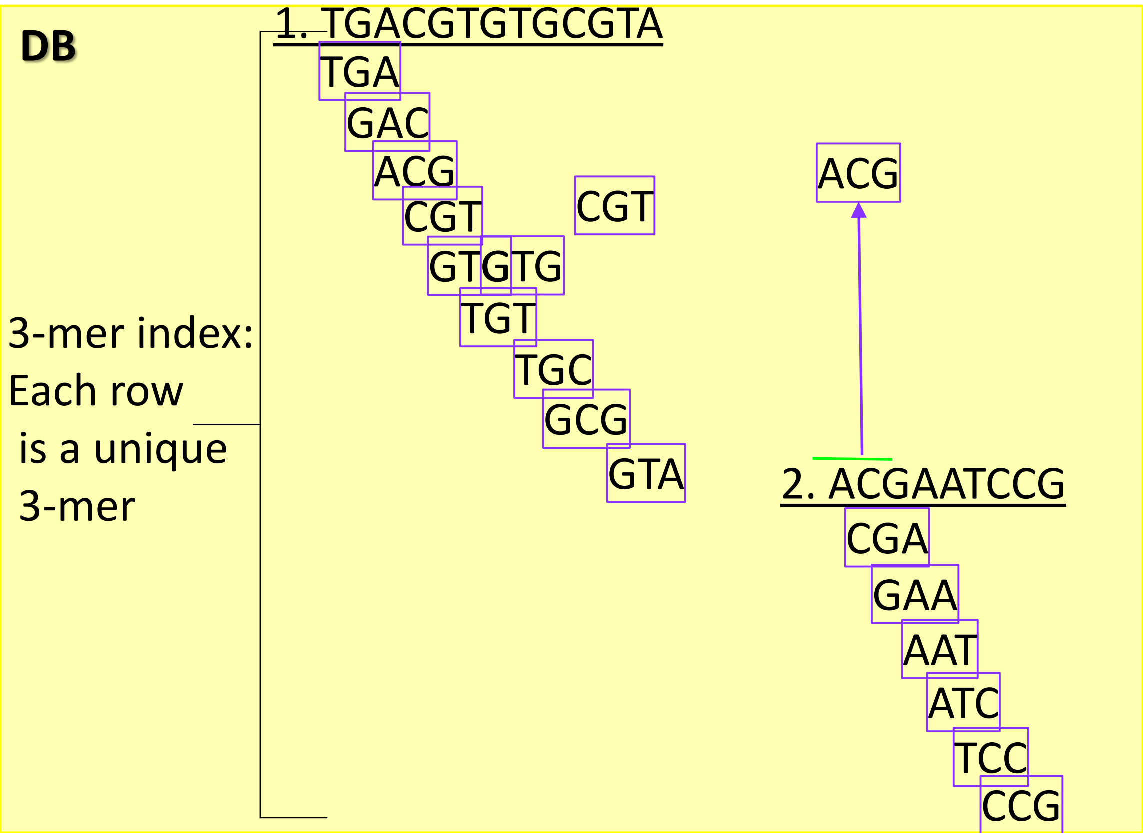
Altschul et al, J Mol Biol. 1990; Altschul et al, Nucleic Acids Res, 1997

BLAST

Assume all 3-mer are high-scoring words

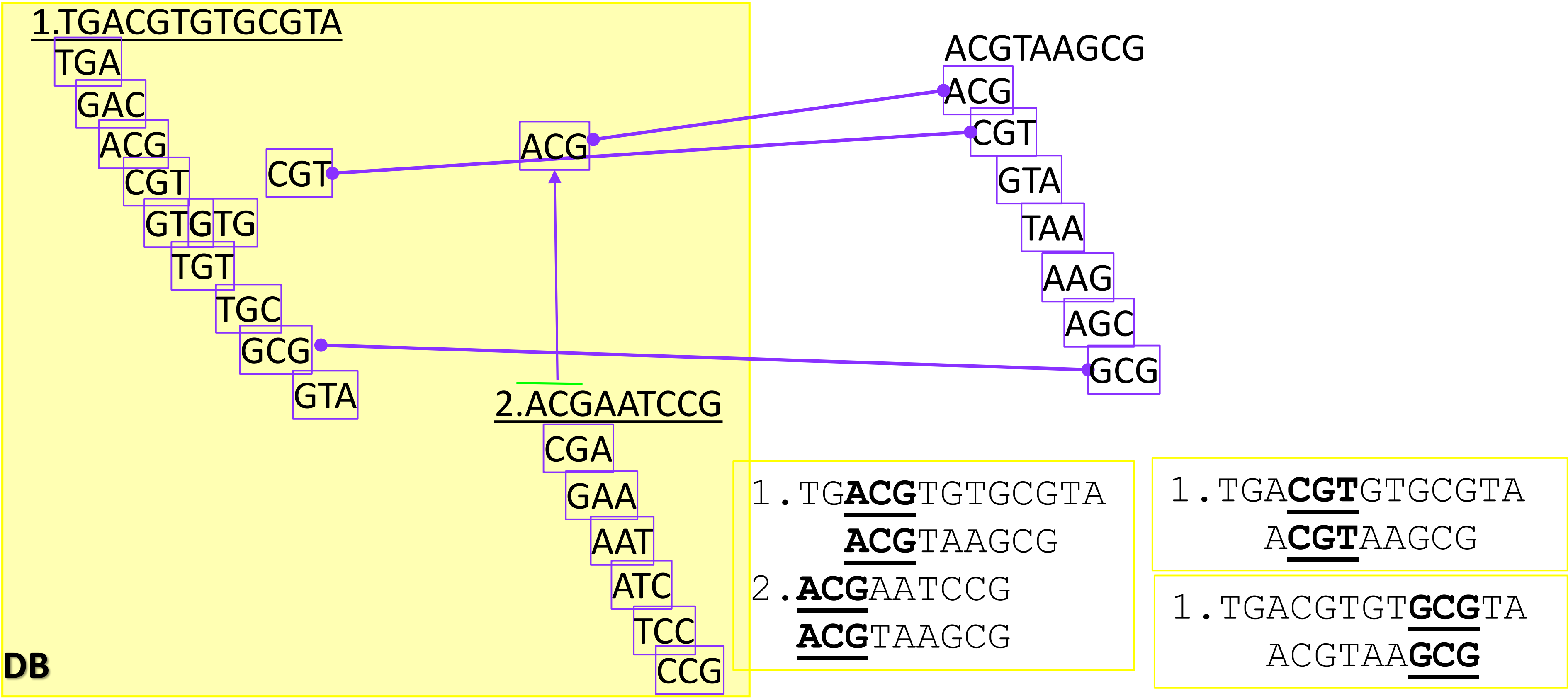
- Process: The seed-index-map-extend-merge strategy
  - Seed and index:
    - Construct common words (k-mers) for sequences in a database
    - Assume the database has two sequences and k=3
      - 1. TGACGTGTGCGTA
      - 2. ACGAATCCG

- 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



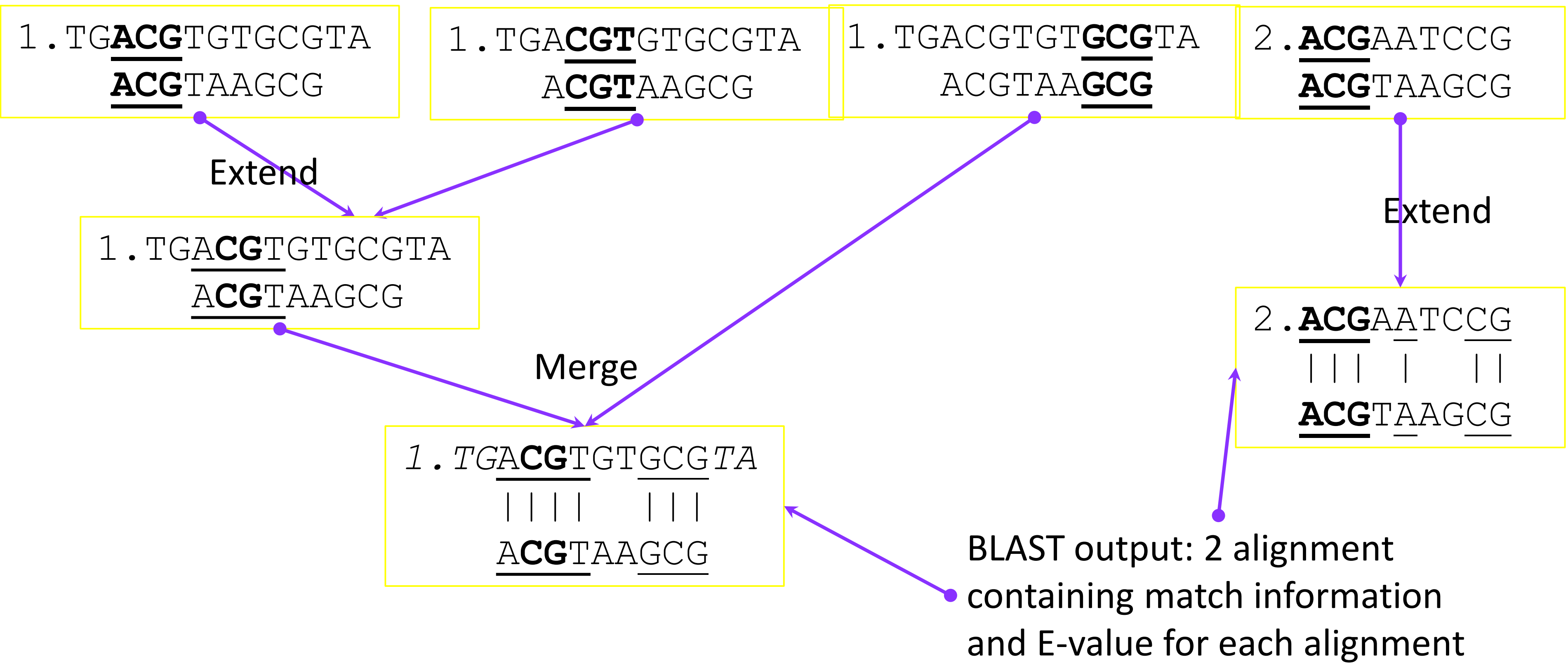
```
1.TG**ACG**TGTGCGTA
    **ACG**TAAGCG
2.**ACG**AATCCG
  **ACG**TAAGCG
```

```
1.TGA**CGT**GTGCGTA
    A**CGT**AAGCG
```

```
1.TGACGTGT**GCG**TA
    ACGTAA**GCG**
```

BLAST

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



1.TG**ACG**TGTGCGTA
ACGTAAGCG

1.TGA**CGT**GTGCGTA
A**CGT**AAGCG

1.TGACGTGT**GCG**TA
ACGTAA**GCG**

2.**ACG**AATCCG
ACGTAAGCG

Extend

1.TGA**CG**TGTGCGTA
A**CG**TAAGCG

Extend

2.**ACG**AATCCG
||| | ||
ACGTAAGCG

Merge

*1.TG*A**CG**TGTGCG*TA*
|||| |||
A**CG**TAAGCG

BLAST output: 2 alignment containing match information and E-value for each alignment

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 (λS), and a constant (K).
- Karlin-Altschul equation: $E = Kmne^{-\lambda S}$
  - Score (S) from scoring matrix
  - K and λ 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 >= *S* is described by a Poisson distribution
    - the probability of finding exactly *k* HSPs with score >=*S* is given by e^{-E} * E^{k}/k!
    - Specifically the chance of finding zero HSPs with score >=*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 = “TCATC”, BWT
  - Adds $ and assume $<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 $TCAT**C** 2 ATC$T**C** 4 C$TCA**T** 1 CATC$**T** 3 TC$TC**A** 0 TCATC**$**	

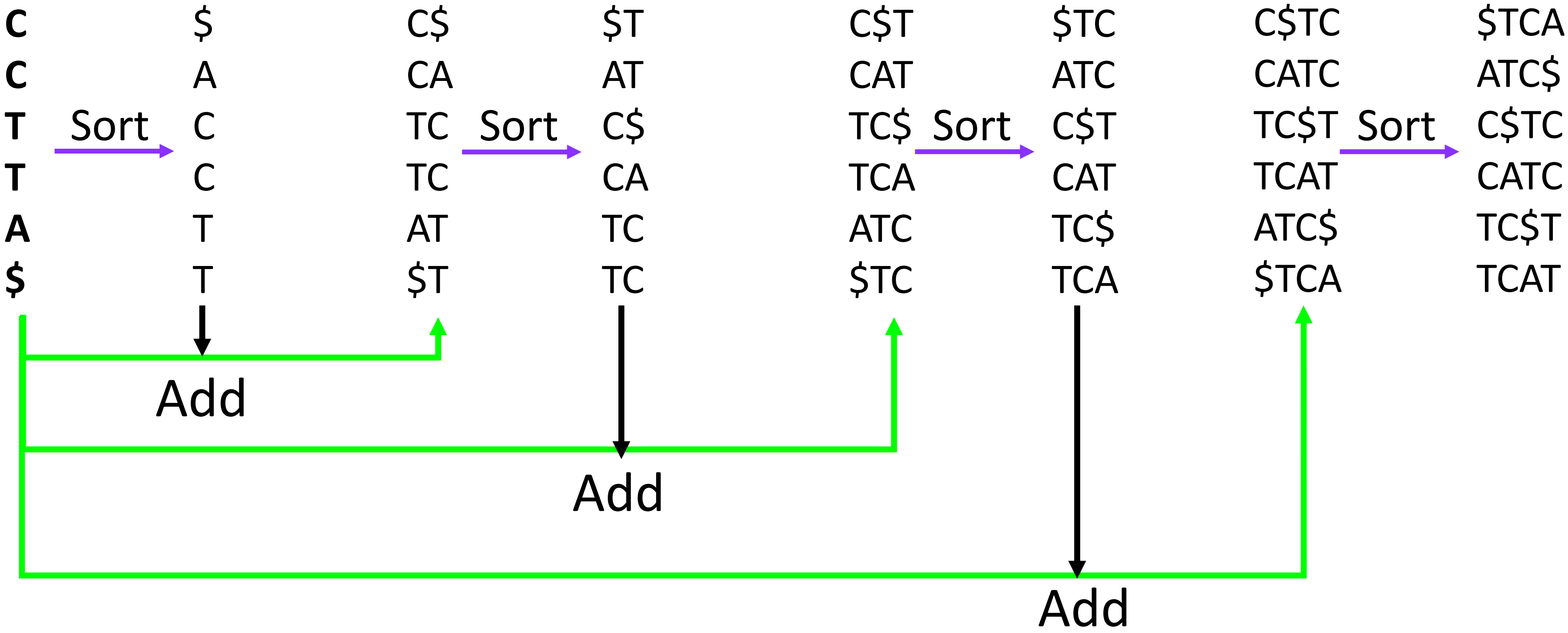
BWT procedures

- 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 $TCAT**C** 2 ATC$T**C** 4 C$TCA**T** 1 CATC$**T** 3 TC$TC**A** 0 TCATC**$**	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$` for `TCATC$`



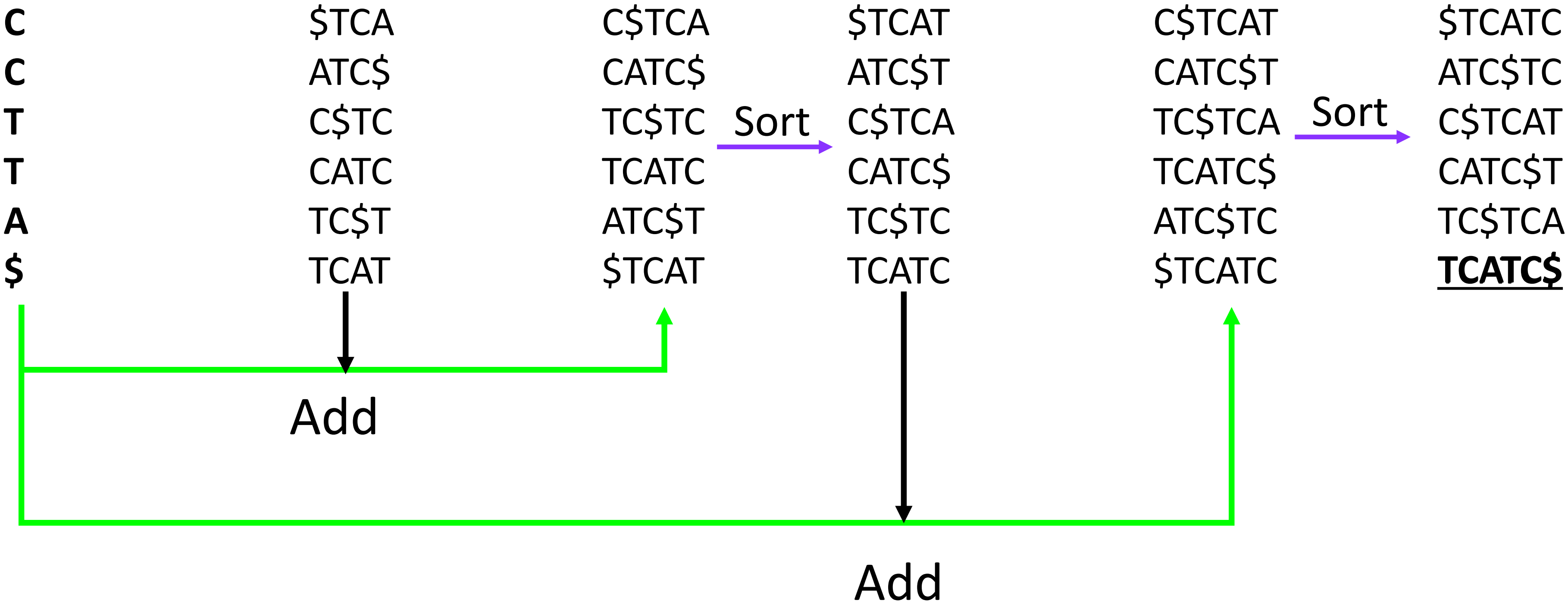
C		$	C$		$T	C$T		$TC	C$TC		$TCA
C		A	CA		AT	CAT		ATC	CATC		ATC$
T	Sort	C	TC	Sort	C$	TC$	Sort	C$T	TC$T	Sort	C$TC
T		C	TC		CA	TCA		CAT	TCAT		CATC
A		T	AT		TC	ATC		TC$	ATC$		TC$T
$		T	$T		TC	$TC		TCA	$TCA		TCAT

Add

Add

Add

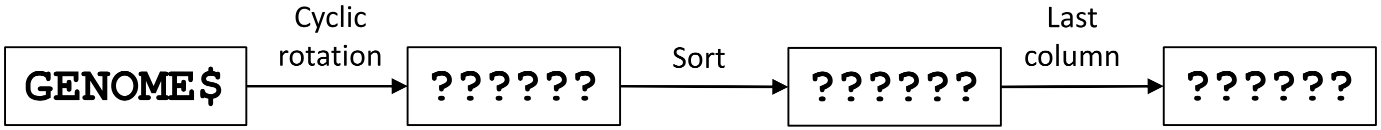
BWT reverse transformation



C	$TCA	C$TCA	$TCAT	C$TCAT	$TCATC
C	ATC$	CATC$	ATC$T	CATC$T	ATC$TC
T	C$TC	TC$TC	C$TCA	TC$TCA	C$TCAT
T	CATC	TCATC	CATC$	TCATC$	CATC$T
A	TC$T	ATC$T	TC$TC	ATC$TC	TC$TCA
$	TCAT	$TCAT	TCATC	$TCATC	**TCATC$**

What’s the BWT for GENOME?

BWT (GENOME$) = ?



GENOME$

Cyclic rotation

??????

Sort

??????

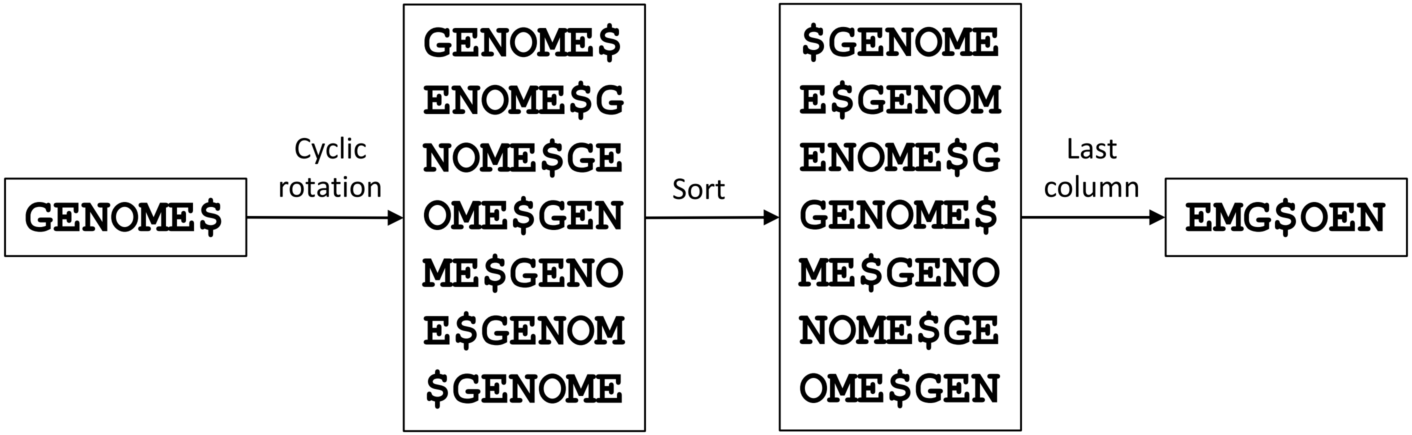
Last column

??????

After class exercise: reproduce the BWT transformation above

What’s the BWT for GENOME?

BWT (GENOME$) = EMG$OEN



GENOME$

Cyclic rotation

GENOME$
ENOME$G
NOME$GE
OME$GEN
ME$GENO
E$GENOM
$GENOME

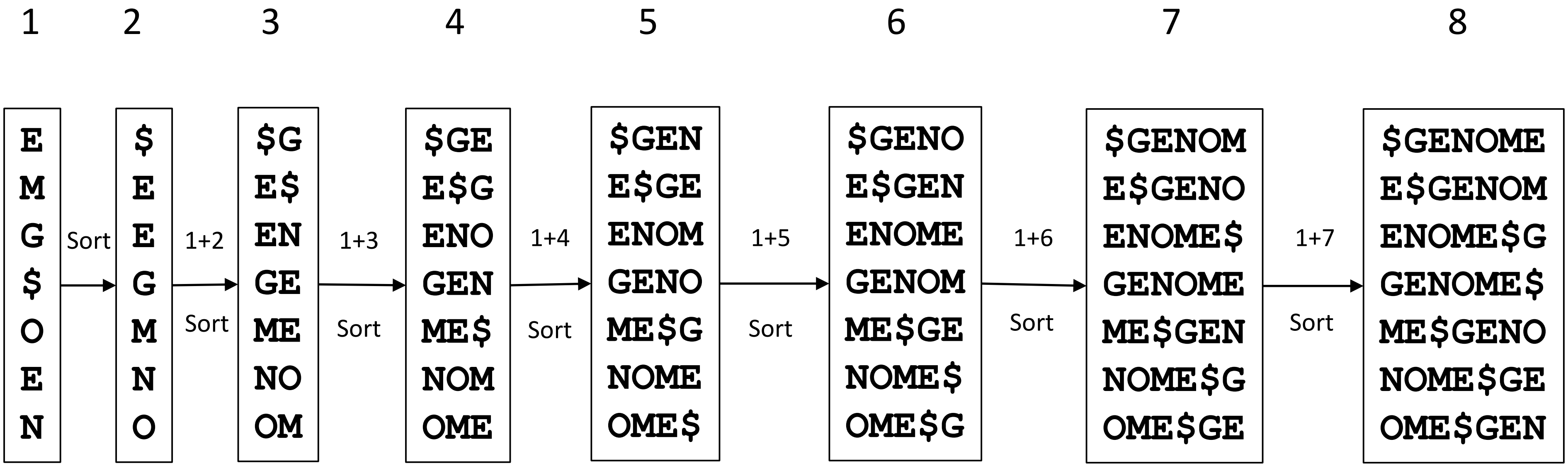
Sort

$GENOME
E$GENOM
ENOME$G
GENOME$
ME$GENO
NOME$GE
OME$GEN

Last column

EMG$OEN

BWT reverse transformation



1	2	3	4	5	6	7	8
E	$	$G	$GE	$GEN	$GENO	$GENOM	$GENOME
M	E	E$	E$G	E$GE	E$GEN	E$GENO	E$GENOM
G	E	EN	ENO	ENOM	ENOME	ENOME$	ENOME$G
$	G	GE	GEN	GENO	GENOM	GENOME	GENOME$
O	M	ME	ME$	ME$G	ME$GE	ME$GEN	ME$GENO
E	N	NO	NOM	NOME	NOME$	NOME$G	NOME$GE
N	O	OM	OME	OME$	OME$G	OME$GE	OME$GEN

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 $\mathbb{LF}$ functions on the BWT of the original string
    - $\mathbb{LF}$ (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

- LF is a function mapping from the last column of the BWT matrix to the first column
- The ith occurrence of the character X in the last column of BWT matrix corresponds to the ith occurrence of X in the first column.
- Examples:
  - The 1st occurrence of C in the last column is the same C as the 1st occurrence of C in the first column
  - The 2nd occurrence of T in the last column is the same T as the 2nd 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$TCAT
3	CATC$**T**
4	TC$TCA
5	**T**CATC$

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

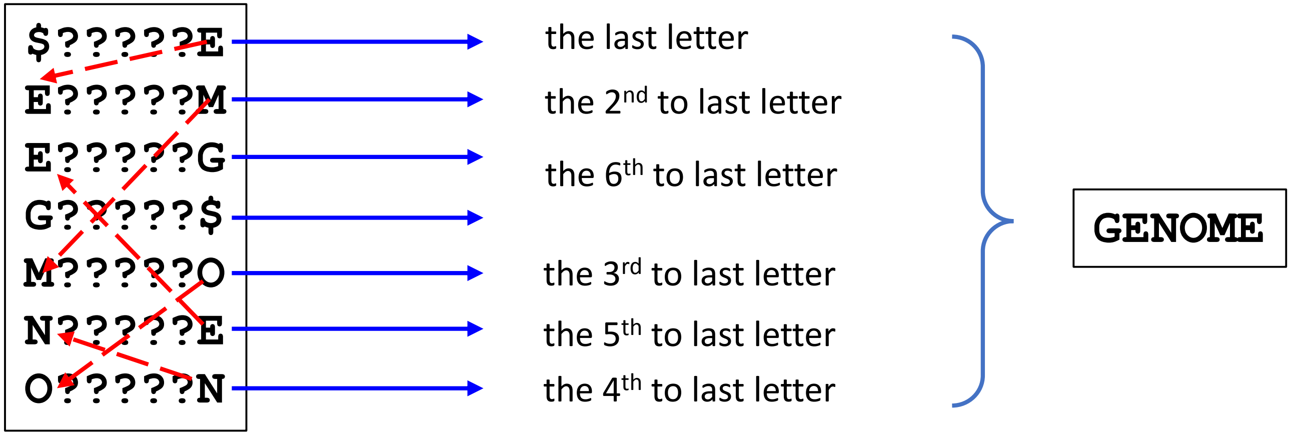
How LF recovers original string

- 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?



```
$?????E
E?????M
E?????G
G?????$
M?????O
N?????E
O?????N
```

the last letter

the 2nd to last letter

the 6th to last letter

the 3rd to last letter

the 5th to last letter

the 4th to last letter

GENOME

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:
    - $L(pW) = \mathbb{LF}(L(W), p)$
    - $U(pW) = \mathbb{LF}(U(W)+1, p)-1$

BWT search

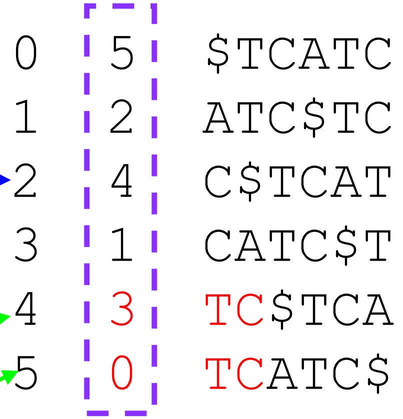
- Goal:
  - Original string: `TCATC$`; BWT: `CCTTA$`
  - Query: AT (find it in original string using BWT matrix)
- Search procedure:
  - 1. for ‘T’ in “AT”
    - L(T) = 4
    - U(T) = 5
  - 2. for ‘A’ in “AT”
    - L(AT) = 𝔏𝔉 (L(T),‘A’)=𝔏𝔉 (4, ‘A’)=1+0=1
    - U(AT) = 𝔏𝔉 (U(T)+1,‘A’)-1= 𝔏𝔉 (6,‘A’)-1=1+1-1=1
  - The matched row is: 1
    - The index in original string is 2

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

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

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) = $\mathfrak{LF}$ (L(C), ‘T’) =$\mathfrak{LF}$ (2, ‘T’)=4+0=4
    - U(TC) = $\mathfrak{LF}$ (U(C)+1,‘T’)-1 =$\mathfrak{LF}$ (4,‘T’)-1=4+2-1=5
  - The matched row is: 4 and 5.
    - The index in original string is 0 and 3.



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

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

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) = 𝔏𝔉 (L(C), ‘A’)= 𝔏𝔉 (2, ‘A’)=1+0=1
        - U(AC) = 𝔏𝔉 (U(C)+1,‘A’)-1=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) = LF (L(C), ‘T’)=LF (2,‘T’)=4+0=4
    - U(TC) = LF (U(C)+1,‘T’)-1=LF (4,‘T’)-1 = 4+2-1=5
  - 3. for “A” in “ATC”
    - L(ATC) = LF (4, ‘A’) = 1 + 0 = 1
    - U(ATC) = LF (6, ‘A’)-1 =1+1-1 = 1
  - The matched row is 1
    - The original index is 2

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

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

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

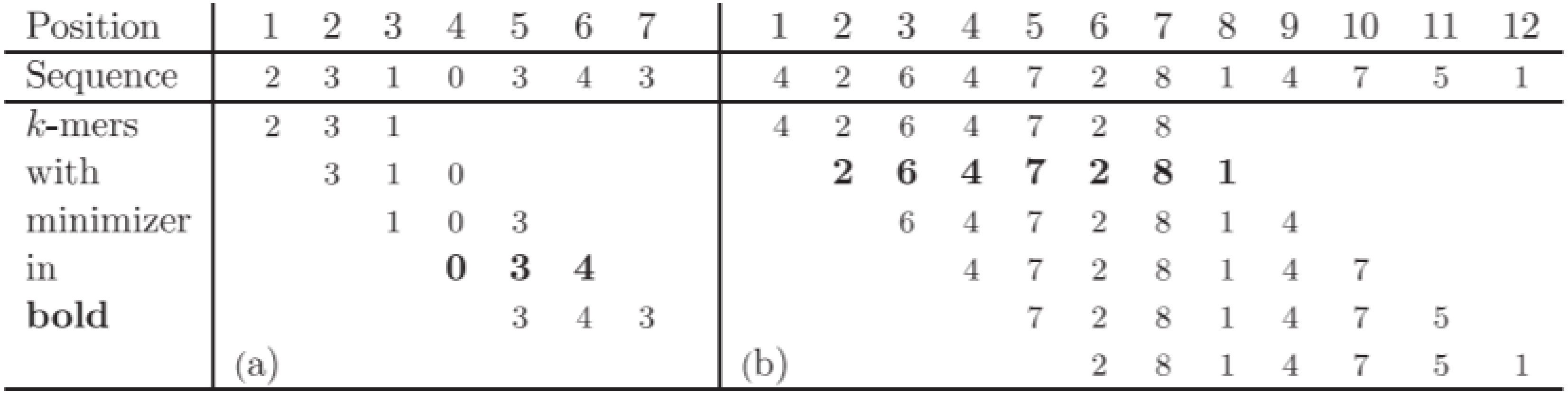
Li, Bioinformatics, 2018

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 Ti and Tj 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
k-mers	2	3	1					4	2	6	4	7	2	8					
with		3	1	0					**2**	**6**	**4**	**7**	**2**	**8**	**1**				
minimizer			1	0	3					6	4	7	2	8	1	4			
in				**0**	**3**	**4**					4	7	2	8	1	4	7		
bold					3	4	3					7	2	8	1	4	7	5	
	(a)							(b)					2	8	1	4	7	5	1

Roberts et al, Bioinformatics, 2004

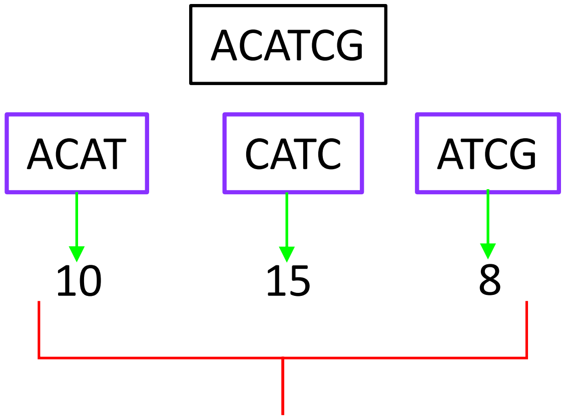
Minimap2

- 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

The sequence

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

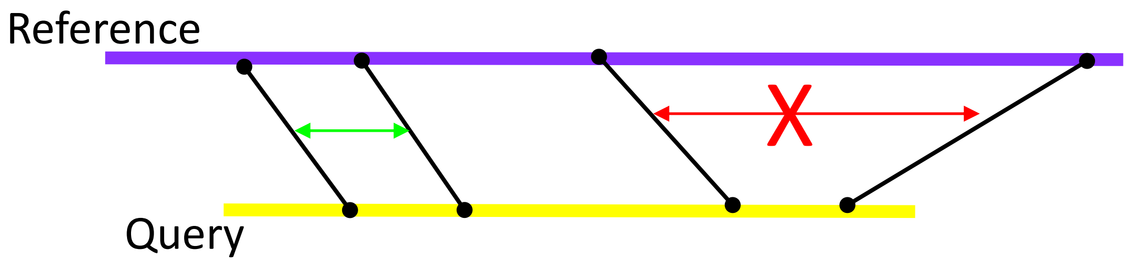
Li, Bioinformatics, 2018

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