# CS681: Advanced Topics in Computational Biology

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

**http://www.cs.bilkent.edu.tr/~calkan/teaching/cs681/**

## Read Mapping

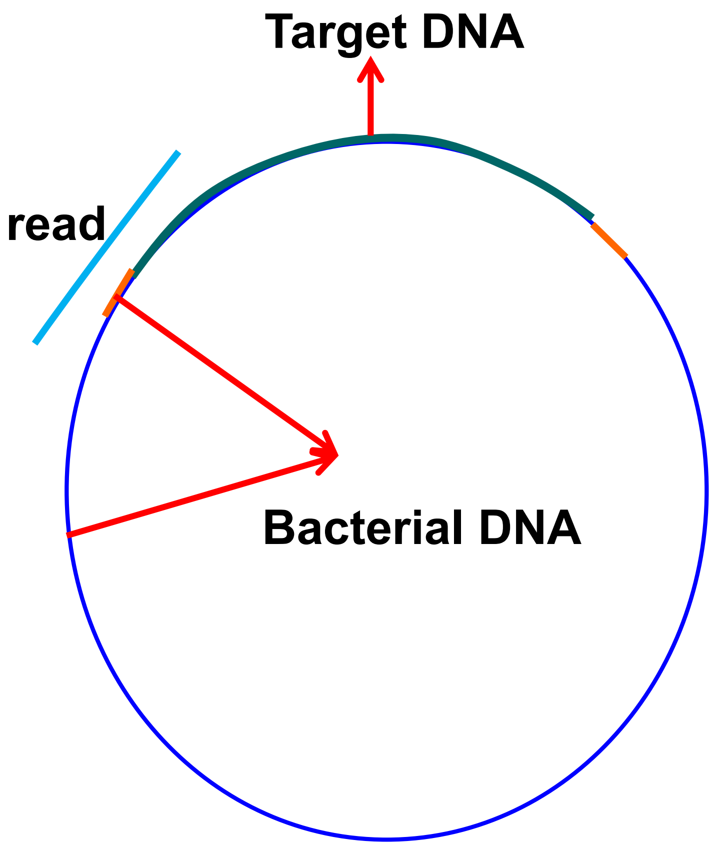
- When we have a reference genome & reads from DNA sequencing, which part of the genome does it come from?
- Challenges:
  - Sanger sequencing
    - Cloning vectors
    - Millions of long (~1000 bp reads)
  - High throughput sequencing:
    - Billions of short reads with low error
    - Hundreds of millions of long reads with high error
  - Common: contamination
    - Typically ~1-2% of reads come from different sources; e.g., human resequencing contaminated with yeast, E. coli, etc.
  - Common: Repeats & Duplications

# Read Mapping

- Accuracy
  - Due to repeats, we need a confidence score in alignment
- Sensitivity
  - Don’t lose information
- Speed!!!!!!!!
- Memory usage
- Output
  - Keep all needed information, but don’t overflow your disks -- SAM/BAM/CRAM format
- All read mapping algorithms perform alignment at some point (read vs. reference)

# Sanger vs HTS: cloning vectors

- Sanger reads may contain sequence from the cloning vector; thus mapping needs *local alignment.*
- No cloning vectors in HTS, *global alignment* is fine.



## Local vs. Global Alignment

- The Global Alignment Problem tries to find the best alignment from **start** to **end** for two sequences
- The Local Alignment Problem tries to find the subsequences of two sequences that give the best alignment
- Solutions to both are extensions of Longest Common Subsequence

# Local vs. Global Alignment (cont'd)

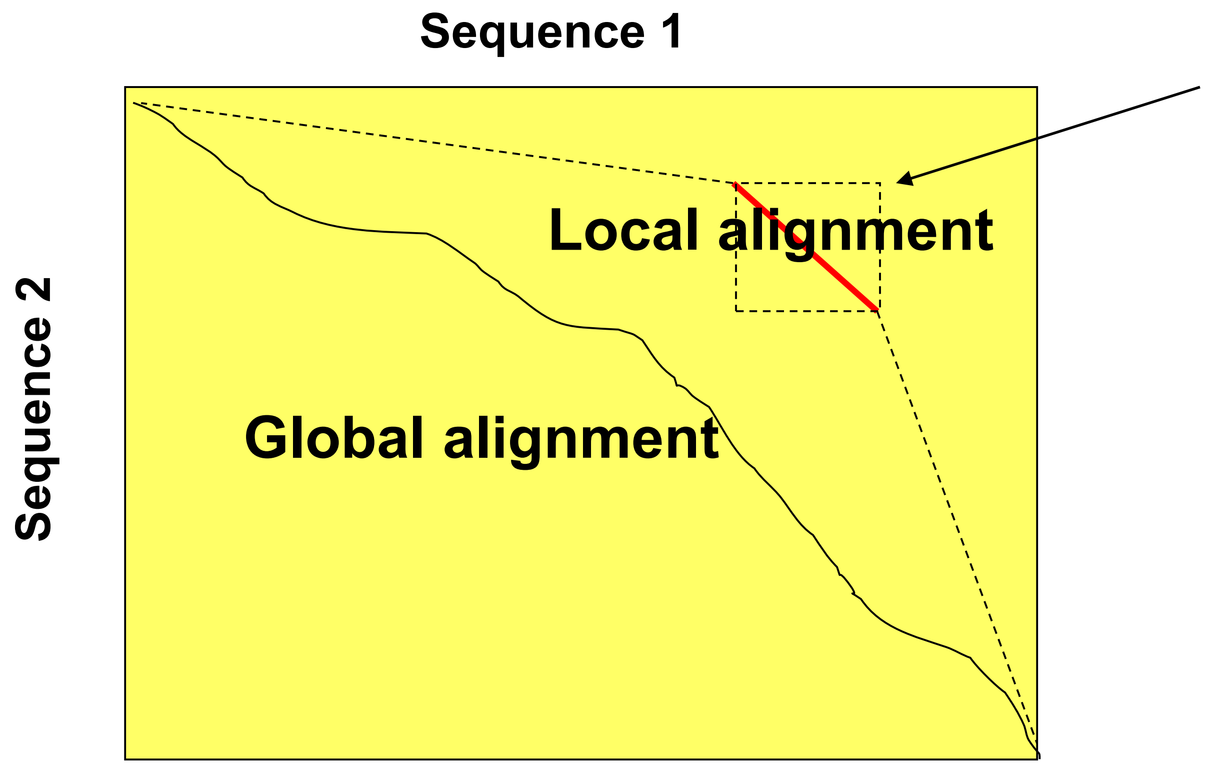
- **Global Alignment**

```
--T--CC-C-AGT--TATGT-CAGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || | | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG-T-CAGAT--C
```

- **Local Alignment—better alignment to find conserved segment**

```
                  tccCAGTTATGTCAGgggacacgagcatgcagagac
                     ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

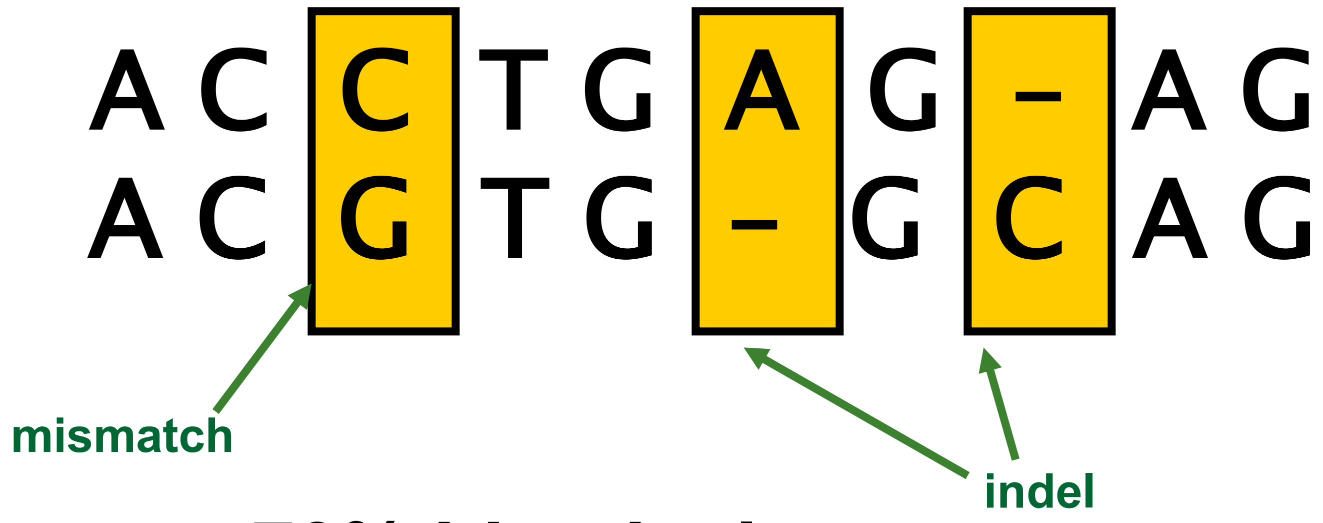
# Local Alignment: Example



**Compute a "mini" Global Alignment to get Local**

## Percent Sequence Identity

- The extent to which two nucleotide or amino acid sequences are invariant



**70% identical**

## Global Alignment

- Hamming distance:
  - Easiest; two sequences $s_1$, $s_2$, where $|s_1|=|s_2|$
  - $HD(s_1, s_2)$ = #mismatches
- Edit distance
  - Include indels in alignment
  - Levenstein's edit distance algorithm, simple recursion with match score = +1, mismatch=indel=-1; O(mn)
  - Needleman-Wunsch: extension with scoring matrices and *affine gap penalties;* O(mn)

# Edit Distance vs Hamming Distance

Hamming distance always compares i-th letter of v with i-th letter of w

```
v = ATATATAT
    ||||||||
w = TATATATA
```

**Hamming distance:**

**d(v, w)=8**

Edit distance may compare i-th letter of v with j-th letter of w

```
v = -ATATATAT
     |||||||
w = TATATATA-
```

**Edit distance:**

**d(v, w)=2**

**(one insertion and one deletion)**

# The Global Alignment Problem

Find the best alignment between two strings under a given scoring schema

Input : strings **v** and **w** and a scoring schema
Output : Alignment of maximum score

$\uparrow\rightarrow$ = -*б*
= *1* if match
$\searrow$ = -*μ* if mismatch

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\ s_{i-1,j-1} - \mu & \text{if } v_i \neq w_j \\ s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \end{cases}$$

***μ* : mismatch penalty**
***σ* : indel penalty**

## Scoring matrices

- Different scores for different character match & mismatches
- Amino acid substitution matrices
  - PAM
  - BLOSUM
- DNA substitution matrices
  - DNA is less conserved than protein sequences
  - Less effective to compare coding regions at nucleotide level

## Scoring matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ.

In the case of an amino acid sequence alignment, the scoring matrix would be a (20+1)x(20+1) size. The addition of 1 is to include the score for comparison of a gap character "-".

This will simplify the algorithm as follows:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

## Scoring Indels: Naive Approach

- A fixed penalty $\sigma$ is given to every indel:
  - $-\sigma$ for 1 indel,
  - $-2\sigma$ for 2 consecutive indels
  - $-3\sigma$ for 3 consecutive indels, etc.

Can be too severe penalty for a series of 100 consecutive indels

# Affine Gap Penalties

- In nature, a series of *k* indels often come as a single event rather than a series of *k* single nucleotide events:

**ATA__GC**
**ATATTGC**

**ATAG_GC**
**AT_GTGC**

**This is more likely.**

**Normal scoring would give the same score for both alignments**

**This is less likely.**

# Accounting for Gaps

- *Gaps*- contiguous sequence of spaces in one of the rows

- Score for a gap of length $x$ is:

  $$-(\rho + \sigma x)$$

  where $\rho > 0$ is the penalty for introducing a gap:

  gap opening penalty

  $\rho$ will be large relative to $\sigma$:

  gap extension penalty

  because you do not want to add too much of a penalty for extending the gap.

## Affine Gap Penalties

- Gap penalties:
  - $-\rho-\sigma$ when there is 1 indel
  - $-\rho-2\sigma$ when there are 2 indels
  - $-\rho-3\sigma$ when there are 3 indels, etc.
  - $-\rho- x\cdot\sigma$ (-gap opening - $x$ gap extensions)
- Somehow reduced penalties (as compared to naïve scoring) are given to runs of horizontal and vertical edges

# Affine Gap Penalty Recurrences

$$\overset{\downarrow}{s}_{i,j} = \max \begin{cases} \overset{\downarrow}{s}_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho + \sigma) \end{cases}$$

Continue Gap in *w* (deletion)

Start Gap in *w* (deletion): from middle

$$\overset{\rightarrow}{s}_{i,j} = \max \begin{cases} \overset{\rightarrow}{s}_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

Continue Gap in *v* (insertion)

Start Gap in *v* (insertion):from middle

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ \overset{\downarrow}{s}_{i,j} \\ \overset{\rightarrow}{s}_{i,j} \end{cases}$$

Match or Mismatch

End deletion: from top

End insertion: from bottom

# Ukkonen's Approximate String Matching

**Regular alignment**

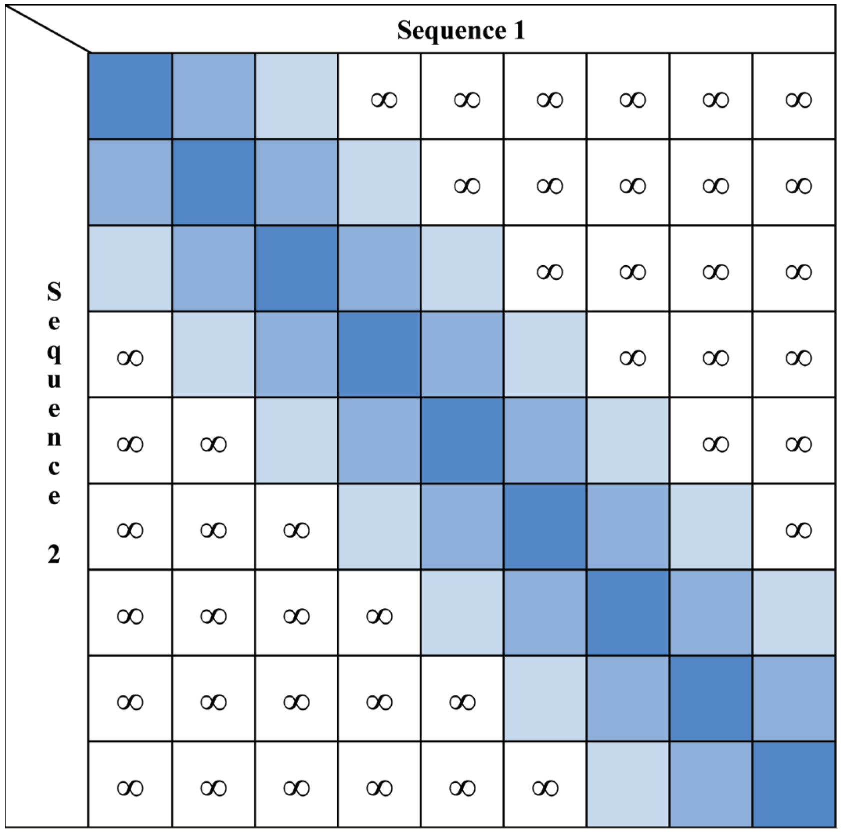
**Observation:**
**If max allowed edit distance is small, you don't go far away from the diagonal**

**(global alignment only)**

| | | A | U | U | G | A | C | A | G | G |
|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **A** | 1 | ↖**0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **U** | 2 | 1 | ↖**0** | ←**1** | ←**2** | ←**3** | 4 | 5 | 6 | 7 |
| **C** | 3 | 2 | 1 | 1 | 2 | 3 | ↖**3** | 4 | 5 | 6 |
| **A** | 4 | 3 | 2 | 2 | 2 | 2 | 3 | ↖**3** | 4 | 5 |
| **G** | 5 | 4 | 3 | 3 | 2 | 3 | 3 | 4 | ↖**3** | 4 |
| **G** | 6 | 5 | 4 | 4 | 3 | 3 | 4 | 4 | 4 | ↖**3** |
| **C** | 7 | 6 | 5 | 5 | 4 | 4 | 3 | 4 | 5 | ↑**4** |
| **C** | 8 | 7 | 6 | 6 | 5 | 5 | 4 | 4 | 5 | ↑**5** |

**AUUGACAGG - -**
**AU - - - CAGGCC**

# Ukkonen's alignment



**If maximum allowed number of indels is *t*, then you only need to calculate 2*t*-1 diagonals around the main diagonal.**

## The Local Alignment Recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.

- The recurrence:

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex

**Smith-Waterman Algorithm**

## Smith-Waterman

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

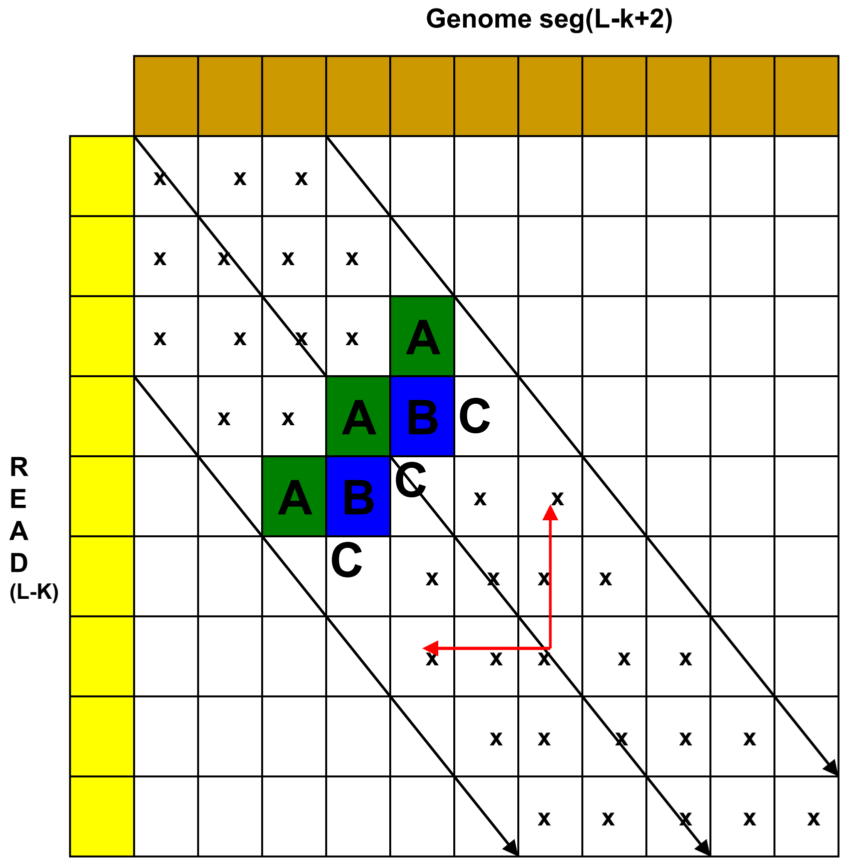
- Start from the maximum score s(i,j) on the alignment matrix
- Move to m(i-1, j), m(i, j-1) or m(i-1, j-1) until s(i,j)=0 or i=j=0
- O(mn)

## Faster Implementations

- GPGPU: general purpose graphics processing units
  - Should avoid branch statements (if-then-else)
- FPGA: field programmable gate arrays
- SIMD instructions: single-instruction multiple data
  - SSE instruction set (Intel)
    - Also available on AMD processors
    - Same instruction is executed on multiple data concurrently

# Alignment with SSE

- Applicable to both global and local alignment
- Using SSE instruction set we can compute each diagonal in parallel
- Each diagonal will be in saved in a 128 bit SSE specific register
- The diagonal C, can be computed from diagonal A and B in parallel
- Number of SSE registers is limited, we can not hold the matrix, but only the two last diagonals is needed anyway.



## READ MAPPERS

## Mapping Reads

*Problem:* We are given a read, *R,* and a reference sequence, *S*. Find the best or all occurrences of *R* in *S*.

Example:

R = AAACGAGTTA

S = TTAATGC*AAACGAGTTA*CCCAATATATATAT*AAACCAGTTA*TT

Considering no error: one occurrence.

Considering up to 1 substitution error: two occurrences.

Considering up to 10 substitution errors: many meaningless occurrences!

***Don't forget to search in both forward and reverse strands!!!***

## Mapping Reads (continued)

*Variations:*

- Sequencing error
  - No error: $R$ is a perfect subsequence of $S$.
  - Only substitution error: $R$ is a subsequence of $S$ up to a few substitutions.
  - Indel and substitution error: $R$ is a subsequence of $S$ up to a few short indels and substitutions.
- Junctions (for instance in alternative splicing)
  - Fixed order/orientation

    $R = R_1R_2\ldots R_n$ and $R_i$ map to different non-overlapping loci in $S$, but to the same strand and preserving the order.
  - Arbitrary order/orientation

    $R = R_1R_2\ldots R_n$ and $R_i$ map to different non-overlapping loci in $S$.

# Hash based seed-and-extend aligners

- Hash based seed-and-extend (hash table, suffix array, suffix tree)
  - Index the k-mers in the genome
    - Continuous seeds and gapped seeds
  - When searching a read, find the location of a k-mer in the read; then extend through alignment
    - Apply pre-alignment filters
      - GateKeeper, adjacency filter, q-gram filters
  - Requires large memory; this can be reduced with cost to run time
  - mr(s)FAST, RazerS3, MAQ, MOSAIK
  - GPGPU and heterogeneous computing implementations: Saruman, Mummer-GPU, CORAL

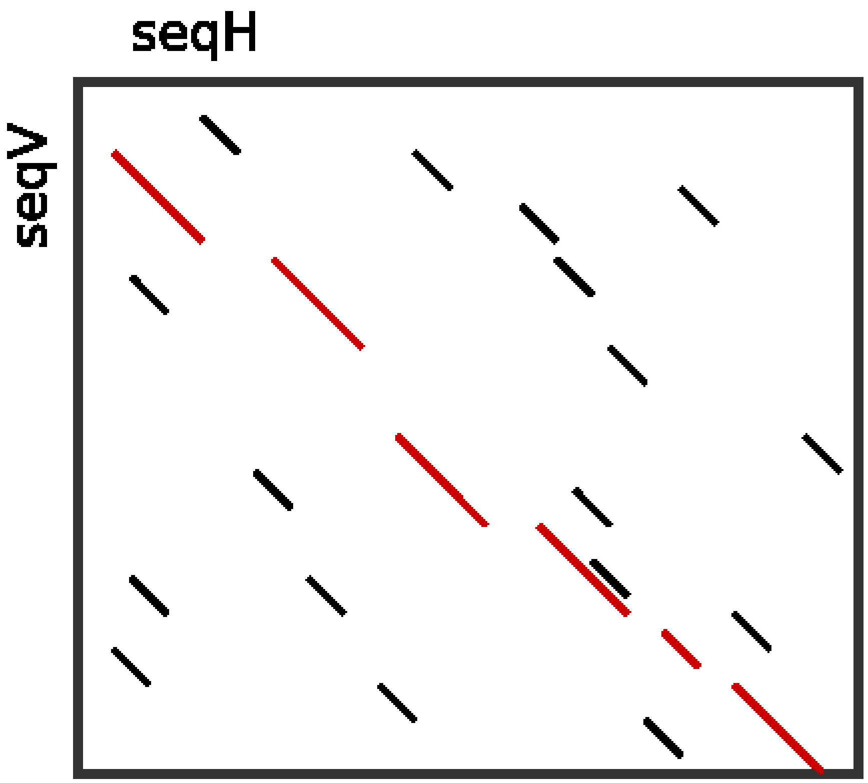
# (pure) BWT-FM aligners

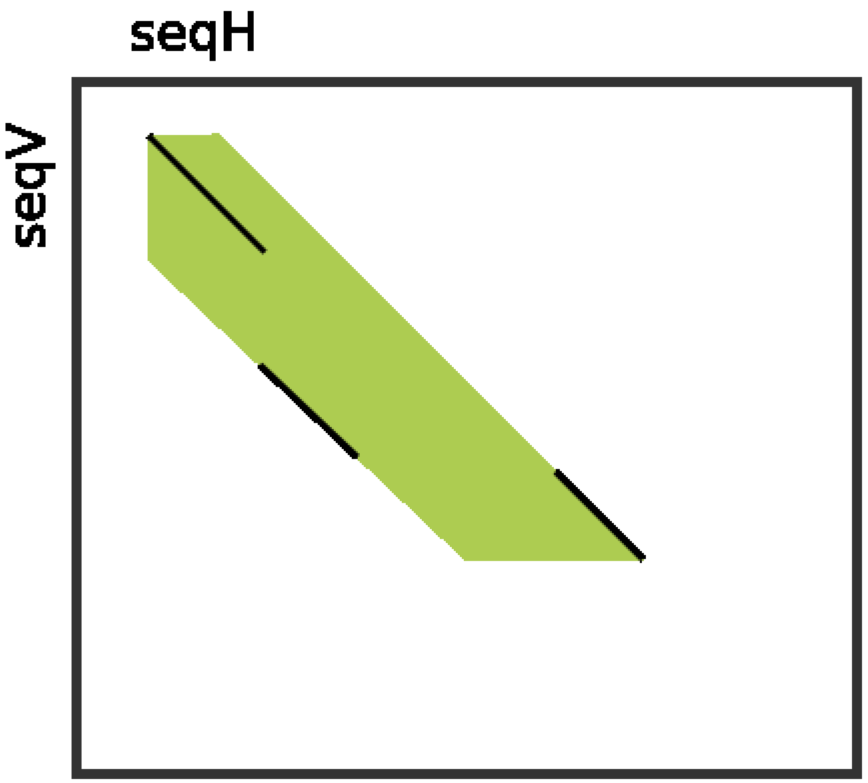
- Burrows-Wheeler Transform & Ferragina-Manzini Index based aligners
  - BWT is a data compression method used to compress the genome index
  - Perfect hits can be found very quickly, memory lookup costs increase for imperfect hits
  - Less memory
  - Reduced sensitivity for high error rate (impractical for long reads)
  - BWA-aln, Bowtie, SOAP2

# Hybrid aligners

- Seed with BWT-FM, then align
  - Apply "chaining" to reduce need-to-align regions (acts as a pre-alignment filter)
  - Usable for both short and long reads
  - BWA-MEM, Bowtie2 (short reads)
  - MashMap and minimap2 (long reads)

# Seeding & chaining

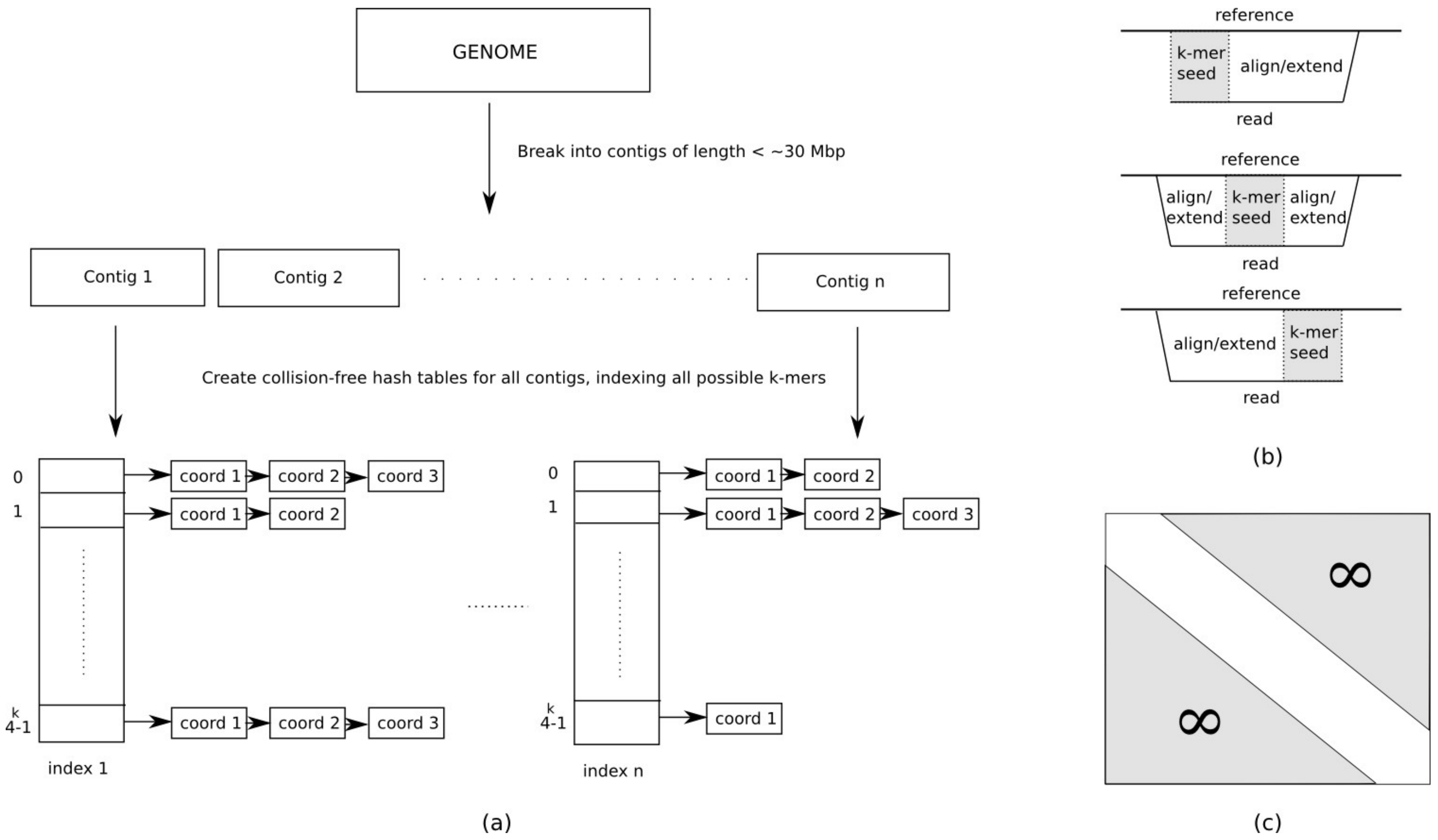




## Long read mappers

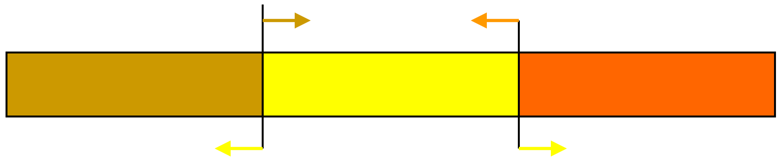
- PacBio and ONT:
  - BLASR (suffix-tree based indexing)
  - MashMap and Minimap2 (minimizers + chaining + Smith-Waterman)
    - Paper presentation candidate
  - NGM-LR (hash table + chaining + alignment w/ convex gap penalty model
    - Paper presentation candidate

# Hash Based Aligners



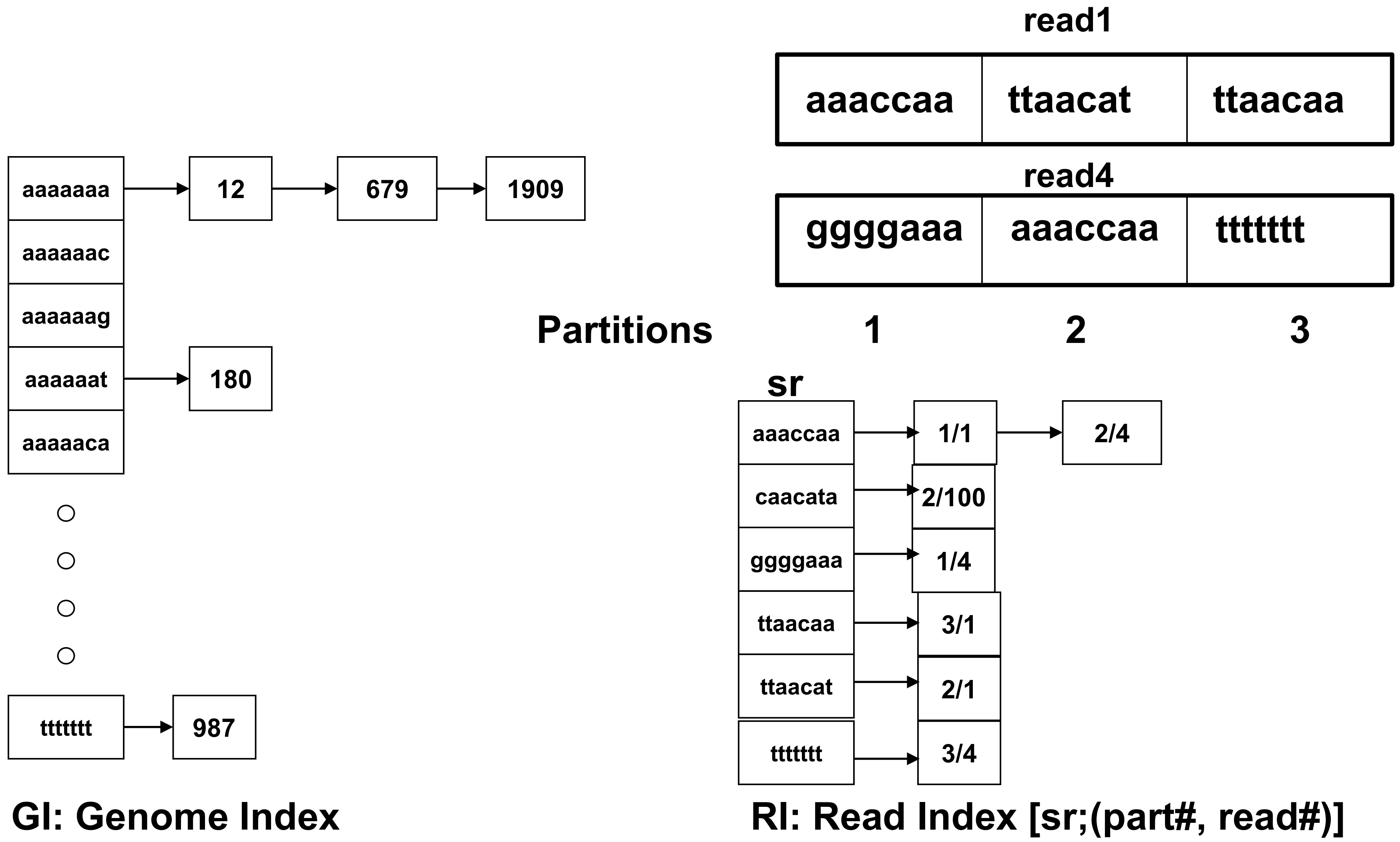
# Seed and extend

- Break the read into *n* segments of k-mers.
  - For perfect sensitivity under edit distance *e*
    - There is at least one *l*-mer where l = floor(*L*/(e+1)); *L*=read length
    - For fixed *l*=*k*; *n* = *e*+1 and k ≤ *L* / *n*
  - Large k -> large memory
  - Small k -> more hash hits
- Lets consider the read length is 36 bp, and k=12.



- if we are looking for 2 edit distance (mismatch, indel) this would guaranty to find all of the hits

# Cache oblivious search



*Hach et al, Nat Methods 2010*

## Cache oblivious search

- GI and RI are both sorted
- Scan GI; for all GI[i] = RI[j].*sr*
    - Map all partition/read_number combinations in RI[j]
    - All of the above have the same *sr* and its corresponding GI[i] list*;* therefore:
        - They have the same *seed* locations: same sequence content in the reference genome to *extend*
        - Once GI[i] and corresponding ref(GI[i].1, GI[i].2, …) are loaded from *main memory* to *cache memory;* then you re-use the **faster** cache memory contents; minimizing cache hits and main-to-cache transfers

*Hach et al, Nat Methods 2010*

# Cache oblivious search

| Mapper | Level 2 Cache Misses per Instruction | Instruction per cycle |
|---|---|---|
| Bowtie | 0.0016 | 0.94 |
| BWA | 0.0016 | 0.93 |
| MAQ | 0.0060 | 0.56 |
| mrsFAST | 0.0008 | 1.24 |

*Hach et al, Nat Methods 2010*

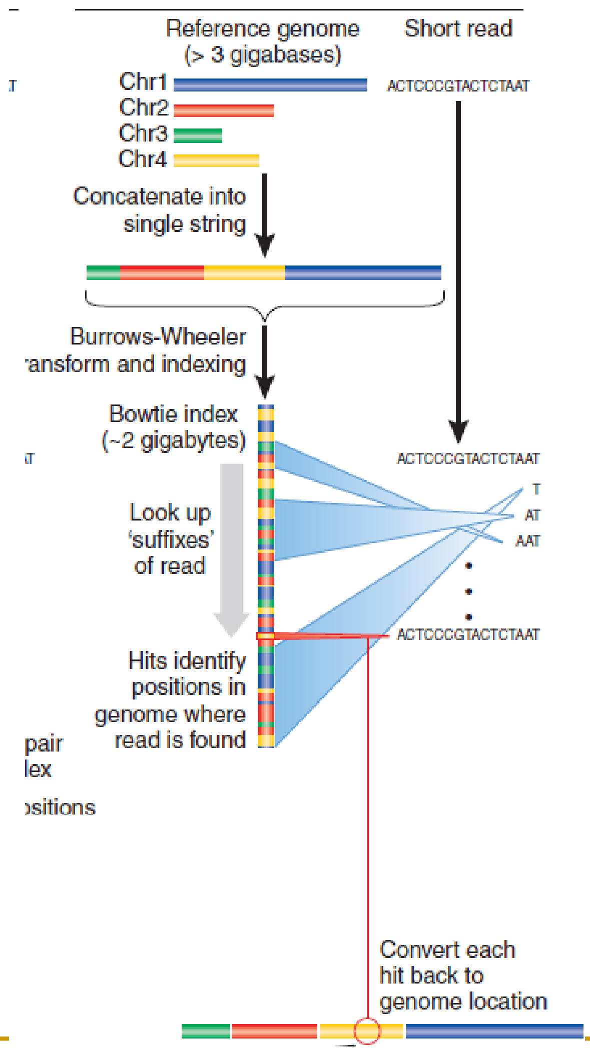
## Spaced seeds

- Instead of a k-mer with contiguous hit (1111..1); use space seeds
  - Seed S is defined by Length and Weight
- 0's are "don't care" characters
  - 111111001111111100 requires
    - 6 matches + 2 "don't care"s + 8 matches + 2 "don't care"s; a valid hit:

```
CGACTAGCTAGCTAGCTA
CGACTAAGTAGCTAGCGC
```

    - Length = 18;  weight = 14

# Burrows-Wheeler Transform



Reference genome (> 3 gigabases)
Short read
Chr1
Chr2
Chr3
Chr4
ACTCCCGTACTCTAAT
Concatenate into single string
Burrows-Wheeler ransform and indexing
Bowtie index (~2 gigabytes)
Look up 'suffixes' of read
ACTCCCGTACTCTAAT
T
AT
AAT
ACTCCCGTACTCTAAT
Hits identify positions in genome where read is found
pair
lex
sitions
Convert each hit back to genome location

- Store entire reference genome.
- Align tag base by base from the end.
- When tag is traversed, all active locations are reported.
- If no match is found, then back up and try a substitution.

## Burrows-Wheeler Transformation

1. Append to the input string a special char, $, smaller than all alphabet.

**mississippi$**

# Burrows-Wheeler Transformation (cnt'd)

2. Generate all rotations.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| m | i | s | s | i | s | s | i | p | p | i | $ |
| i | s | s | i | s | s | i | p | p | i | $ | m |
| s | s | i | s | s | i | p | p | i | $ | m | i |
| s | i | s | s | i | p | p | i | $ | m | i | s |
| i | s | s | i | p | p | i | $ | m | i | s | s |
| s | s | i | p | p | i | $ | m | i | s | s | i |
| s | i | p | p | i | $ | m | i | s | s | i | s |
| i | p | p | i | $ | m | i | s | s | i | s | s |
| p | p | i | $ | m | i | s | s | i | s | s | i |
| p | i | $ | m | i | s | s | i | s | s | i | p |
| i | $ | m | i | s | s | i | s | s | i | p | p |
| $ | m | i | s | s | i | s | s | i | p | p | i |

# Burrows-Wheeler Transformation (cnt'd)

3. Sort rotations according to the alphabetical order.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $ | m | i | s | s | i | s | s | i | p | p | i |
| i | $ | m | i | s | s | i | s | s | i | p | p |
| i | p | p | i | $ | m | i | s | s | i | s | s |
| i | s | s | i | p | p | i | $ | m | i | s | s |
| i | s | s | i | s | s | i | p | p | i | $ | m |
| m | i | s | s | i | s | s | i | p | p | i | $ |
| p | i | $ | m | i | s | s | i | s | s | i | p |
| p | p | i | $ | m | i | s | s | i | s | s | i |
| s | i | p | p | i | $ | m | i | s | s | i | s |
| s | i | s | s | i | p | p | i | $ | m | i | s |
| s | s | i | p | p | i | $ | m | i | s | s | i |
| s | s | i | s | s | i | p | p | i | $ | m | i |

## Burrows-Wheeler Transformation (cnt'd)

4. Output the last column.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $ | m | i | s | s | i | s | s | i | p | p | i |
| i | $ | m | i | s | s | i | s | s | i | p | p |
| i | p | p | i | $ | m | i | s | s | i | s | s |
| i | s | s | i | p | p | i | $ | m | i | s | s |
| i | s | s | i | s | s | i | p | p | i | $ | m |
| m | i | s | s | i | s | s | i | p | p | i | $ |
| p | i | $ | m | i | s | s | i | s | s | i | p |
| p | p | i | $ | m | i | s | s | i | s | s | i |
| s | i | p | p | i | $ | m | i | s | s | i | s |
| s | i | s | s | i | p | p | i | $ | m | i | s |
| s | s | i | p | p | i | $ | m | i | s | s | i |
| s | s | i | s | s | i | p | p | i | $ | m | i |

## Burrows-Wheeler Transformation (cnt'd)

**mississippi$**

$\downarrow$

ipssm$pissii

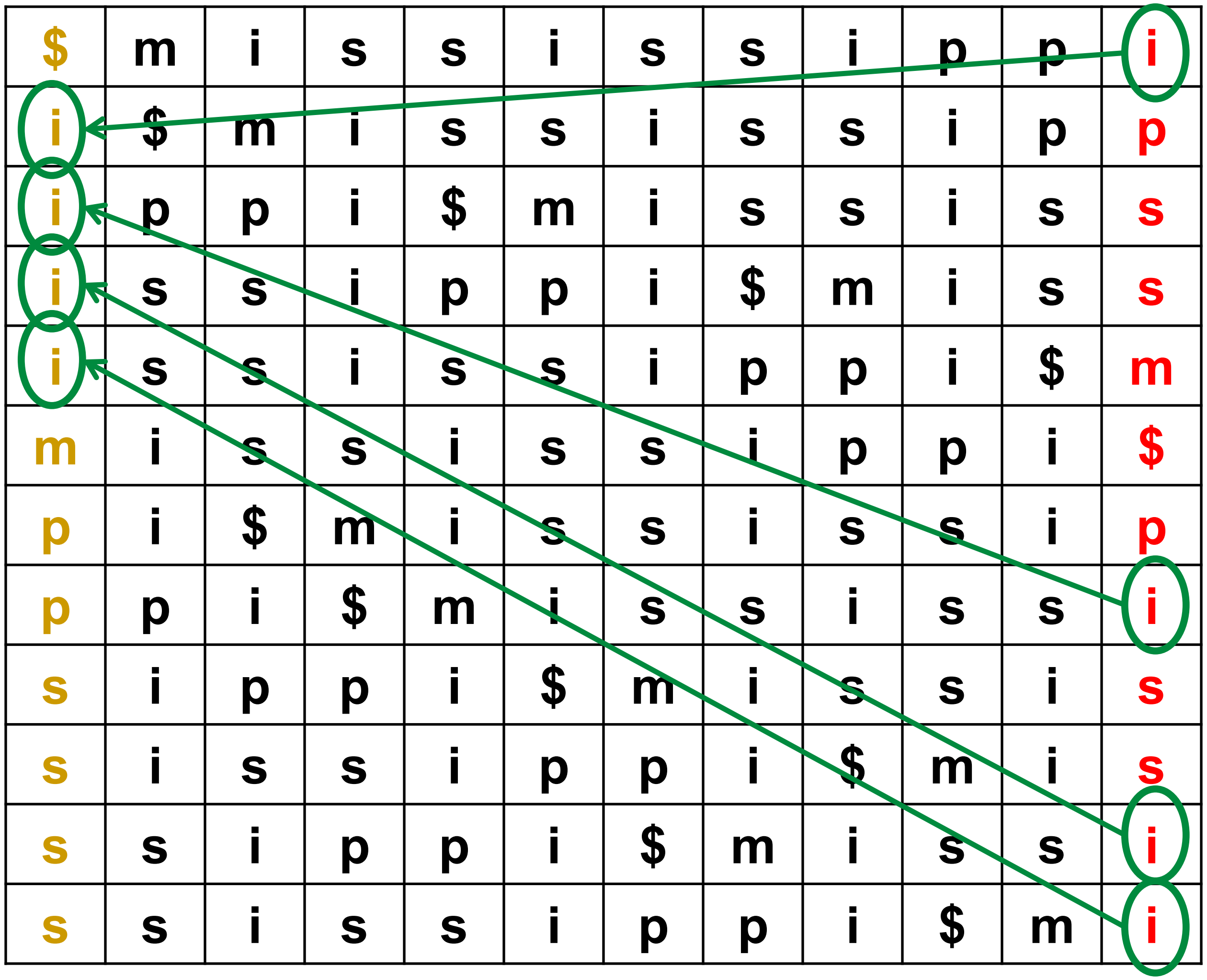
# Ferragina-Manzini Index

First column: F

Last column: L

Let’s make an L to F map.

Observation: The nth i in L is the nth i in F.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $ | m | i | s | s | i | s | s | i | p | p | i |
| i | $ | m | i | s | s | i | s | s | i | p | p |
| i | p | p | i | $ | m | i | s | s | i | s | s |
| i | s | s | i | p | p | i | $ | m | i | s | s |
| i | s | s | i | s | s | i | p | p | i | $ | m |
| m | i | s | s | i | s | s | i | p | p | i | $ |
| p | i | $ | m | i | s | s | i | s | s | i | p |
| p | p | i | $ | m | i | s | s | i | s | s | i |
| s | i | p | p | i | $ | m | i | s | s | i | s |
| s | i | s | s | i | p | p | i | $ | m | i | s |
| s | s | i | p | p | i | $ | m | i | s | s | i |
| s | s | i | s | s | i | p | p | i | $ | m | i |

# Ferragina-Manzini Index: L to F map

Store/compute a two dimensional Occ(*j*,‘c’) table of the number of occurrences of char ‘c’ up to position *j* (inclusive).

and one dimensional Cnt(‘c’) and Rank(‘c’) tables

| | $ | i | m | p | s |
|---|---|---|---|---|---|
| i | 0 | 1 | 0 | 0 | 0 |
| p | 0 | 1 | 0 | 1 | 0 |
| s | 0 | 1 | 0 | 1 | 1 |
| s | 0 | 1 | 0 | 1 | 2 |
| m | 0 | 1 | 1 | 1 | 2 |
| $ | 1 | 1 | 1 | 1 | 2 |
| p | 1 | 1 | 1 | 2 | 2 |
| i | 1 | 2 | 1 | 2 | 2 |
| s | 1 | 2 | 1 | 2 | 3 |
| s | 1 | 2 | 1 | 2 | 4 |
| i | 1 | 3 | 1 | 2 | 4 |
| i | 1 | 4 | 1 | 2 | 4 |

**Occ(*j*,‘c’)**

**Cnt(‘c’)**

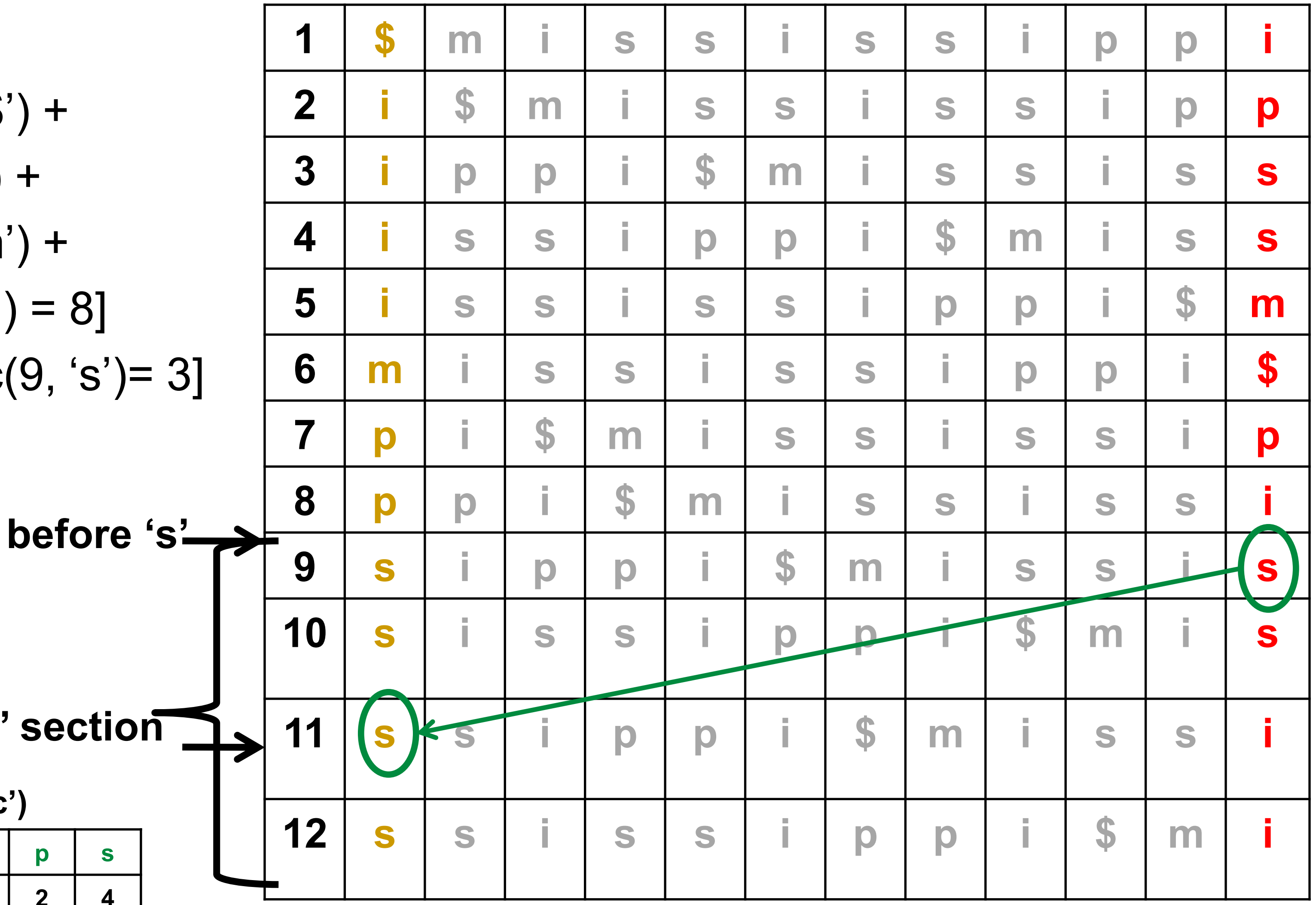
| $ | i | m | p | s |
|---|---|---|---|---|
| 1 | 4 | 1 | 2 | 4 |

**Rank(‘c’)**

| $ | i | m | p | s |
|---|---|---|---|---|
| 12 | 2 | 1 | 9 | 3 |

# Ferragina-Manzini Index: L to F map

[Cnt('$') +
Cnt('i') +
Cnt('m') +
Cnt('p') = 8]
+ [Occ(9, 's')= 3]
= 11



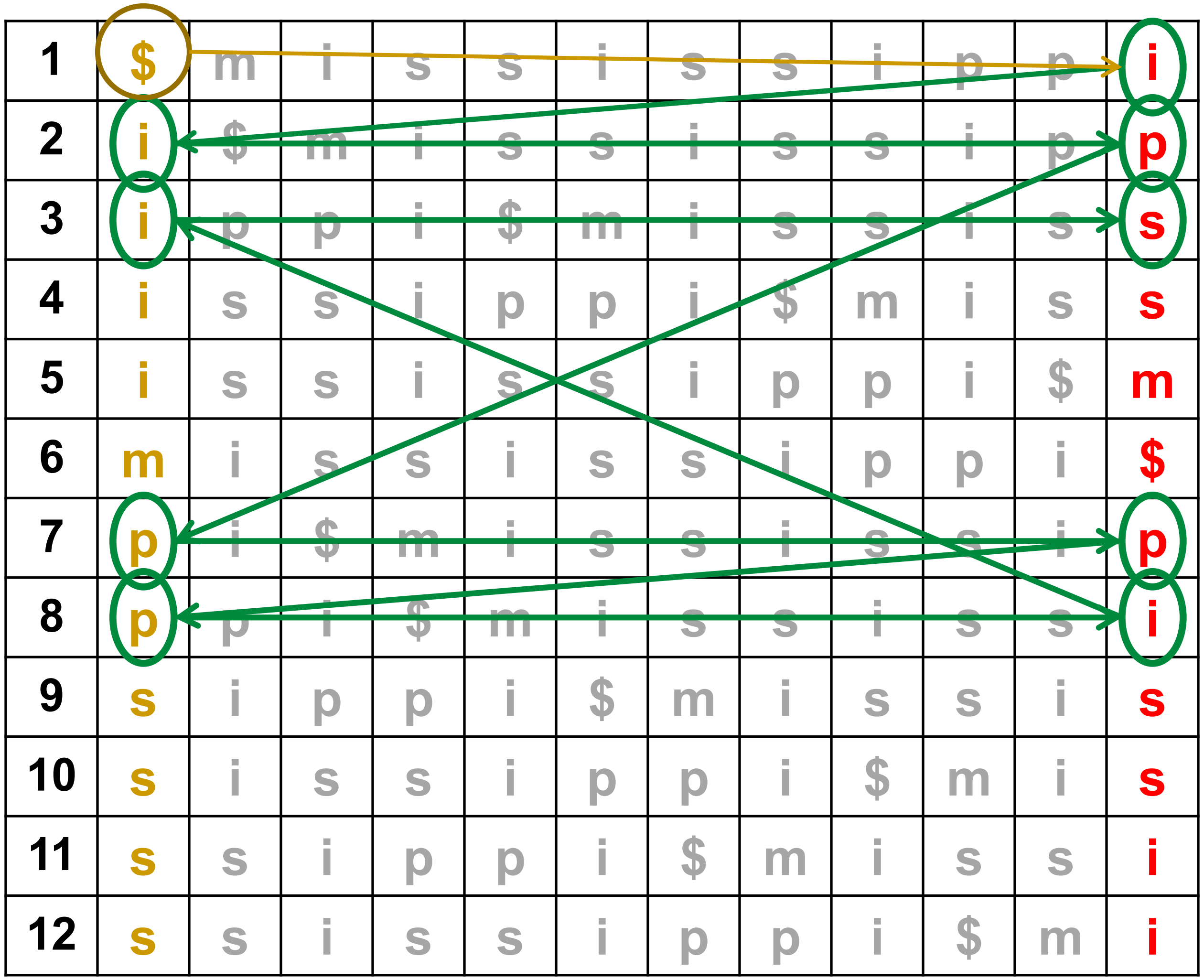
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $ | m | i | s | s | i | s | s | i | p | p | i |
| 2 | i | $ | m | i | s | s | i | s | s | i | p | p |
| 3 | i | p | p | i | $ | m | i | s | s | i | s | s |
| 4 | i | s | s | i | p | p | i | $ | m | i | s | s |
| 5 | i | s | s | i | s | s | i | p | p | i | $ | m |
| 6 | m | i | s | s | i | s | s | i | p | p | i | $ |
| 7 | p | i | $ | m | i | s | s | i | s | s | i | p |
| 8 | p | p | i | $ | m | i | s | s | i | s | s | i |
| 9 | s | i | p | p | i | $ | m | i | s | s | i | s |
| 10 | s | i | s | s | i | p | p | i | $ | m | i | s |
| 11 | s | s | i | p | p | i | $ | m | i | s | s | i |
| 12 | s | s | i | s | s | i | p | p | i | $ | m | i |

Cnt('c')

| $ | i | m | p | s |
|---|---|---|---|---|
| 1 | 4 | 1 | 2 | 4 |

# Ferragina-Manzini Index: Reverse traversal

(1) i
(2) p
(7) p
(8) i
(3) s
(9) s
(11) i
(4) s
(10) s
(12) i
(5) m
(6) $



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $ | m | i | s | s | i | s | s | i | p | p | i |
| 2 | i | $ | m | i | s | s | i | s | s | i | p | p |
| 3 | i | p | p | i | $ | m | i | s | s | i | s | s |
| 4 | i | s | s | i | p | p | i | $ | m | i | s | s |
| 5 | i | s | s | i | s | s | i | p | p | i | $ | m |
| 6 | m | i | s | s | i | s | s | i | p | p | i | $ |
| 7 | p | i | $ | m | i | s | s | i | s | s | i | p |
| 8 | p | p | i | $ | m | i | s | s | i | s | s | i |
| 9 | s | i | p | p | i | $ | m | i | s | s | i | s |
| 10 | s | i | s | s | i | p | p | i | $ | m | i | s |
| 11 | s | s | i | p | p | i | $ | m | i | s | s | i |
| 12 | s | s | i | s | s | i | p | p | i | $ | m | i |

# Mapping with BWT-FM

**Auxillary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.**

| | a | c | g | t |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |

*Original sequence*

**BWT**

**FM indices**

| | **SA** | | | | | | |
|---|---|---|---|---|---|---|---|
| | | **$**agcagcagac**t** | t | a | c | g | t |
| 1 | 9 | **a**ct$agcagca**g** | g | 0 | 0 | 1 | 1 |
| 2 | 7 | **a**gact$agcag**c** | c | 0 | 1 | 1 | 1 |
| 3 | 4 | **a**gcagact$ag**c** | c | 0 | 2 | 1 | 1 |
| 4 | 1 | **a**gcagcagac**t$** | $ | 0 | 2 | 1 | 1 |
| 5 | 6 | **c**agact$agca**g** | g | 0 | 2 | 2 | 1 |
| 6 | 3 | **c**agcagact$a**g** | g | 0 | 2 | 3 | 1 |
| 7 | 10 | **c**t$agcagcag**a** | a | 1 | 2 | 3 | 1 |
| 8 | 8 | **g**act$agcagc**a** | a | 2 | 2 | 3 | 1 |
| 9 | 5 | **g**cagact$agc**a** | a | 3 | 2 | 3 | 1 |
| 10 | 2 | **g**cagcagact$**a** | a | 4 | 2 | 3 | 1 |
| 11 | 11 | **t**$agcagcaga**c** | c | 4 | 3 | 3 | 1 |

# Mapping with BWT-FM

**Auxillary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.**

*Original sequence*

**gca**

| | SA | | BWT |
|---|---|---|---|
| | | **$**agcagcagac**t** | t |
| 1 | 9 | **a**ct$agcagca**g** | g |
| 2 | 7 | **a**gact$agcag**c** | c |
| 3 | 4 | **a**gcagact$ag**c** | c |
| 4 | 1 | **a**gcagcagact**$** | $ |
| 5 | 6 | **c**agact$agca**g** | g |
| 6 | 3 | **c**agcagact$a**g** | g |
| 7 | 10 | **c**t$agcagcag**a** | a |
| 8 | 8 | **g**act$agcagc**a** | a |
| 9 | 5 | **g**cagact$agc**a** | a |
| 10 | 2 | **g**cagcagact$**a** | a |
| 11 | 11 | **t**$agcagcaga**c** | c |

| | a | c | g | t |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |

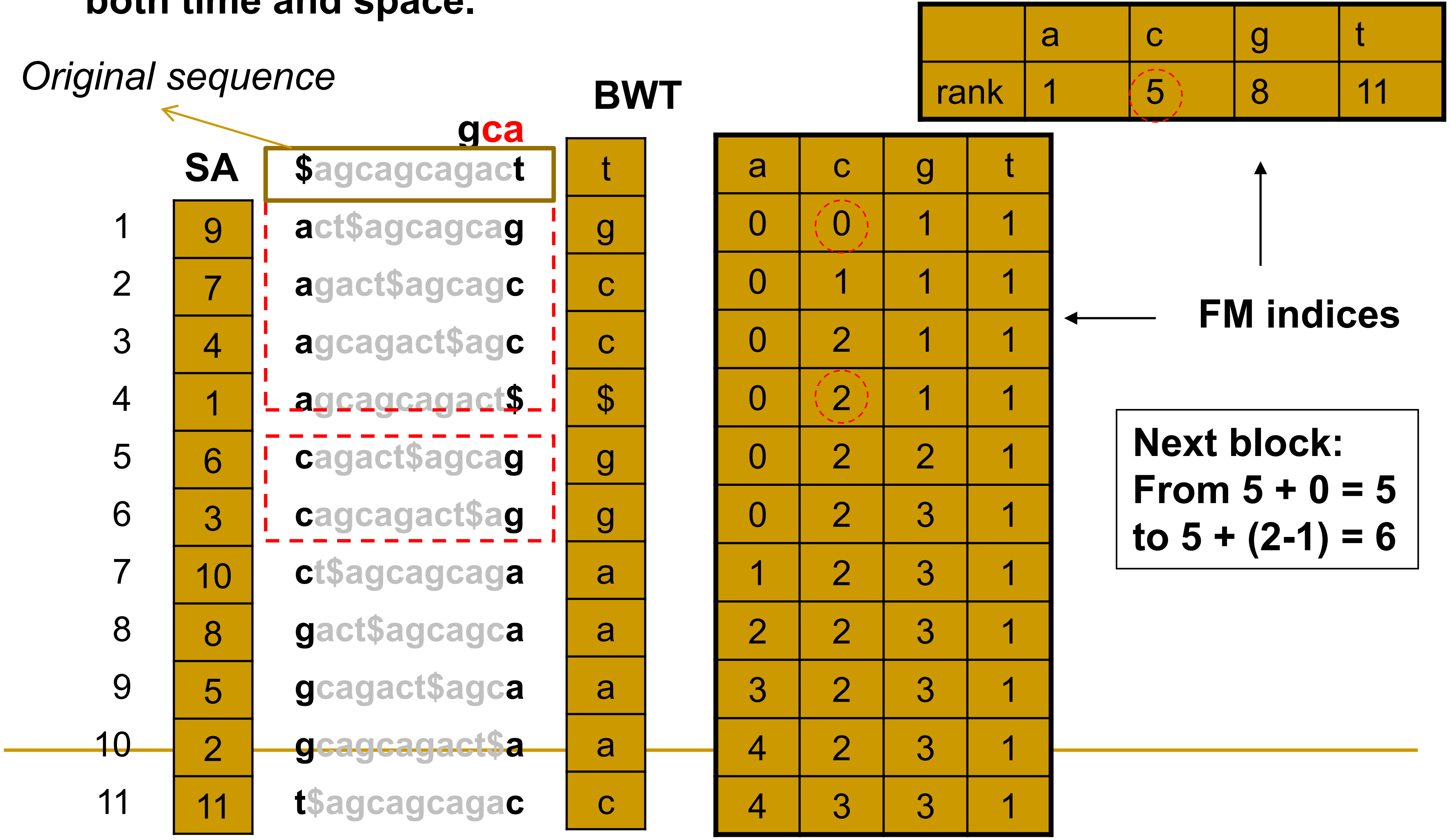
| a | c | g | t |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 2 | 1 |
| 0 | 2 | 3 | 1 |
| 1 | 2 | 3 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 2 | 3 | 1 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 3 | 1 |

**FM indices**

**Next block:**
**From 1 + 0 = 1**
**to 1 + (4-1) = 4**

# Mapping with BWT-FM

**Auxillary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.**



| | a | c | g | t |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |

| | SA | | BWT | a | c | g | t |
|---|---|---|---|---|---|---|---|
| | | $agcagcagact | t | | | | |
| 1 | 9 | act$agcagcag | g | 0 | 0 | 1 | 1 |
| 2 | 7 | agact$agcagc | c | 0 | 1 | 1 | 1 |
| 3 | 4 | agcagact$agc | c | 0 | 2 | 1 | 1 |
| 4 | 1 | agcagcagact$ | $ | 0 | 2 | 1 | 1 |
| 5 | 6 | cagact$agcag | g | 0 | 2 | 2 | 1 |
| 6 | 3 | cagcagact$ag | g | 0 | 2 | 3 | 1 |
| 7 | 10 | ct$agcagcaga | a | 1 | 2 | 3 | 1 |
| 8 | 8 | gact$agcagca | a | 2 | 2 | 3 | 1 |
| 9 | 5 | gcagact$agca | a | 3 | 2 | 3 | 1 |
| 10 | 2 | gcagcagact$a | a | 4 | 2 | 3 | 1 |
| 11 | 11 | t$agcagcagac | c | 4 | 3 | 3 | 1 |

**Next block:**
**From 5 + 0 = 5**
**to 5 + (2-1) = 6**

# Mapping with BWT-FM

**Auxillary data structures for efficient pattern matching: how to find the corresponding chars in the first column efficiently, in terms of both time and space.**

|  | a | c | g | t |
|---|---|---|---|---|
| rank | 1 | 5 | 8 | 11 |

*Original sequence*

**gca**

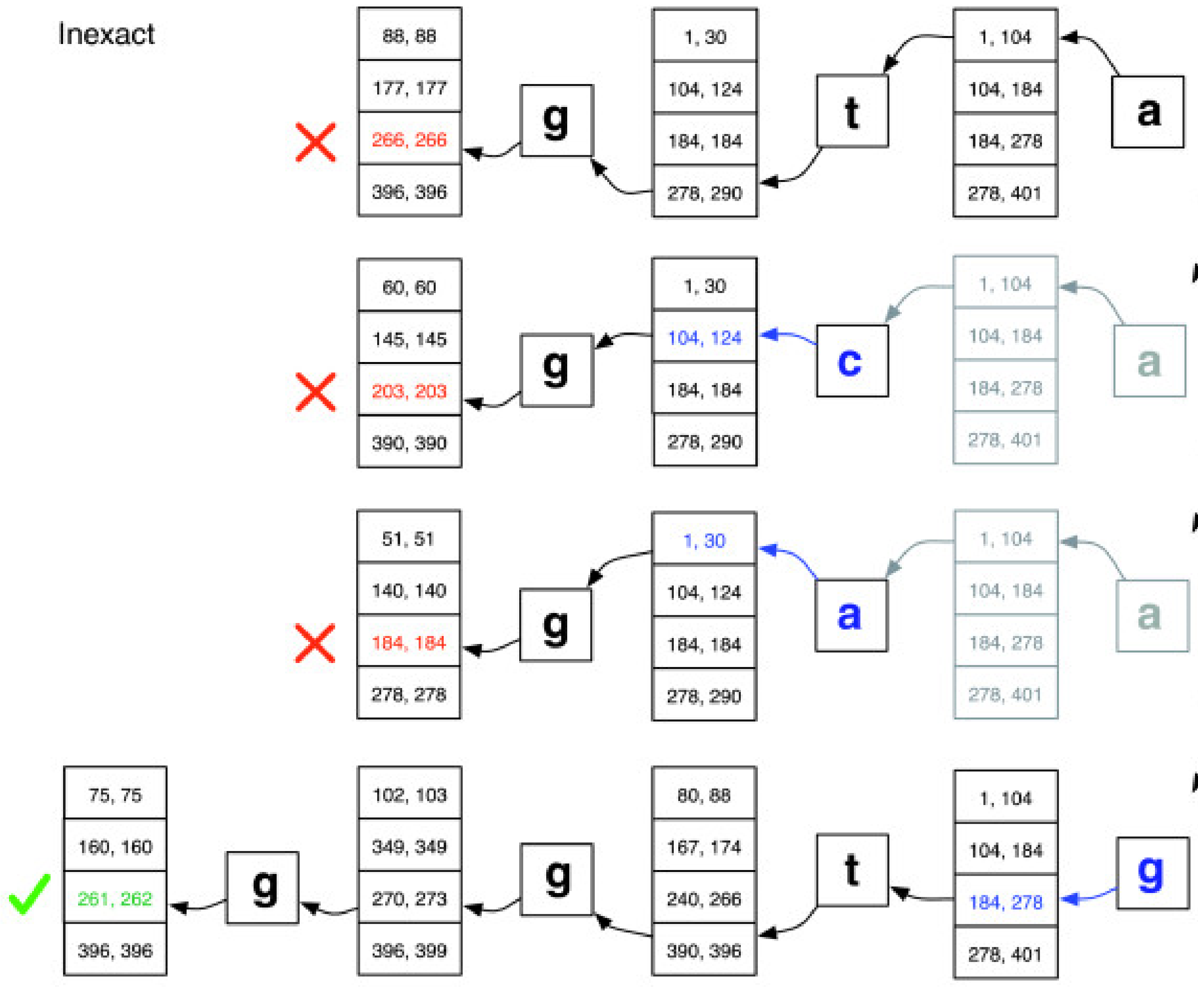
|  | SA |  | BWT |
|---|---|---|---|
|  |  | **$agcagcagact** | t |
| 1 | 9 | act$agcagcag | g |
| 2 | 7 | agact$agcagc | c |
| 3 | 4 | agcagact$agc | c |
| 4 | 1 | agcagcagact$ | $ |
| 5 | 6 | cagact$agcag | g |
| 6 | 3 | cagcagact$ag | g |
| 7 | 10 | ct$agcagcaga | a |
| 8 | 8 | gact$agcagca | a |
| 9 | 5 | gcagact$agca | a |
| 10 | 2 | gcagcagact$a | a |
| 11 | 11 | t$agcagcagac | c |

| a | c | g | t |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 1 | 1 |
| 0 | 2 | 2 | 1 |
| 0 | 2 | 3 | 1 |
| 1 | 2 | 3 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 2 | 3 | 1 |
| 4 | 2 | 3 | 1 |
| 4 | 3 | 3 | 1 |

**FM indices**

**Next block:**
**From 8 + 1 = 9**
**to 8 + (3-1) = 10**

# Inexact match



# Mapping Quality

- MAPQ = -10 * $\log_{10}$(Prob(mapping is wrong))

For reference sequence *x;* read sequence *z:*
***p(z | x,u)* =** probability that *z* comes from position *u*
= multiplication of $p_e$ of mismatched bases of *z*

For posterior probability **p(u | x,z)** assume uniform prior distribution **p(u|x)**
*L*=|x| and *l*=|z|. Apply Bayesian formula:

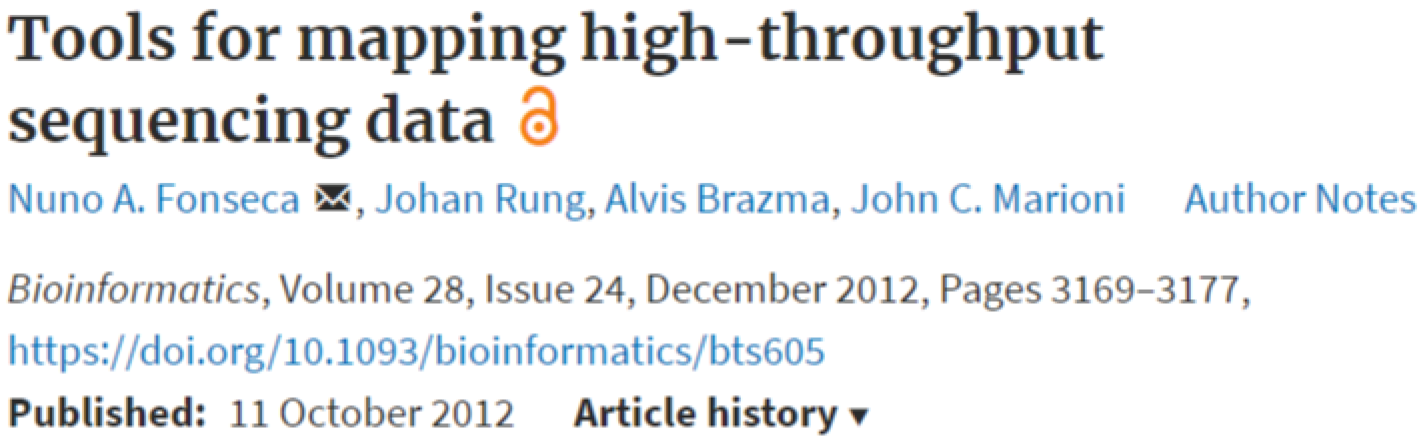
$$p_s(u|x,z) = \frac{p(z|x,u)}{\sum_{v=1}^{L-l+1} p(z|x,v)}$$

$$Q_s(u|x,z) = -10 \log_{10}[1 - p_s(u|x,z)].$$

**Calculated for one "best" hit**

Li et al., Genome Research, 2008

# Further reading

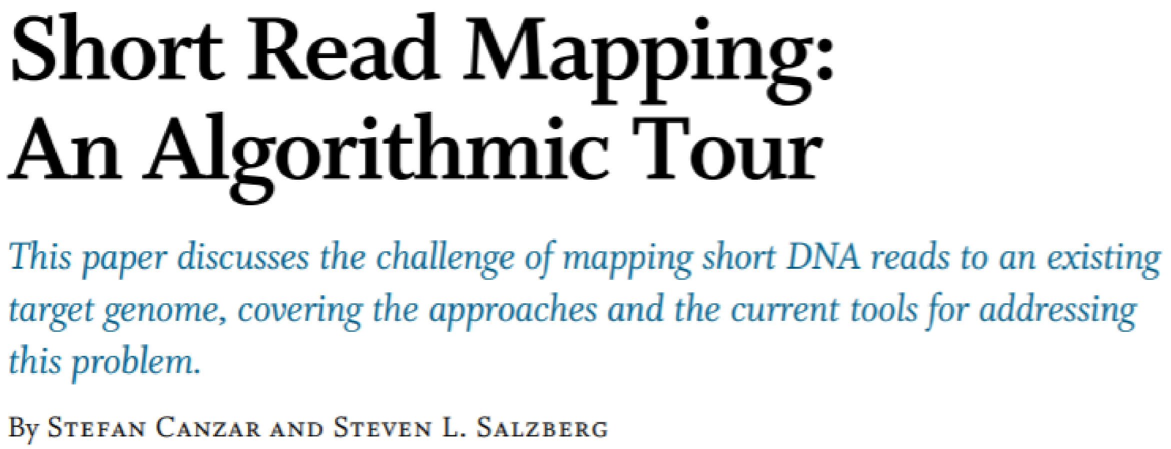


**Tools for mapping high-throughput sequencing data**

Nuno A. Fonseca ✉, Johan Rung, Alvis Brazma, John C. Marioni    Author Notes

*Bioinformatics*, Volume 28, Issue 24, December 2012, Pages 3169–3177,
https://doi.org/10.1093/bioinformatics/bts605

**Published:** 11 October 2012    **Article history** ▾



**Short Read Mapping: An Algorithmic Tour**

*This paper discusses the challenge of mapping short DNA reads to an existing target genome, covering the approaches and the current tools for addressing this problem.*

By Stefan Canzar and Steven L. Salzberg