Doubly-periodic string comparison

Nikita Gaevoy and Alexander Tiskin

Department of Mathematics and Computer Science, St Petersburg University

Longest common subsequence under string concatenation

$lcs(\text{"RUMPLESTILTSKIN"}, \text{"STEAK"}) = 3$
$lcs(\text{"RUMPLESTILTSKIN"}, \text{"STILTON"}) = 6$

$\rightsquigarrow lcs(\text{"RUMPLESTILTSKIN"}, \text{"STEAK"} + \text{"STILTON"}) = ?$

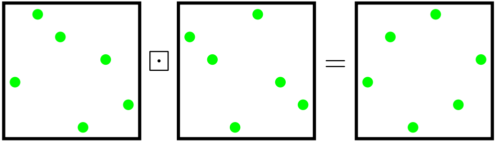
Standard approach: dynamic programming

Divide-and-conquer as an alternative?

Unit-Monge matrices under distance (a.k.a. tropical) multiplication

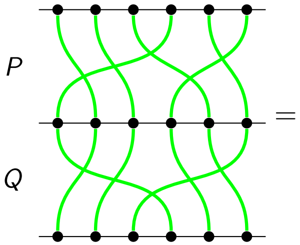
$$\begin{bmatrix}0&1&2&3&4&5&6\\0&1&1&2&3&4&5\\0&1&1&1&2&3&4\\0&1&1&1&2&2&3\\0&0&0&0&1&1&2\\0&0&0&0&1&1&1\\0&0&0&0&0&0&0\end{bmatrix} \odot \begin{bmatrix}0&1&2&3&4&5&6\\0&1&2&3&3&4&5\\0&0&1&2&2&3&4\\0&0&0&1&1&2&3\\0&0&0&1&1&1&2\\0&0&0&1&1&1&1\\0&0&0&0&0&0&0\end{bmatrix} = \begin{bmatrix}0&1&2&3&4&5&6\\0&1&2&3&3&4&5\\0&1&1&2&2&3&4\\0&1&1&2&2&3&3\\0&0&0&1&1&2&2\\0&0&0&1&1&1&1\\0&0&0&0&0&0&0\end{bmatrix}$$

Permutation matrices under sticky multiplication

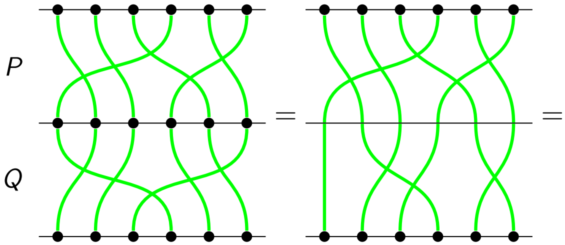


P Q R

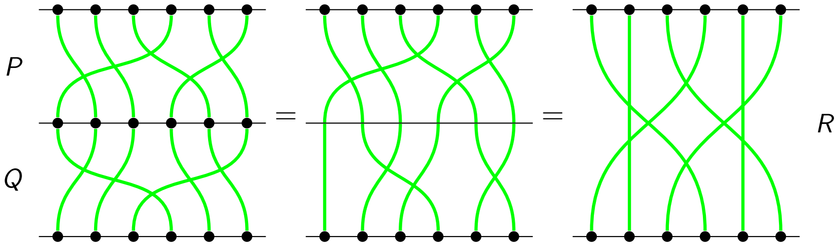
Sticky braids (a.k.a. Hecke words)



Sticky braids (a.k.a. Hecke words)



Sticky braids (a.k.a. Hecke words)



Striking connection between these seemingly unrelated structures:

- behaviour of LCS length under string concatenation
- distance multiplication of unit-Monge matrices = sticky multiplication of permutation matrices
- multiplication of sticky braids

These structures:

- are isomorphic monoids with a deep algebraic meaning
- admit a fast multiplication algorithm
- have far-reaching algorithmic applications
- have connections to fields from computational geometry to combinatorics and statistical mechanics
- have practical applications in bioinformatics

Introduction

Preliminaries

"Squared paper" notation: $x^- = x - \frac{1}{2}$ $x^+ = x + \frac{1}{2}$

Integers $i, j \in \{\ldots -2, -1, 0, 1, 2, \ldots\} = [-\infty : +\infty] \supseteq [i, j]$

Half-integers $\hat{\imath}, \hat{\jmath} \in \{\ldots -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots\} = \langle -\infty : +\infty \rangle \supseteq \langle i, j \rangle$

Planar dominance:

- $(i, j) \ll (i', j')$ iff $i < i'$ and $j < j'$ ("above-left" relation)
- $(i, j) \gtrless (i', j')$ iff $i > i'$ and $j < j'$ ("below-left" relation)

where "above/below" follow matrix convention (not the Cartesian one!)

Strings ($=$ sequences) over an alphabet of size σ

Substrings (contiguous) vs subsequences (not necessarily contiguous)

Prefixes, suffixes (special cases of substring)

Algorithmic problems: input strings a, b of length m, n respectively

String matching: finding an exact pattern in a string

String comparison: finding similar patterns in two strings

- global: whole string a vs whole string b
- semi-local: whole string a vs substrings in b (approximate matching); prefixes in a vs suffixes in b
- local: substrings in a vs substrings in b

String matching: finding an exact pattern in a string

String comparison: finding similar patterns in two strings

- global: whole string a vs whole string b
- semi-local: whole string a vs substrings in b (approximate matching); prefixes in a vs suffixes in b
- local: substrings in a vs substrings in b

We focus on semi-local comparison:

- fundamentally important for both global and local comparison
- exciting mathematical properties

Standard approach to string comparison: dynamic programming

Our approach: the algebra of sticky braids

Can be used either iteratively, or recursively divide-and-conquer

Divide-and-conquer is more efficient for:

- comparing dynamic strings (truncation, concatenation)
- comparing compressed strings (e.g. LZ-compression)
- comparing strings in parallel

The "conquer" step involves a magic "superglue" (efficient sticky braid multiplication)

Dominance-sum matrix (a.k.a. distribution matrix) of matrix D:
$D^{\Sigma}[i, j] = \sum_{\hat{\imath}>i,\hat{\jmath}<j} D\langle i, j\rangle$

Monge and unit-Monge matrices

Monge matrices

Dominance-sum matrix (a.k.a. distribution matrix) of matrix D:
$D^{\Sigma}[i,j] = \sum_{\hat{\imath}>i, \hat{\jmath}<j} D\langle i,j \rangle$

Cross-difference matrix (a.k.a. density matrix) of matrix E:
$E^{\square}\langle \hat{\imath}, \hat{\jmath} \rangle = E[\hat{\imath}^-, \hat{\jmath}^+] - E[\hat{\imath}^-, \hat{\jmath}^-] - E[\hat{\imath}^+, \hat{\jmath}^+] + E[\hat{\imath}^+, \hat{\jmath}^-]$

Dominance-sum matrix (a.k.a. distribution matrix) of matrix D:
$D^{\Sigma}[i,j] = \sum_{\hat{\imath}>i,\hat{\jmath}<j} D\langle i,j\rangle$

Cross-difference matrix (a.k.a. density matrix) of matrix E:
$E^{\square}\langle\hat{\imath},\hat{\jmath}\rangle = E[\hat{\imath}^-,\hat{\jmath}^+] - E[\hat{\imath}^-,\hat{\jmath}^-] - E[\hat{\imath}^+,\hat{\jmath}^+] + E[\hat{\imath}^+,\hat{\jmath}^-]$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^{\square} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Dominance-sum matrix (a.k.a. distribution matrix) of matrix D:
$D^{\Sigma}[i,j] = \sum_{\hat\imath > i, \hat\jmath < j} D\langle i,j\rangle$

Cross-difference matrix (a.k.a. density matrix) of matrix E:
$E^{\square}\langle \hat\imath, \hat\jmath\rangle = E[\hat\imath^-, \hat\jmath^+] - E[\hat\imath^-, \hat\jmath^-] - E[\hat\imath^+, \hat\jmath^+] + E[\hat\imath^+, \hat\jmath^-]$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^{\square} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

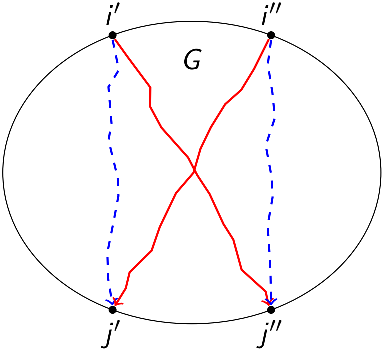
$(D^{\Sigma})^{\square} = D$ for all D

Matrix E is simple, if $(E^{\square})^{\Sigma} = E$: only zeros in left column and bottom row

Matrix E is Monge, if $E^{\square}$ is nonnegative

Intuition: boundary-to-boundary distances in a (weighted) planar graph

G. Monge (1746–1818)



$$E[i',j'] + E[i'',j''] \leq E[i',j''] + E[i'',j']$$

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

Matrix E is unit-Monge, if $E^{\square}$ is a permutation matrix

Intuition: boundary-to-boundary distances in a grid-like graph (in particular, the LCS/alignment grid for a pair of strings)

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

Matrix E is unit-Monge, if $E^{\square}$ is a permutation matrix

Intuition: boundary-to-boundary distances in a grid-like graph (in particular, the LCS/alignment grid for a pair of strings)

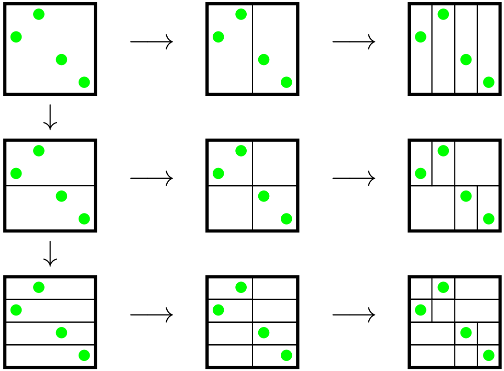
Simple unit-Monge matrix (a.k.a. "rank function"): P^{Σ}, where P is a permutation matrix

P used as implicit representation of P^{Σ}

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Efficient P^{Σ} queries: range tree on nonzeros of P [Bentley: 1980]

- binary search tree by i-coordinate
- under every node, binary search tree by j-coordinate
- empty rectangles not recorded



Efficient P^Σ queries: (contd.)

Every range tree node represents canonical rectangle (Cartesian product of canonical intervals), stores its nonzero count

Overall, $\leq n \log n$ canonical rectangles non-empty

P^Σ element query: $\gtrless$-dominance counting:

- $\sum$ nonzeros $\gtrless$-dominated by query point
- $= \sum$ nonzero counts in $\leq \log^2 n$ disjoint canonical rectangles

Total size $O(n \log n)$, query time $O(\log^2 n)$

Efficient P^{Σ} queries: (contd.)

Every range tree node represents canonical rectangle (Cartesian product of canonical intervals), stores its nonzero count

Overall, $\leq n \log n$ canonical rectangles non-empty

P^{Σ} element query: $\gtrless$-dominance counting:

- $\sum$ nonzeros $\gtrless$-dominated by query point
- $= \sum$ nonzero counts in $\leq \log^2 n$ disjoint canonical rectangles

Total size $O(n \log n)$, query time $O(\log^2 n)$

There are asymptotically more efficient (but less practical) data structures for range counting

Total size $O(n)$, query time $O\left(\frac{\log n}{\log \log n}\right)$ [JáJá+: 2004] [Chan, Pătraşcu: 2010]

Distance semiring (a.k.a. $(\min, +)$-semiring, tropical semiring)

- addition $\oplus$ given by min
- multiplication $\odot$ given by $+$

Matrix distance multiplication

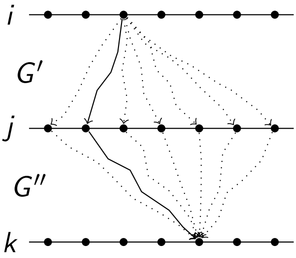
$A \odot B = C \qquad C[i,k] = \bigoplus_j \bigl(A[i,j] \odot B[j,k]\bigr) = \min_j \bigl(A[i,j] + B[j,k]\bigr)$

Intuition: shortest path distances in weighted graphs

Matrix classes closed under $\odot$-multiplication (for given n):

- general (integer, real) matrices $\sim$ general weighted graphs
- Monge matrices $\sim$ planar weighted graphs
- simple unit-Monge matrices $\sim$ grid-like graphs

Intuition: gluing distances in a composition of graphs



Recall: permutation matrices = implicit simple unit-Monge matrices

Matrix sticky multiplication (implicit distance multiplication)

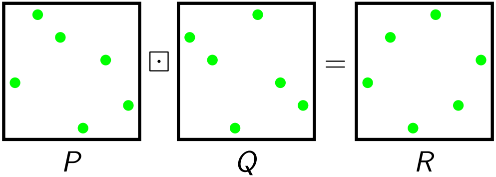
$P \boxdot Q = R$ iff $P^\Sigma \odot Q^\Sigma = R^\Sigma$

The unit-Monge monoid $\mathcal{T}_n$:

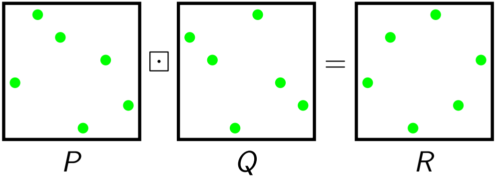
- permutation matrices under $\boxdot$
- simple unit-Monge matrices under $\odot$

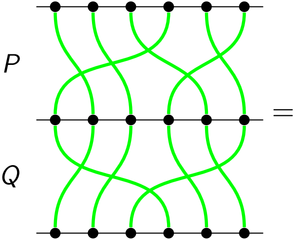
Isomorphic to the Hecke monoid $H_0(\mathcal{S}_n)$

$P \boxdot Q = R$ can be seen as multiplication of sticky braids



$P \boxdot Q = R$ can be seen as multiplication of sticky braids

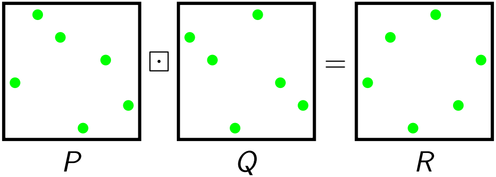


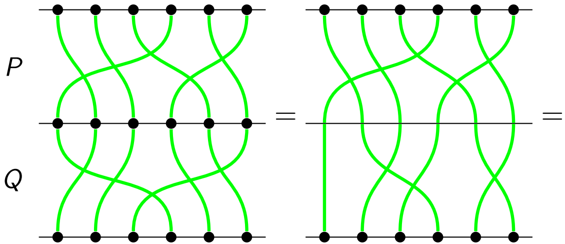


Distance multiplication

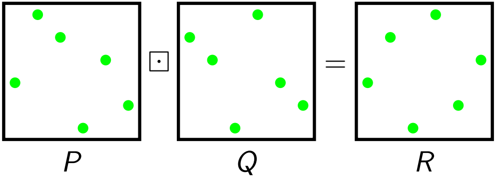
Sticky braids

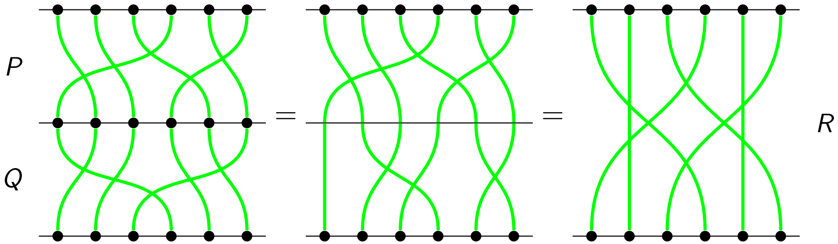
$P \boxdot Q = R$ can be seen as multiplication of sticky braids





$P \boxdot Q = R$ can be seen as multiplication of sticky braids

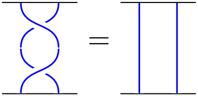




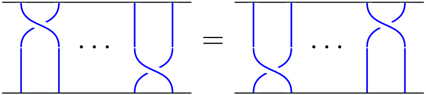
The classical braid group $\mathcal{B}_n$:

- $n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- ∞ elements canonical projection $\mathcal{B}_n \to \mathcal{S}_n$

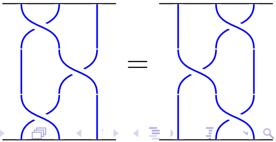
Inversion:
$g_i g_i^{-1} = 1$ for all i



Far commutativity:
$g_i g_j = g_j g_i$ $\quad j-i > 1$



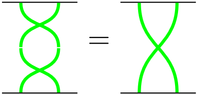
Braid relations:
$g_i g_j g_i = g_j g_i g_j$ $\quad j-i = 1$



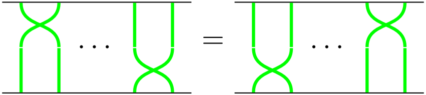
The sticky braid monoid $\mathcal{T}_n$:

- $n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- $n!$ elements canonical bijection $\mathcal{T}_n \leftrightarrow \mathcal{S}_n$

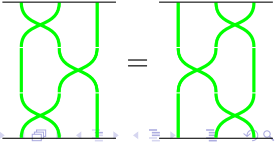
Idempotence:
$g_i^2 = g_i$ for all i



Far commutativity:
$g_i g_j = g_j g_i \quad j - i > 1$



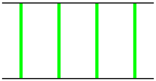
Braid relations:
$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$



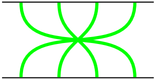
Special elements in $\mathcal{T}_n$

(Denote P^R = counterclockwise rotation of P)

Identity I: $I \boxdot x = x$

$I =$  $= \begin{bmatrix} \bullet & \cdot & \cdot & \cdot \\ \cdot & \bullet & \cdot & \cdot \\ \cdot & \cdot & \bullet & \cdot \\ \cdot & \cdot & \cdot & \bullet \end{bmatrix}$

Zero I^R: $I^R \boxdot x = I^R$

$I^R =$  $= \begin{bmatrix} \cdot & \cdot & \cdot & \bullet \\ \cdot & \cdot & \bullet & \cdot \\ \cdot & \bullet & \cdot & \cdot \\ \bullet & \cdot & \cdot & \cdot \end{bmatrix}$

Zero divisors: $P^R \boxdot P = P \boxdot P^{RRR} = I^R$ for all P

Matrix sticky multiplication

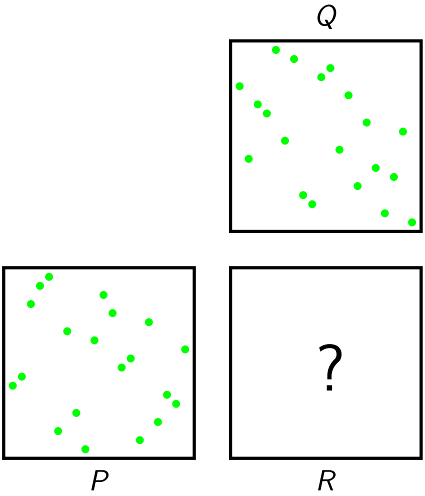
Given permutation matrices P, Q, compute R, such that $P \boxdot Q = R$

Matrix sticky multiplication

Given permutation matrices P, Q, compute R, such that $P \boxdot Q = R$

Matrix distance and sticky multiplication: running time

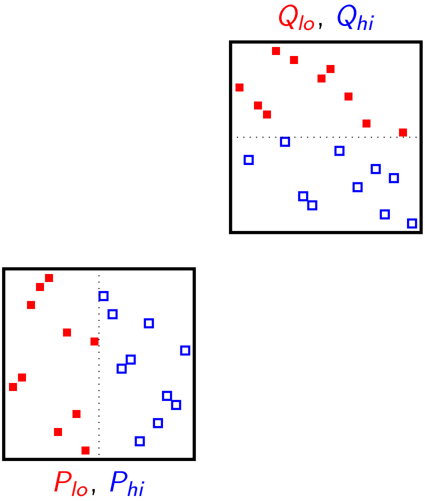
type	time	
general $\odot$	$O(n^3)$	standard
	$O\left(\frac{n^3(\log\log n)^3}{\log^2 n}\right)$	[Chan: 2007]
Monge $\odot$	$O(n^2)$	via [Aggarwal+: 1987]
permutation $\boxdot$	$O(n^{1.5})$	[T: 2006]
	$O(n \log n)$	[T: 2010]



Q
?
P
R

Distance multiplication

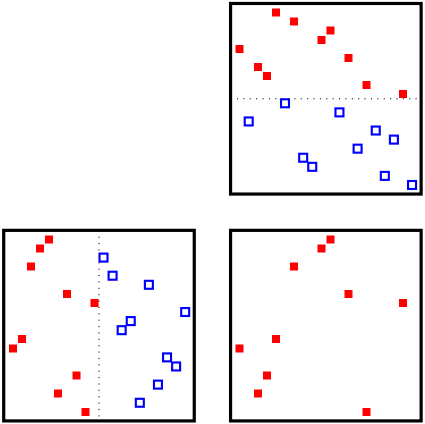
Matrix sticky multiplication



Distance multiplication

Matrix sticky multiplication

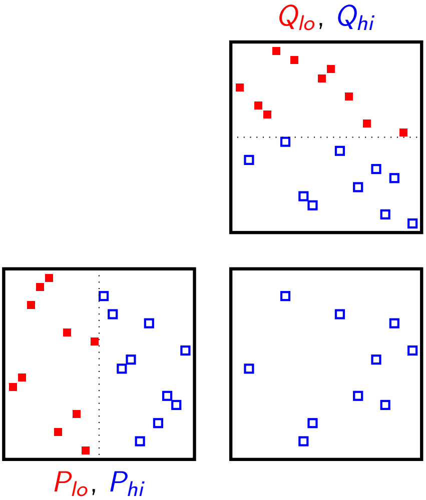
Q_{lo}, Q_{hi}



P_{lo}, P_{hi}

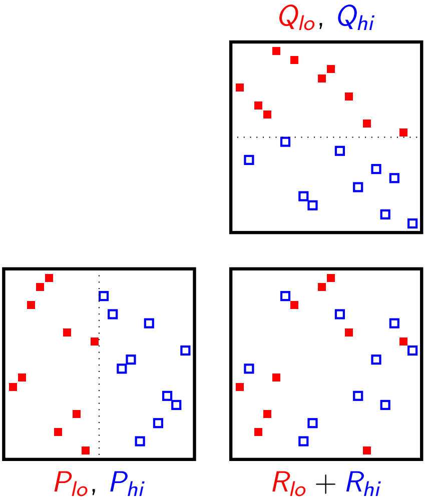
Distance multiplication

Matrix sticky multiplication



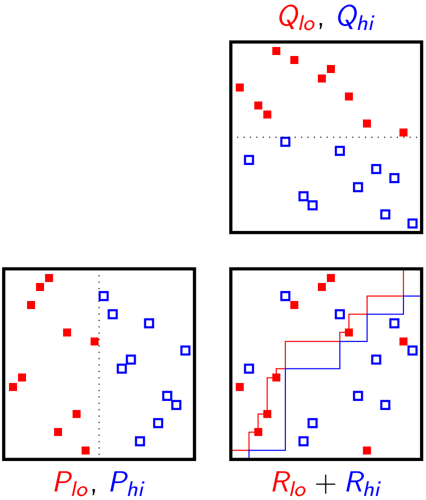
Distance multiplication

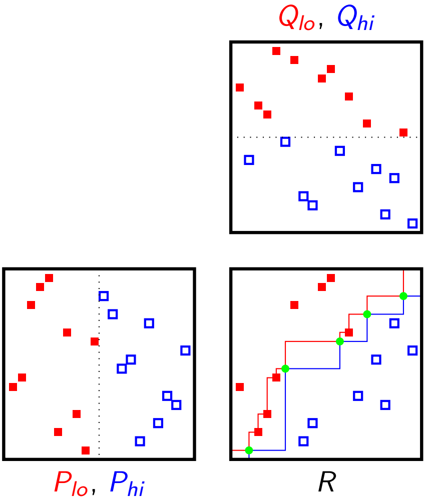
Matrix sticky multiplication



Distance multiplication

Matrix sticky multiplication

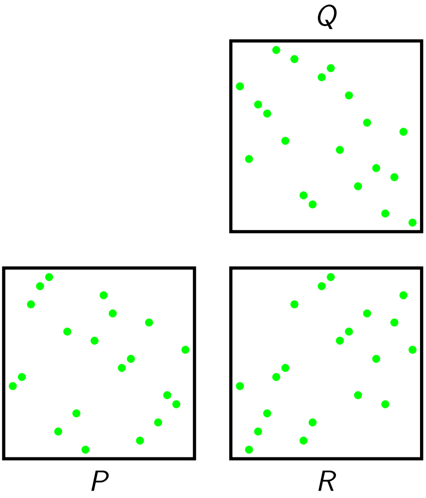




Q_{lo}, Q_{hi}
P_{lo}, P_{hi}
R

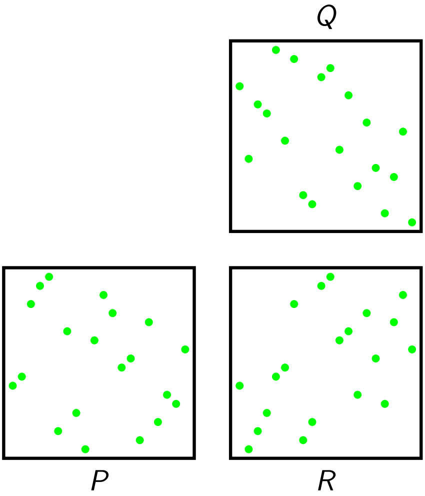
Distance multiplication

Matrix sticky multiplication



Distance multiplication

Matrix sticky multiplication



Matrix sticky multiplication: Steady Ant algorithm

$P \boxdot Q = R \qquad R^\Sigma(i,k) = \min_j \big(P^\Sigma(i,j) + Q^\Sigma(j,k)\big)$

Divide: split range of j $\rightsquigarrow$ two recursive subproblems on $n/2$-matrices

$P_{lo} \boxdot Q_{lo} = R_{lo} \qquad P_{hi} \boxdot Q_{hi} = R_{hi}$

Each subproblem determines good nonzeros, remaining in main problem's solution

Conquer: trace border path through range of i, k (bottom-left to top-right of R), separating good nonzeros of one subproblem from the other

Border path invariant: balance condition on bad nonzeros
$|\{\text{nonzeros of } R_{hi} \text{ above-left}\}| = |\{\text{nonzeros of } R_{lo} \text{ below-right}\}|$

Step through border path: can maintain invariant in time $O(1)$ per step

Keep all good nonzeros; replace bad nonzeros by fresh nonzeros on border path

Conquer time $O(n)$ Overall time $O(n \log n)$

Longest common subsequence

LCS problem

a, b: strings of length m, n

The longest common subsequence (LCS) score:

- length of longest string that is a subsequence of both a and b
- in computational biology, unweighted alignment
- in ergodic theory, used to define the Feldman–Katok metric
- in software engineering, the diff tool

$lcs(\text{"BAABCBCA"}, \text{"CABCABA"}) = length(\text{"ABCBA"}) = 5$

LCS problem

LCS score for a vs b

LCS problem

LCS score for a vs b

LCS: running time

$O(mn)$	[Wagner, Fischer: 1974]
$O\left(\frac{mn}{(\log n)^c}\right)$	[Masek, Paterson: 1980] [Crochemore+: 2003]
	[Paterson, Dančík: 1994] [Bille, Farach-Colton: 2008]
No $O\left((mn)^{1-\epsilon}\right)$ $\quad \epsilon > 0$; assuming SETH	[Abboud+: 2015]
	[Backurs, Indyk: 2015]

Polylog's exponent c depends on alphabet size and computation model

LCS: classical dynamic programming (DP)

Iterate over cells in any $\ll$-compatible order

Active cell update: time $O(1)$

Overall time $O(mn)$

LCS: DP, micro-block speedup (MBS) [MP: 1980; BF: 2008]

Iterate over cells in micro-blocks, in any $\ll$-compatible order

Micro-block size:

- $t = O(\log n)$ when $\sigma = O(1)$
- $t = O\left(\frac{\log n}{\log \log n}\right)$ otherwise

Micro-block interface:

- $O(t)$ characters, each $O(\log \sigma)$ bits, can be reduced to $O(\log t)$ bits
- $O(t)$ small integers, each $O(1)$ bits

Micro-block update: time $O(1)$, by precomputing all possible interfaces

Overall time $O\left(\frac{mn}{\log^2 n}\right)$ when $\sigma = O(1)$, $O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right)$ otherwise

'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

L. Carroll, *Alice in Wonderland*

'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

L. Carroll, *Alice in Wonderland*

Dynamic programming: begins at empty strings, proceeds by appending characters, then stops

What about:

- prepending/deleting characters (dynamic LCS)
- concatenating strings (LCS on compressed strings; parallel LCS)
- taking substrings (= local alignment)

Longest common subsequence

LCS problem

Running DP from both ends: better by $\times 2$, but still not good enough

Is dynamic programming strictly necessary to solve sequence alignment problems?

Eppstein+, *Efficient algorithms for sequence analysis*, 1991

Semi-local LCS problem

LCS scores for a vs b:

- string-substring (whole a vs every substring of b)
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

Output scores can be represented implicitly

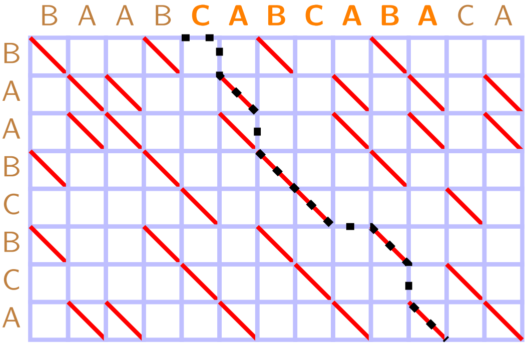
Semi-local LCS problem

LCS scores for a vs b:

- string-substring (whole a vs every substring of b)
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

Output scores can be represented implicitly

Semi-local LCS as maximum paths in the LCS grid



$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"

$blue = 0$ (skip)
$red = 1$ (match)

$lcs(a, b\langle 4 : 11\rangle) = 5$

String-substring LCS: all highest-score top-to-bottom paths

Semi-local LCS: all highest-score boundary-to-boundary paths

Semi-local LCS: output representation and running time

size	query time		
$O(n^2)$	$O(1)$	string-substring	trivial
$O(m^{1/2}n)$	$O(\log n)$	string-substring	[Alves+: 2003]
$O(n)$	$O(n)$	string-substring	[Alves+: 2005]
$O(n \log n)$	$O(\log^2 n)$		[T: 2006]

. . . or any 2D orthogonal range counting data structure

running time		
$O(mn^2) = O(n \cdot mn)$	string-substring	repeated DP
$O(mn)$	string-substring	[Schmidt: 1998; Alves+: 2005]
$O(mn)$		[T: 2006]
$O\left(\frac{mn}{(\log n)^{O(1)}}\right)$		[T: 2006–07]

LCS matrix H and LCS kernel P

$H[i,j]$: max number of matched characters for a vs substring $b\langle i:j\rangle$

$j-i-H[i,j]$: min number of unmatched characters

Properties of matrix $j-i-H[i,j]$:

- simple unit-Monge
- therefore, $= P^{\Sigma}$, where $P = -H^{\square}$ is a permutation matrix

P is the LCS kernel, giving an implicit representation of H

Range tree for P: memory $O(n \log n)$, query time $O(\log^2 n)$

LCS matrix H and LCS kernel P (only string-substring component shown)

0	1	2	3	4	5	6	6	7	8	8	8	8	8
-1	0	1	2	3	4	5	5	6	7	7	7	7	7
-2	-1	0	1	2	3	4	4	5	6	6	6	6	7
-3	-2	-1	0	1	2	3	3	4	5	5	6	6	7
-4	-3	-2	-1	0	1	2	2	3	4	4	5	5	6
-5	-4	-3	-2	-1	0	1	2	3	4	4	5	5	6
-6	-5	-4	-3	-2	-1	0	1	2	3	3	4	4	5
-7	-6	-5	-4	-3	-2	-1	0	1	2	2	3	3	4
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	3	4
-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"

$$H[i,j] = \begin{cases} lcs(a, b\langle i:j\rangle) & i \leq j \\ j - i & i \geq j \end{cases}$$

$H[0,13] = lcs(a,b) = 8$
$H[4,11] = lcs(a, b\langle 4:11\rangle) = 5$

Longest common subsequence

Semi-local LCS problem

LCS matrix H and LCS kernel P (only string-substring component shown)

0	1	2	3	4	5	6	6	7	8	8	8	8	8
-1	0	1	2	3	4	5	5	6	7	7	7	7	7
-2	-1	0	1	2	3	4	4	5	6	6	6	6	7
-3	-2	-1	0	1	2	3	3	4	5	5	6	6	7
-4	-3	-2	-1	0	1	2	2	3	4	4	5	5	6
-5	-4	-3	-2	-1	0	1	2	3	4	4	5	5	6
-6	-5	-4	-3	-2	-1	0	1	2	3	3	4	4	5
-7	-6	-5	-4	-3	-2	-1	0	1	2	2	3	3	4
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	3	4
-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

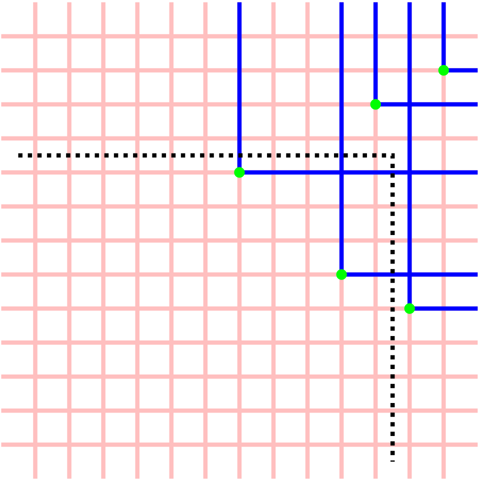
$$H[i,j] = \begin{cases} lcs(a, b\langle i : j\rangle) & i \le j \\ j - i & i \ge j \end{cases}$$

blue | *red*: diff in $H = 0 \mid 1$

green: $P\langle i,j\rangle = 1$

$$H[i,j] = j - i - P^{\Sigma}[i,j]$$

LCS matrix H and LCS kernel P (only string-substring component shown)

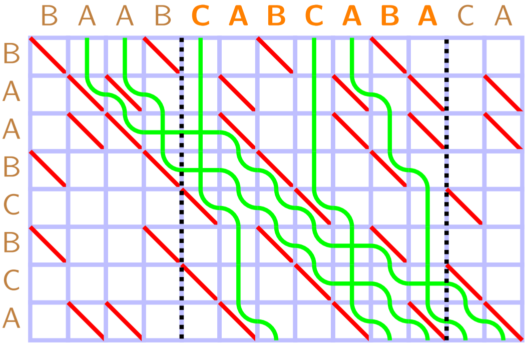


$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"

$$H[i,j] = \begin{cases} lcs(a, b\langle i : j\rangle) & i \leq j \\ j - i & i \geq j \end{cases}$$

$H[4, 11] = 11 - 4 - P^{\Sigma}[i,j] = 11 - 4 - 2 = 5$

LCS kernel as (reduced) sticky braid in LCS grid

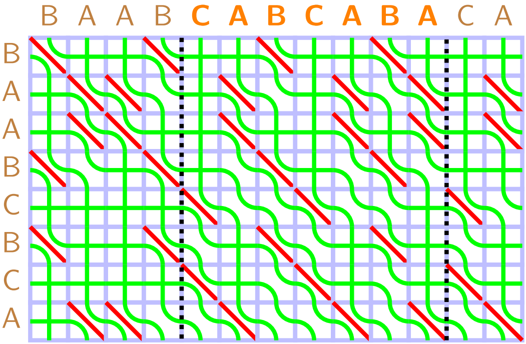


$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"
$H[4, 11] = 11 - 4 - P^{\Sigma}[i, j] = 11 - 4 - 2 = 5$

String-substring LCS: $P\langle i, j \rangle = 1$ iff strand i *(top)* $\rightsquigarrow j$ *(bottom)*

Each strand is a unit obstruction to LCS, if crossed left-to-right

LCS kernel as (reduced) sticky braid in LCS grid



$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"

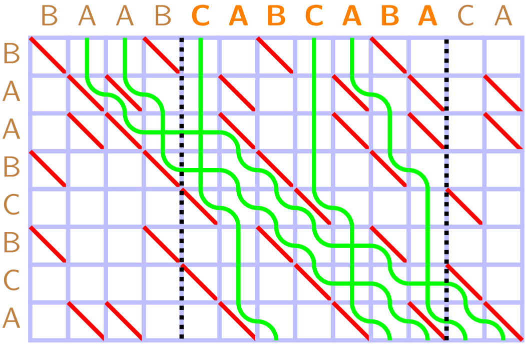
$H[4,11] = 11 - 4 - P^{\Sigma}[i,j] = 11 - 4 - 2 = 5$

Semi-local LCS: $P\langle i,j \rangle = 1$ iff strand i *(top/left)* $\rightsquigarrow$ j *(bottom/right)*

Embedded sticky braid: different strand embeddings possible

LCS kernel: braid with no particular embedding (one shown arbitrarily in pictures)

LCS kernel as (reduced) sticky braid in LCS grid



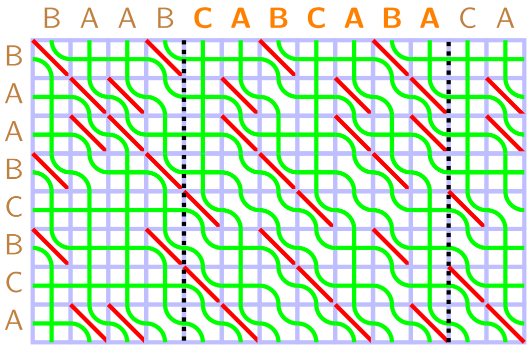
$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"

$H[4, 11] = 11 - 4 - P^{\Sigma}[i, j] =$
$11 - 4 - 2 = 5$

String-substring LCS: $P\langle i, j \rangle = 1$ iff strand i *(top)* $\rightsquigarrow j$ *(bottom)*

Each strand is a unit obstruction to LCS, if crossed left-to-right

LCS kernel as (reduced) sticky braid in LCS grid



$a =$ "BAABCBCA"
$b =$ "BAAB**CABCABA**CA"
$H[4,11] = 11 - 4 - P^{\Sigma}[i,j] = 11 - 4 - 2 = 5$

Semi-local LCS: $P\langle i,j\rangle = 1$ iff strand i *(top/left)* $\rightsquigarrow$ j *(bottom/right)*

Different strand embeddings possible: embedded sticky braid

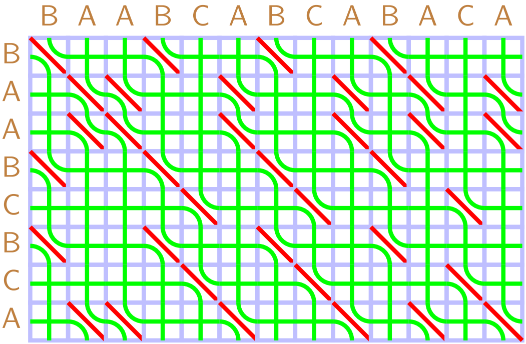
LCS kernel: no particular embedding (but one still chosen in pictures)

Sticky braid: a highly symmetric object (Hecke word $\in H_0(S_n)$)

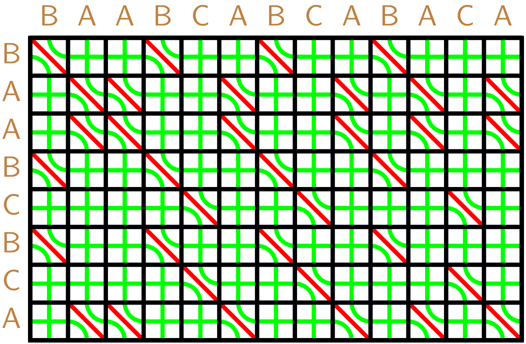
Can be built by assembling subbraids: divide-and-conquer

Flexible approach to local alignment, compressed approximate matching, parallel computation. . .

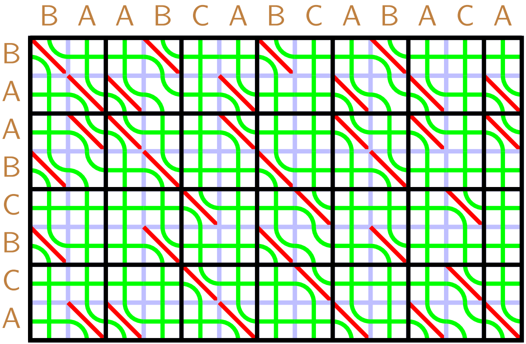
Semi-local LCS by recursive combing



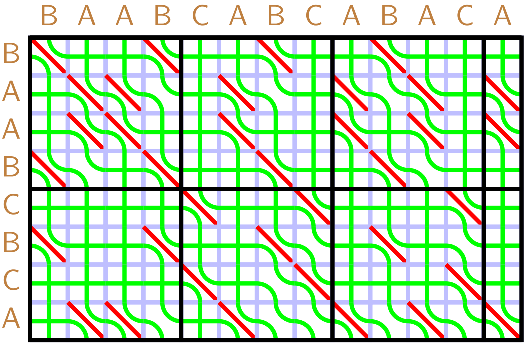
Semi-local LCS by recursive combing



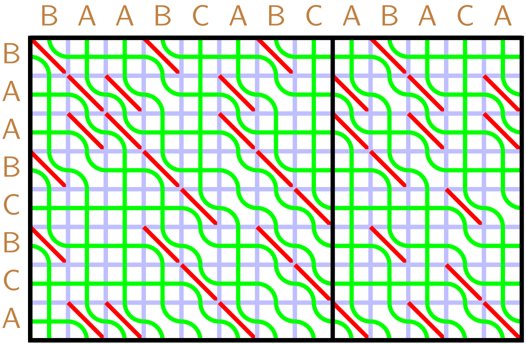
Semi-local LCS by recursive combing



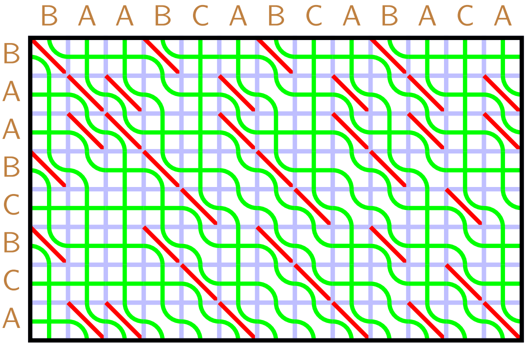
Semi-local LCS by recursive combing



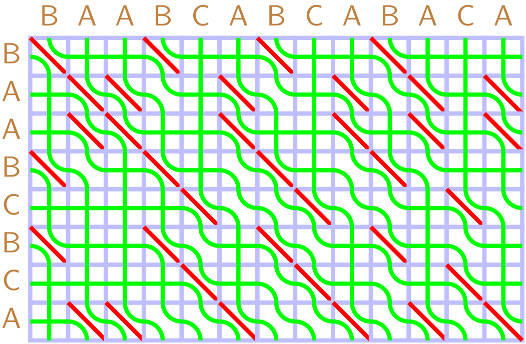
Semi-local LCS by recursive combing



Semi-local LCS by recursive combing



Semi-local LCS by recursive combing



Hierarchy of kernels, no particular embedding

Semi-local LCS: recursive combing

Initialise uncombed sticky braid: mismatch cell = crossing

Recursion on LCS grid

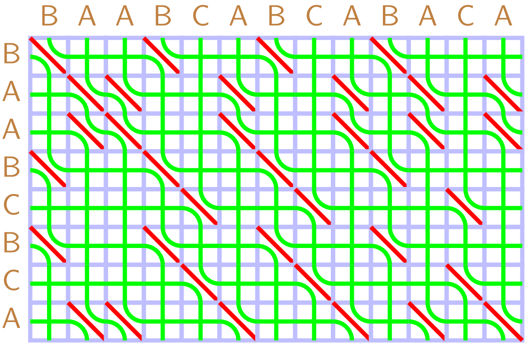
- divide: partition either a or b
- obtain subproblem LCS kernels recursively
- conquer: LCS kernel composition by sticky multiplication

Recursion base: $m = n = 1$

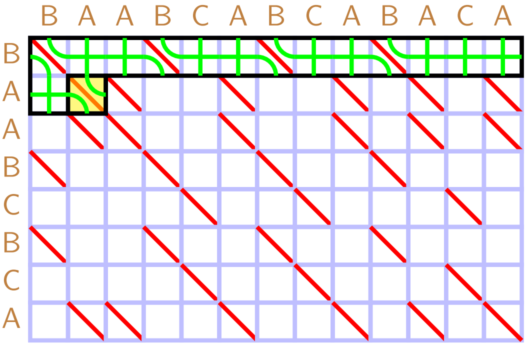
Overall time $O(mn)$

Correctness: by sticky braid relations

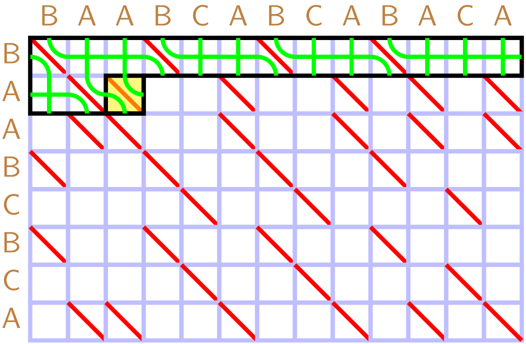
Semi-local LCS by iterative combing



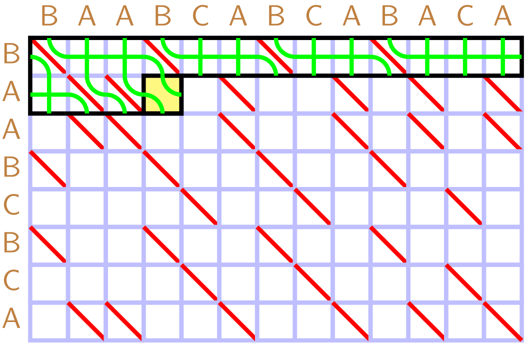
Semi-local LCS by iterative combing



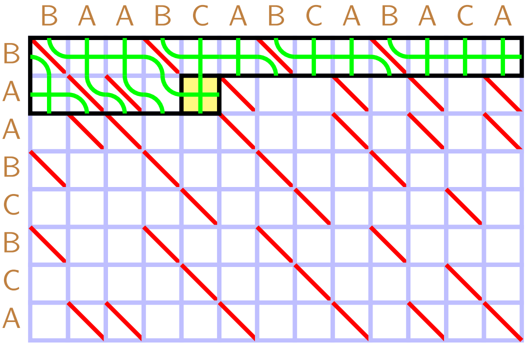
Semi-local LCS by iterative combing



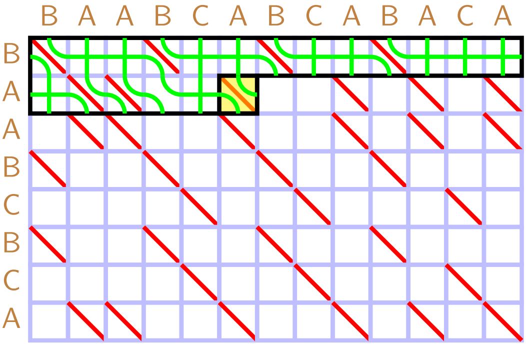
Semi-local LCS by iterative combing



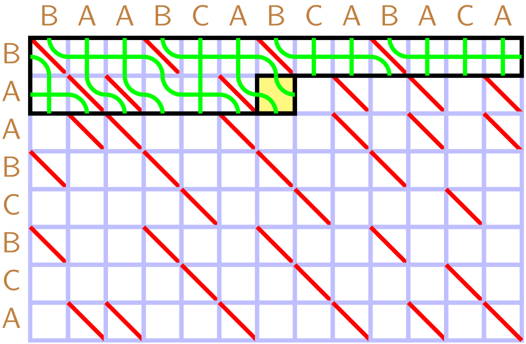
Semi-local LCS by iterative combing



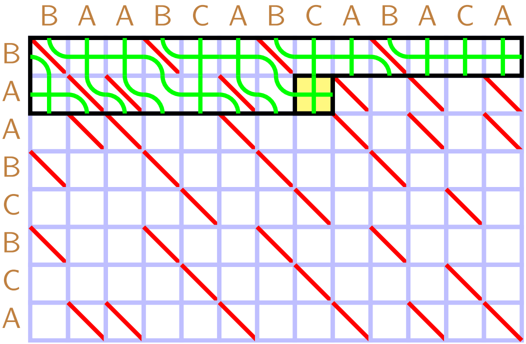
Semi-local LCS by iterative combing



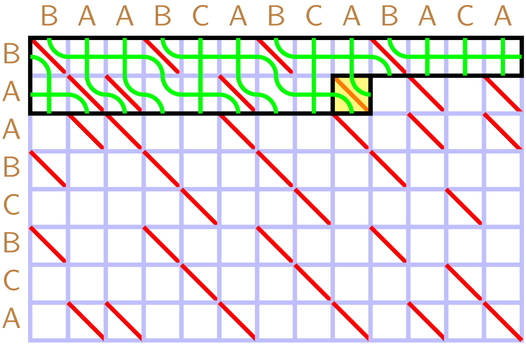
Semi-local LCS by iterative combing



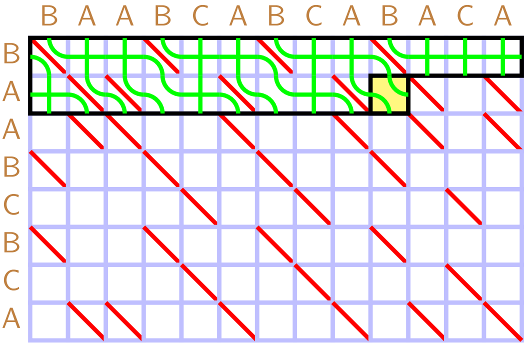
Semi-local LCS by iterative combing



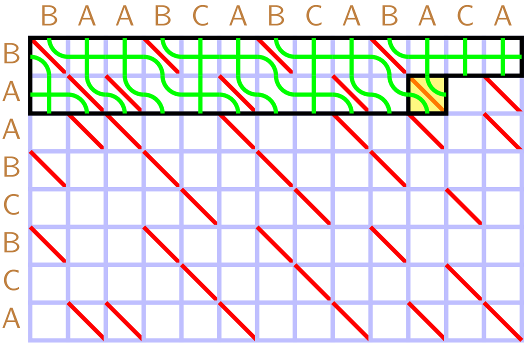
Semi-local LCS by iterative combing



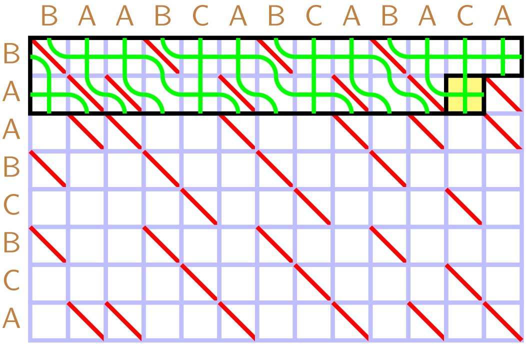
Semi-local LCS by iterative combing



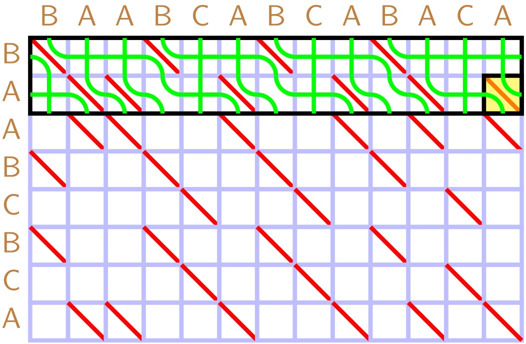
Semi-local LCS by iterative combing



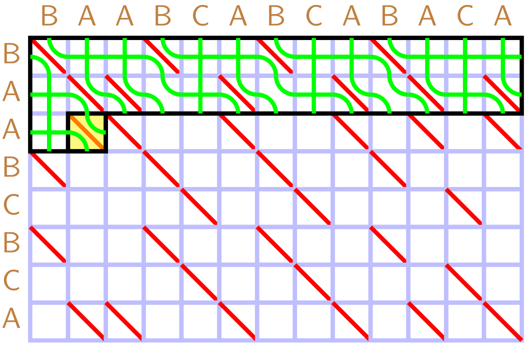
Semi-local LCS by iterative combing



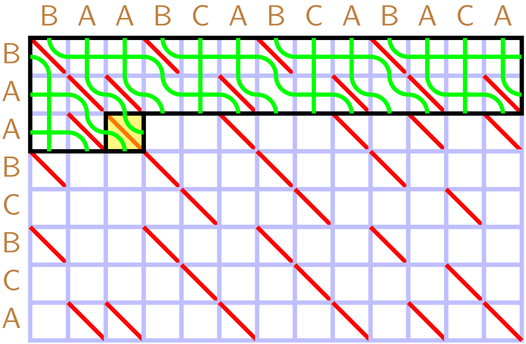
Semi-local LCS by iterative combing



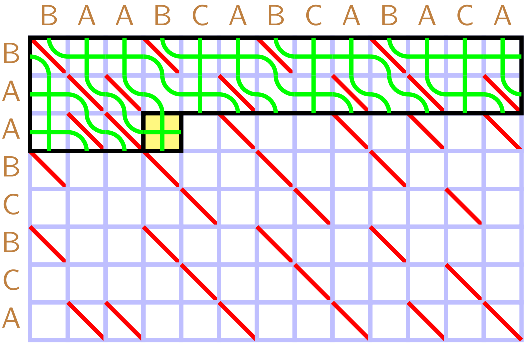
Semi-local LCS by iterative combing



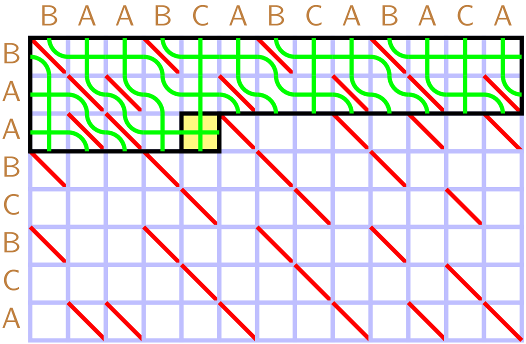
Semi-local LCS by iterative combing



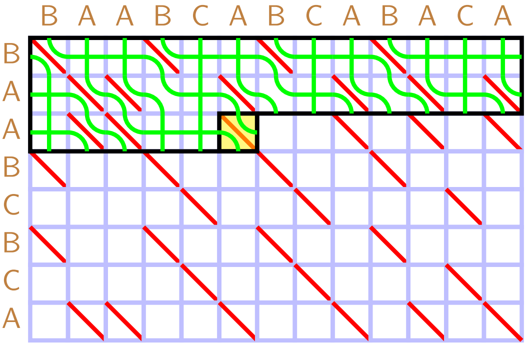
Semi-local LCS by iterative combing



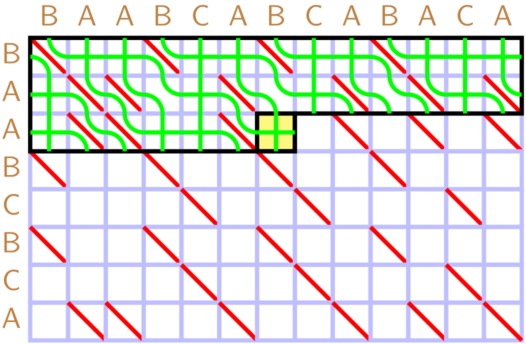
Semi-local LCS by iterative combing



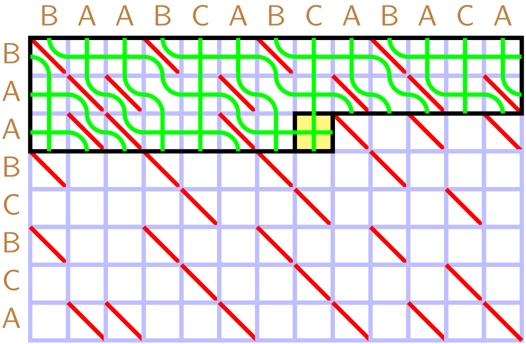
Semi-local LCS by iterative combing



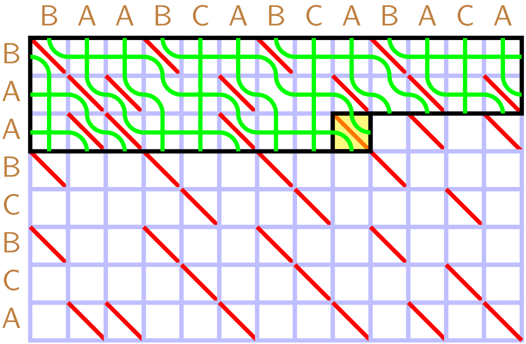
Semi-local LCS by iterative combing



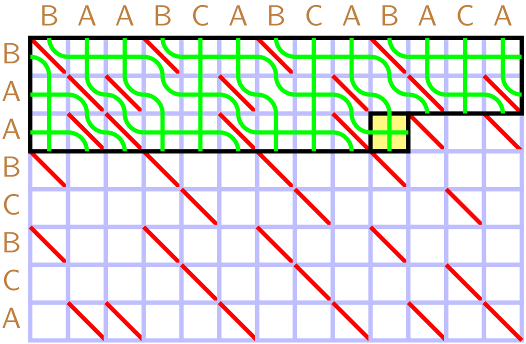
Semi-local LCS by iterative combing



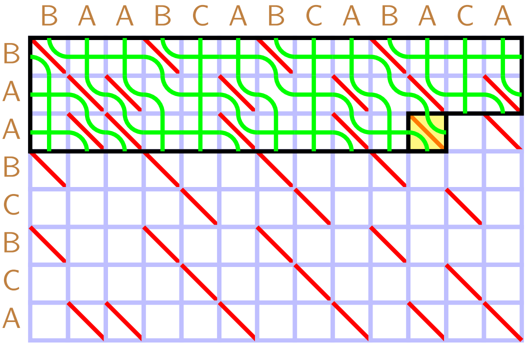
Semi-local LCS by iterative combing



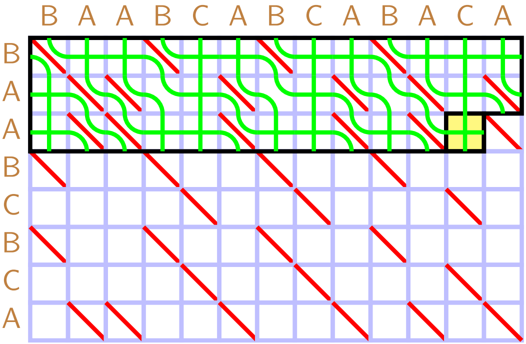
Semi-local LCS by iterative combing



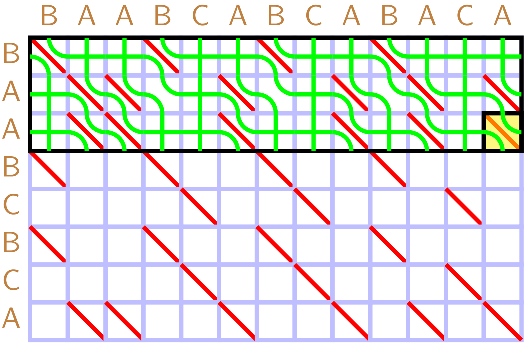
Semi-local LCS by iterative combing



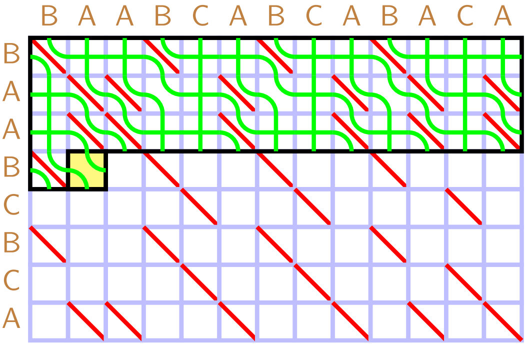
Semi-local LCS by iterative combing



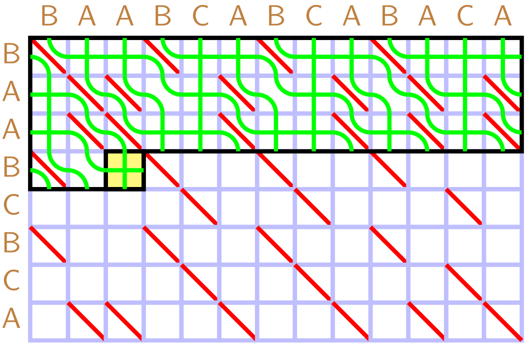
Semi-local LCS by iterative combing



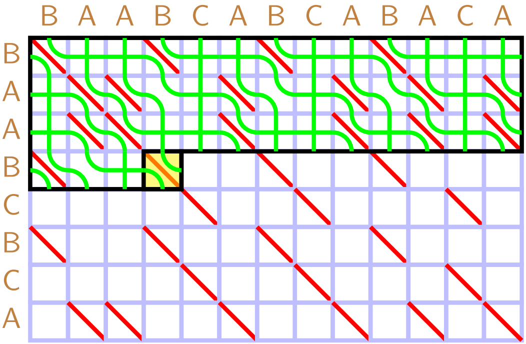
Semi-local LCS by iterative combing



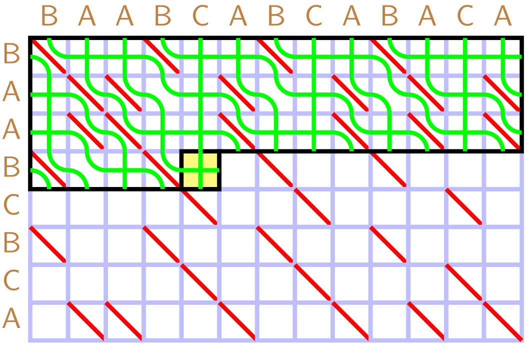
Semi-local LCS by iterative combing



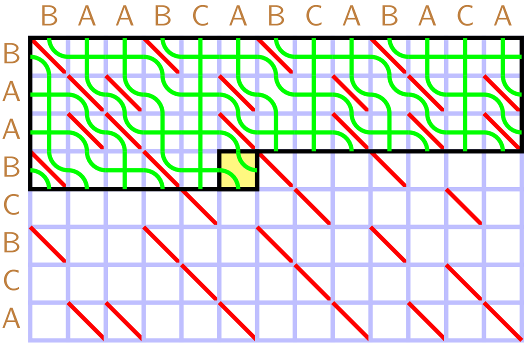
Semi-local LCS by iterative combing



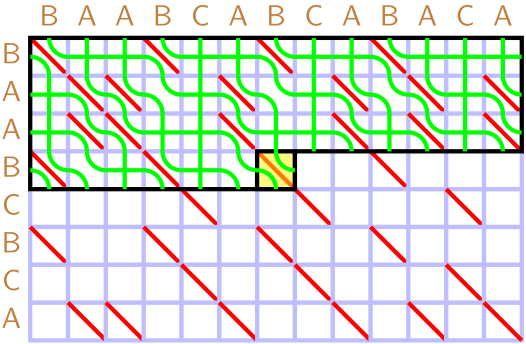
Semi-local LCS by iterative combing



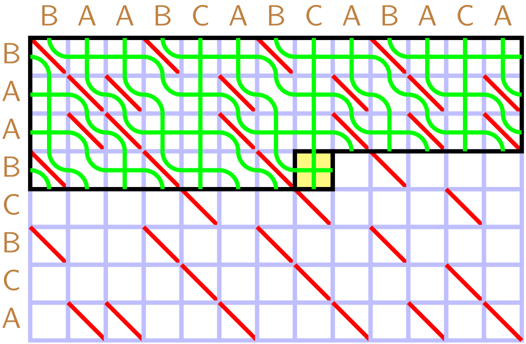
Semi-local LCS by iterative combing



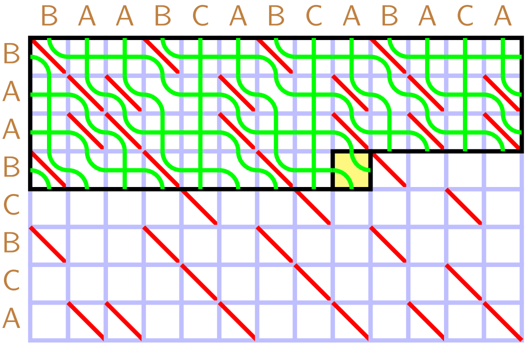
Semi-local LCS by iterative combing



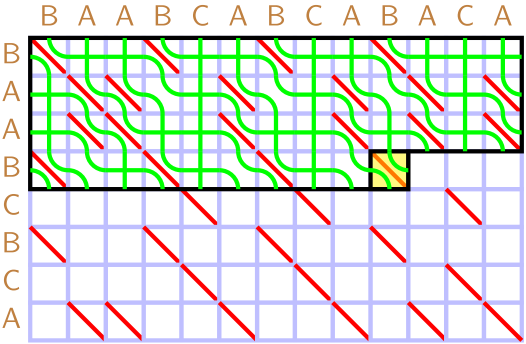
Semi-local LCS by iterative combing



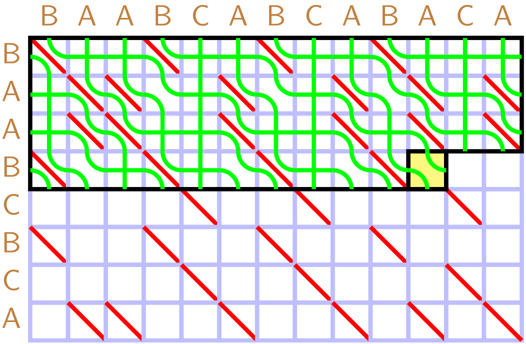
Semi-local LCS by iterative combing



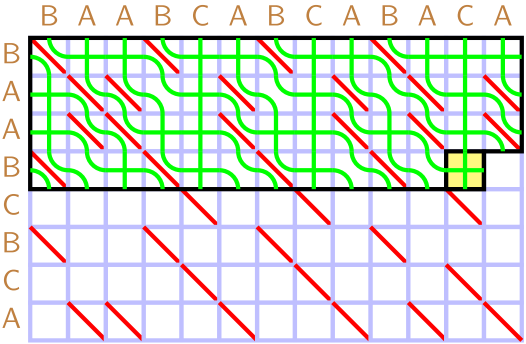
Semi-local LCS by iterative combing



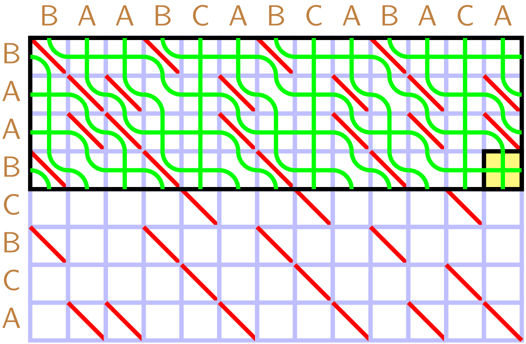
Semi-local LCS by iterative combing



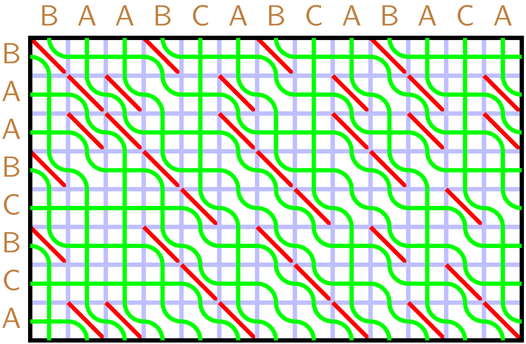
Semi-local LCS by iterative combing



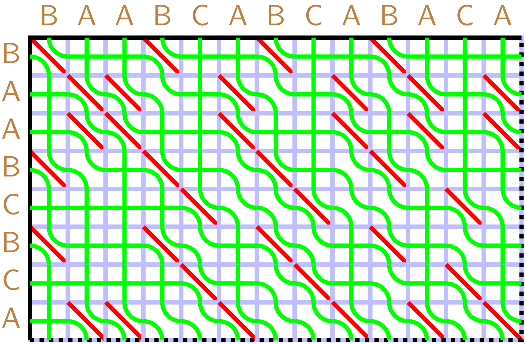
Semi-local LCS by iterative combing



Semi-local LCS by iterative combing



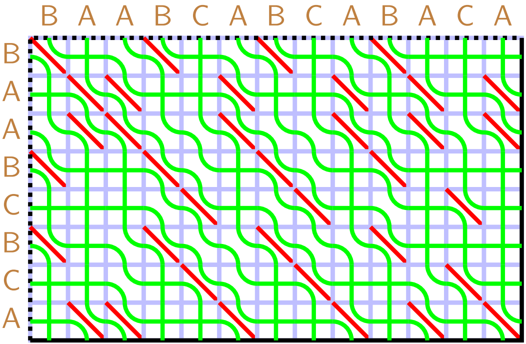
Semi-local LCS by iterative combing



Forward embedded braid: kernel for every prefix of *a* vs every prefix of *b*

Implicit prefix-substring and substring-prefix LCS

Semi-local LCS by iterative combing



Backward embedded braid: kernel for every suffix of *a* vs every suffix of *b*

Implicit suffix-substring and substring-suffix LCS

Semi-local LCS: iterative combing

Initialise uncombed sticky braid: mismatch cell = crossing

Iterate over cells in any $\ll$-compatible order

- match cell: skip (keep strands uncrossed)
- mismatch cell: comb (uncross if strands crossed before)

Active cell update: time $O(1)$

Overall time $O(mn)$

Correctness: by sticky braid relations

Further string comparison
Periodic LCS

a: string of length m u: string of length p

Periodic string-substring LCS problem

LCS scores: a vs every substring of $b = \ldots uuu \ldots = u^{\infty}$

Output scores can be represented implicitly

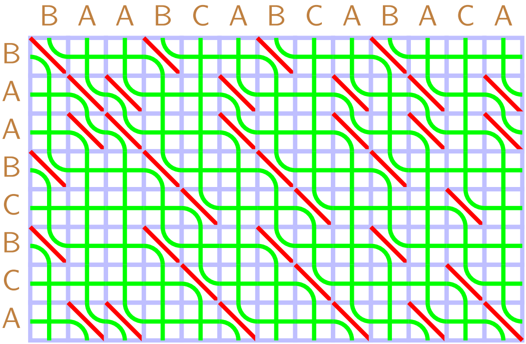
May assume that every character of a occurs in u (otherwise delete it)

Only need substrings in b of length $\leq mp$ (otherwise LCS score $= m$)

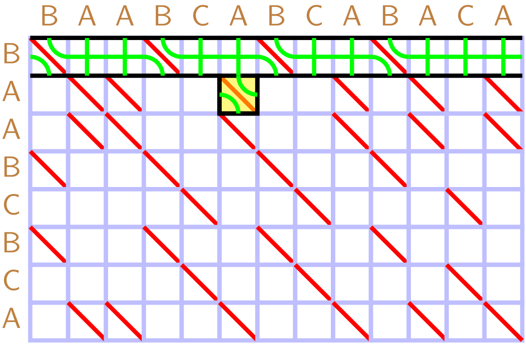
Periodic string-substring LCS: running time

$O(mnp)$	naive
$O(mp)$	[T: 2009]

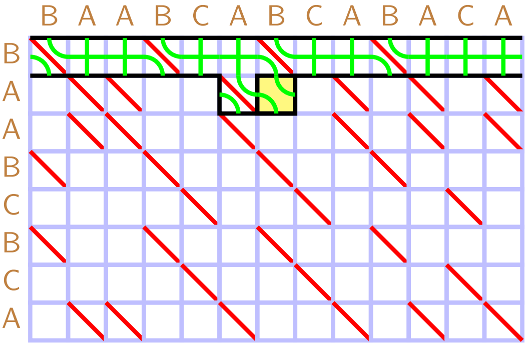
Periodic string-substring LCS by wraparound combing



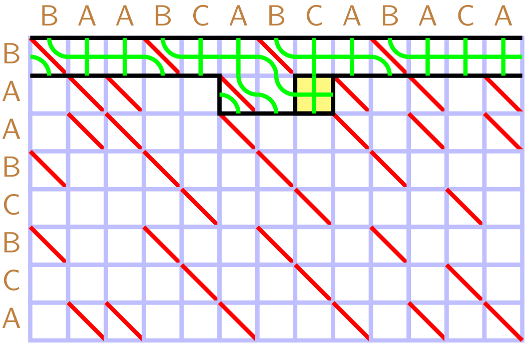
Periodic string-substring LCS by wraparound combing



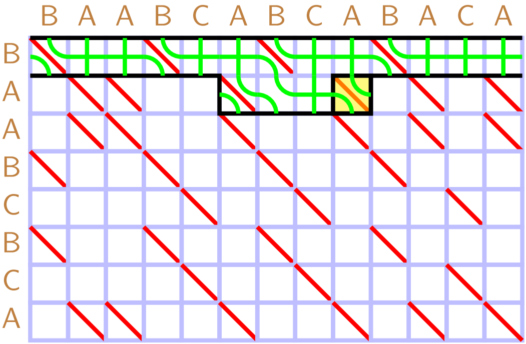
Periodic string-substring LCS by wraparound combing



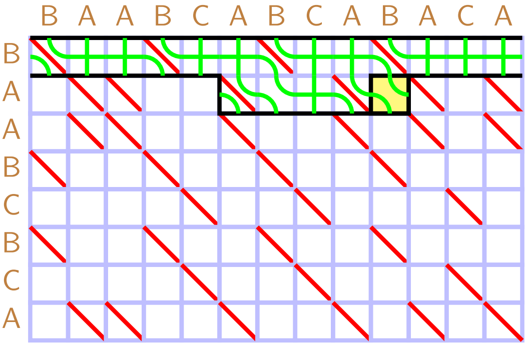
Periodic string-substring LCS by wraparound combing



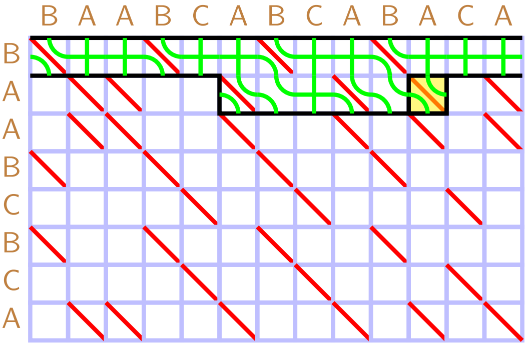
Periodic string-substring LCS by wraparound combing



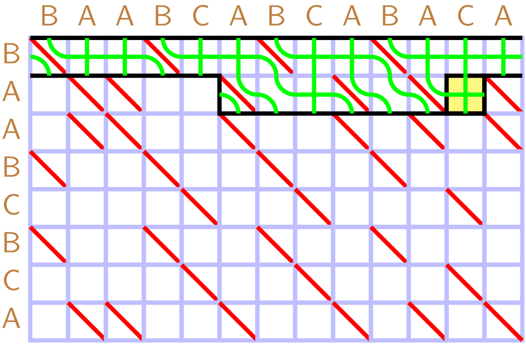
Periodic string-substring LCS by wraparound combing



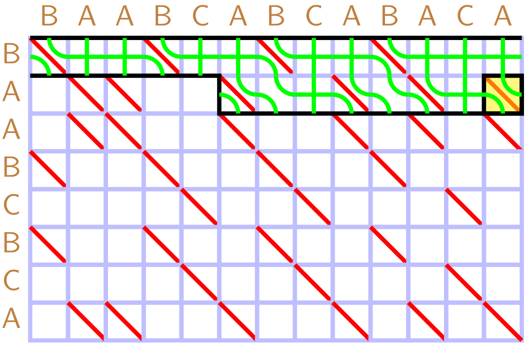
Periodic string-substring LCS by wraparound combing



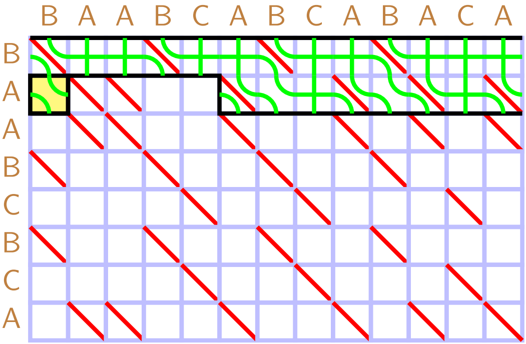
Periodic string-substring LCS by wraparound combing



Periodic string-substring LCS by wraparound combing

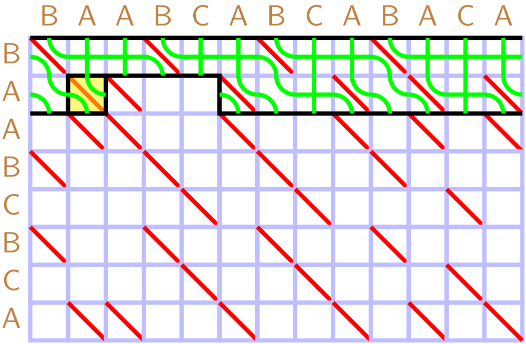


Periodic string-substring LCS by wraparound combing

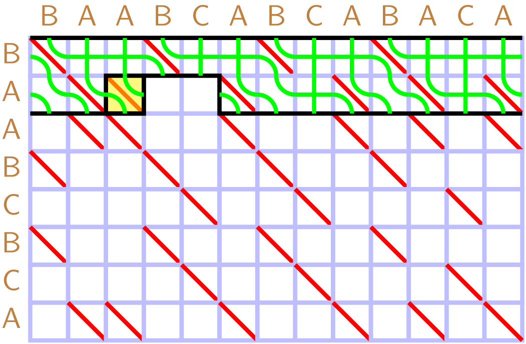


Further string comparison
Periodic LCS

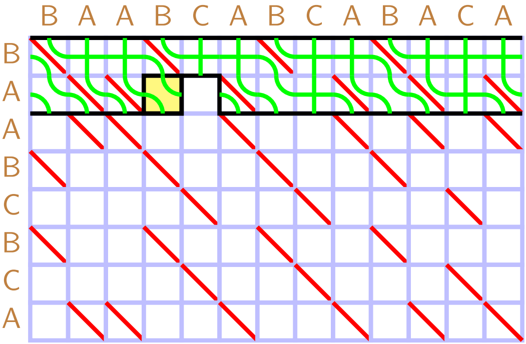
Periodic string-substring LCS by wraparound combing



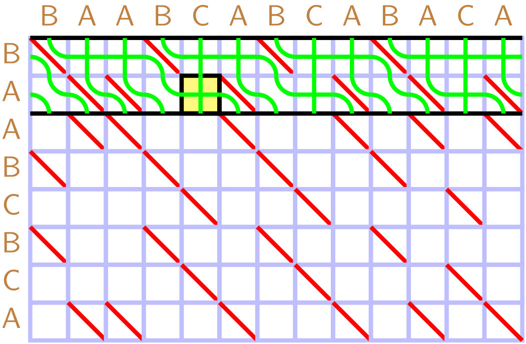
Periodic string-substring LCS by wraparound combing



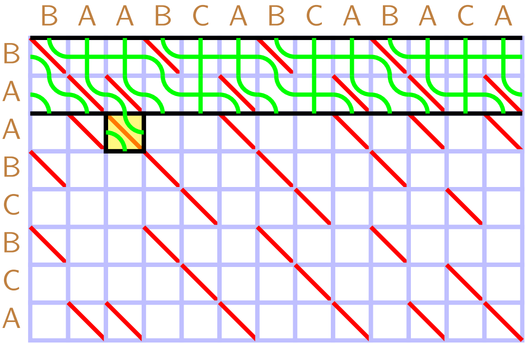
Periodic string-substring LCS by wraparound combing



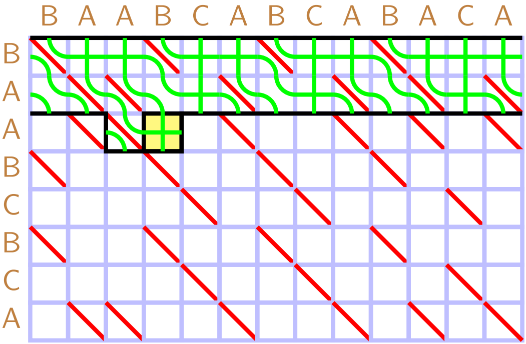
Periodic string-substring LCS by wraparound combing



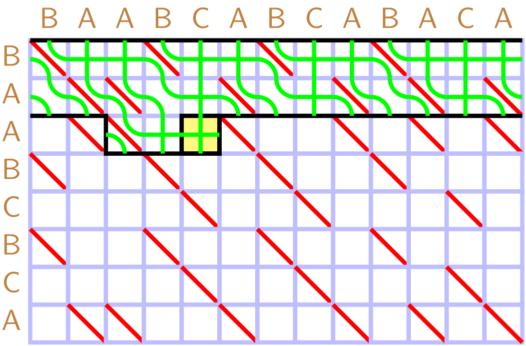
Periodic string-substring LCS by wraparound combing



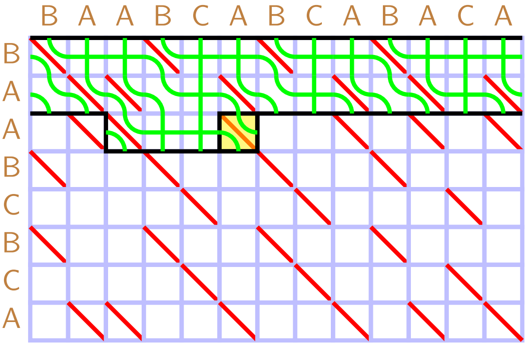
Periodic string-substring LCS by wraparound combing



Periodic string-substring LCS by wraparound combing

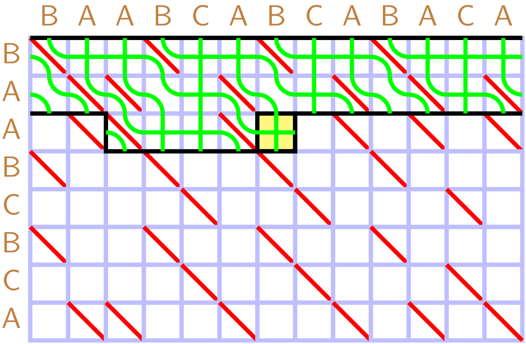


Periodic string-substring LCS by wraparound combing

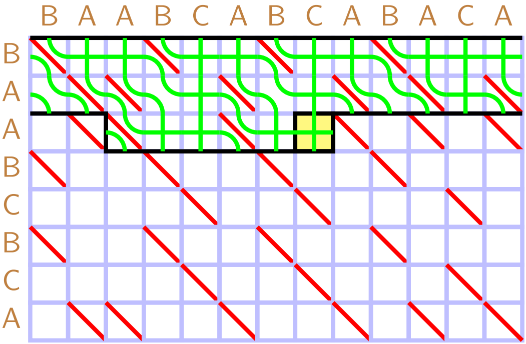


Further string comparison
Periodic LCS

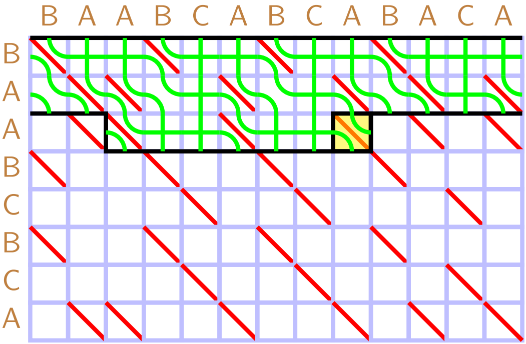
Periodic string-substring LCS by wraparound combing



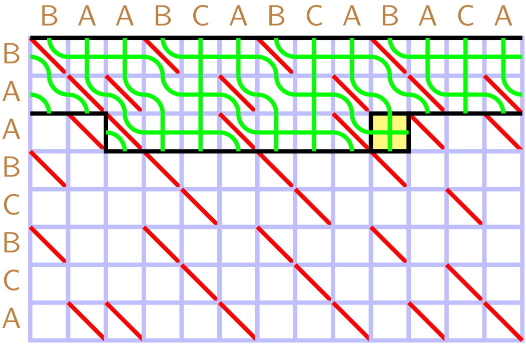
Periodic string-substring LCS by wraparound combing



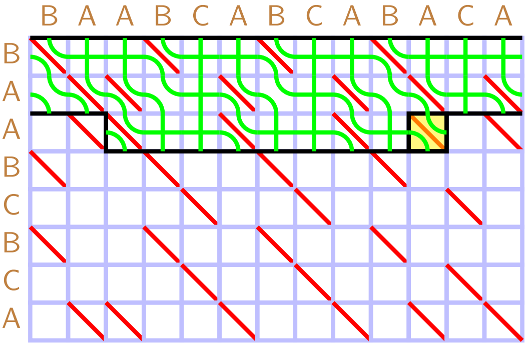
Periodic string-substring LCS by wraparound combing



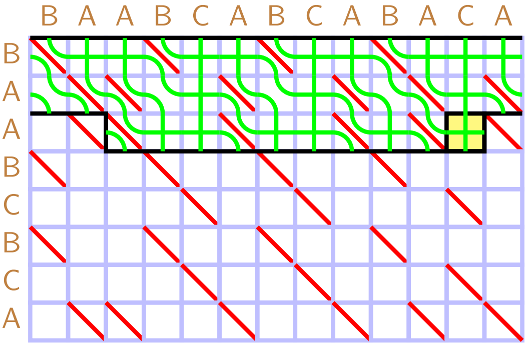
Periodic string-substring LCS by wraparound combing



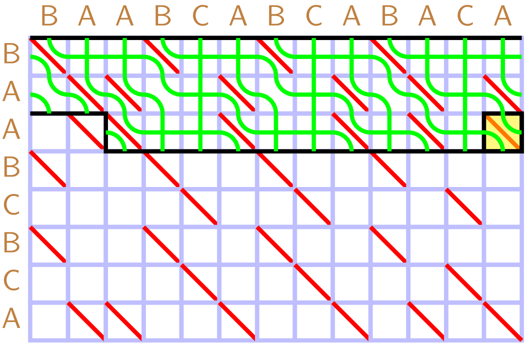
Periodic string-substring LCS by wraparound combing



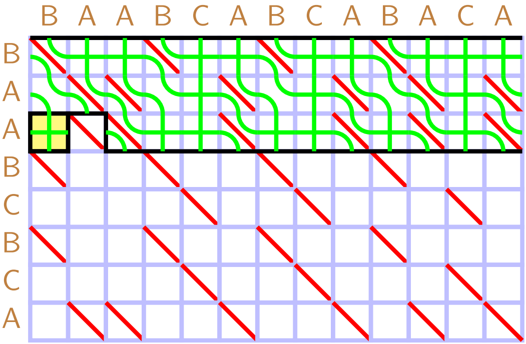
Periodic string-substring LCS by wraparound combing



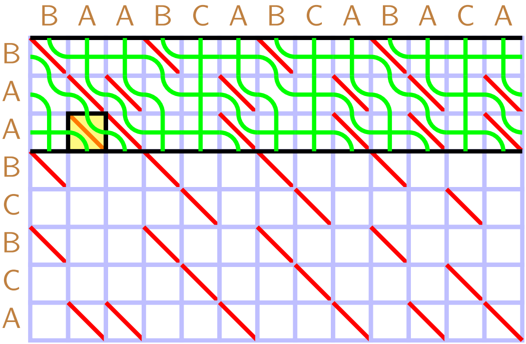
Periodic string-substring LCS by wraparound combing



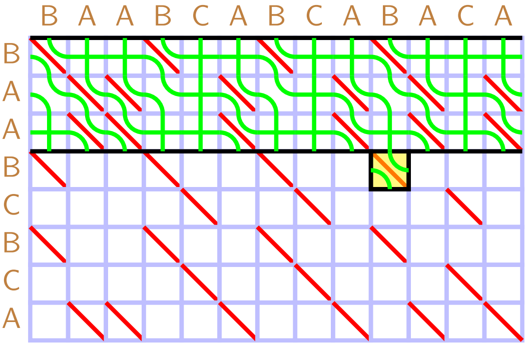
Periodic string-substring LCS by wraparound combing



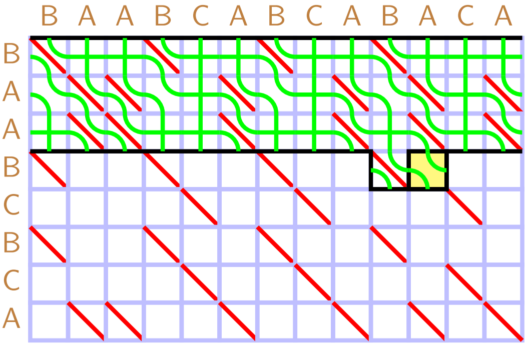
Periodic string-substring LCS by wraparound combing



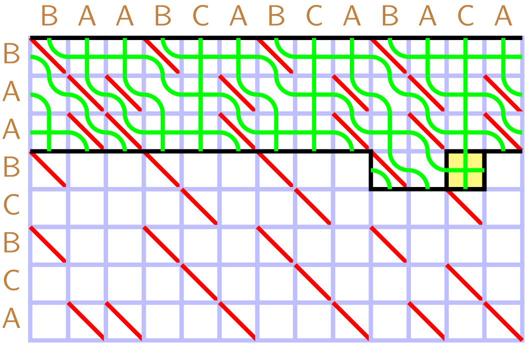
Periodic string-substring LCS by wraparound combing



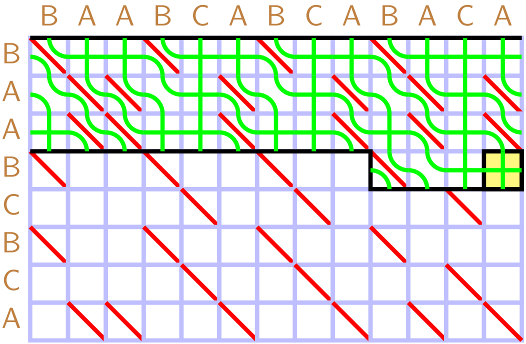
Periodic string-substring LCS by wraparound combing



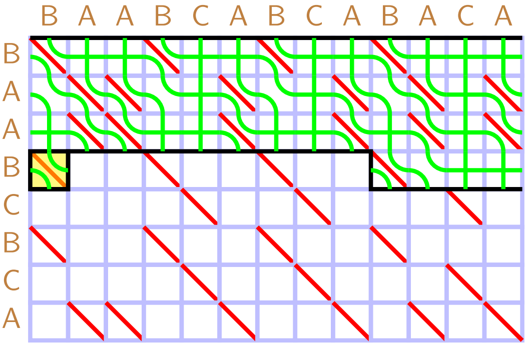
Periodic string-substring LCS by wraparound combing



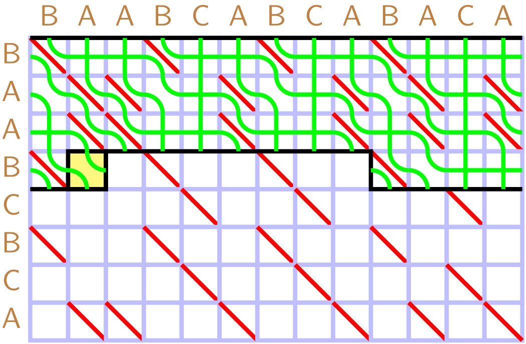
Periodic string-substring LCS by wraparound combing



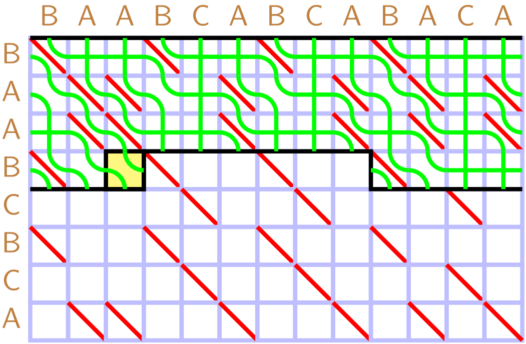
Periodic string-substring LCS by wraparound combing



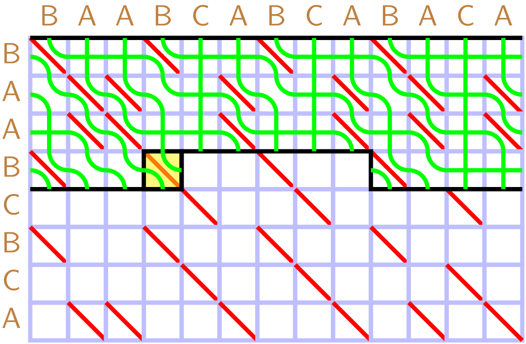
Periodic string-substring LCS by wraparound combing



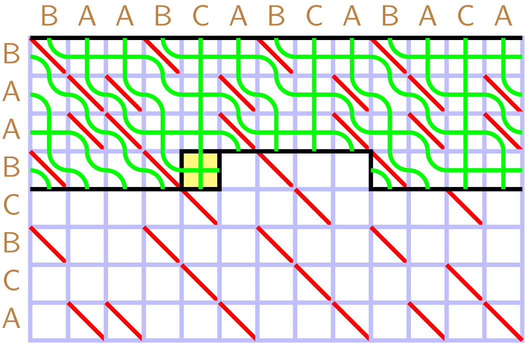
Periodic string-substring LCS by wraparound combing



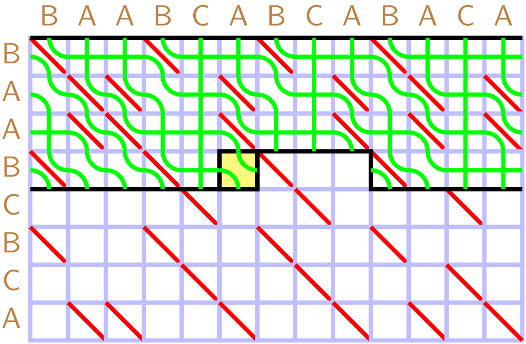
Periodic string-substring LCS by wraparound combing



Periodic string-substring LCS by wraparound combing

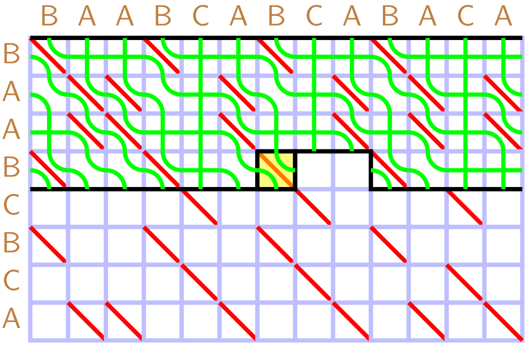


Periodic string-substring LCS by wraparound combing

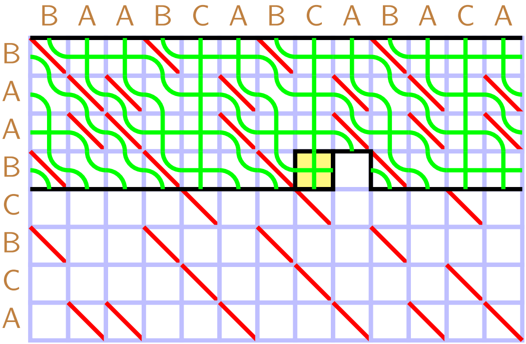


Further string comparison
Periodic LCS

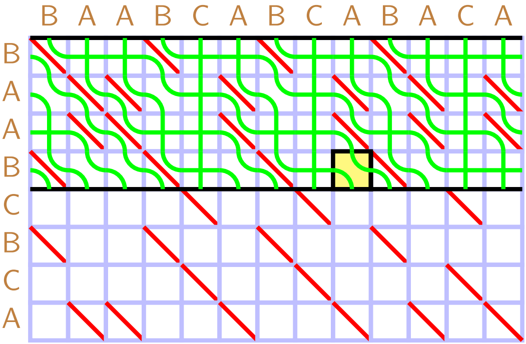
Periodic string-substring LCS by wraparound combing



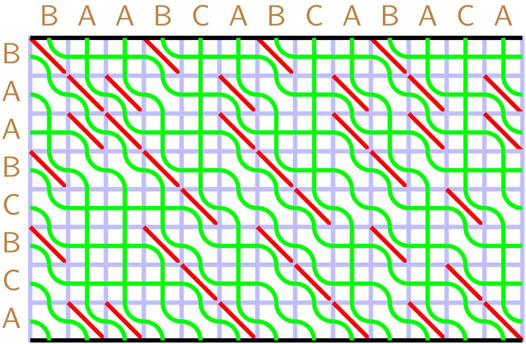
Periodic string-substring LCS by wraparound combing



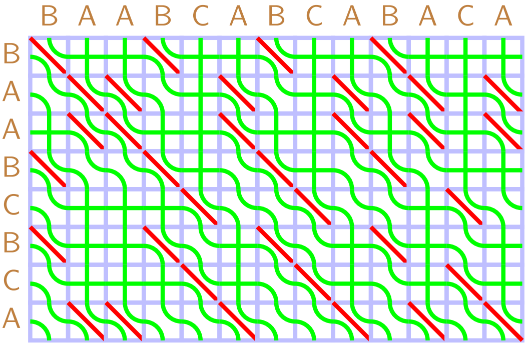
Periodic string-substring LCS by wraparound combing



Periodic string-substring LCS by wraparound combing



Periodic string-substring LCS by wraparound combing



Forward preiodic braid: kernel for every prefix of *a* vs (infinite) *b*

Implicit prefix-substring LCS

Periodic string-substring LCS: Wraparound combing

Initialise uncombed sticky braid: mismatch cell = crossing

Iterate over cells in rows: row begins at match, wraps around at boundary

- match cell: skip (keep uncrossed)
- mismatch cell: comb (uncross if strands crossed before, possibly in a different period)

Active cell update: time $O(1)$

String-substring LCS score: count strands (with multiplicities, whenever substring covers multiple periods)

Overall time $O(mn)$

Further string comparison
Periodic LCS

u: string of length p v: string of length q

Doubly-periodic string-substring LCS problem

LCS scores: $a = uu \dots u = u^k$ vs every substring of $b = \dots vvv \dots = v^\infty$

Output scores can be represented implicitly

May assume that every character of u occurs in v (otherwise delete it)

Only need substrings in b of length $\leq kpq$ (otherwise LCS score $= kp$)

Doubly-periodic string-substring LCS: running time

$O(k^2pq)$	naive
$O(kpq)$	as single-periodic
$O(pq + \log k \cdot q \log q)$	[GT: 2023]

Doubly-periodic string-substring LCS

Single-periodic string-substring LCS for u vs $b = v^\infty$: periodic LCS kernel

Concatenate periods of $a = u^k$ recursively: $\log k$ steps

Each step: $\boxdot$-multiplication of kernels

Sufficient to perform $\boxdot$-multiplication on finite subkernels of three (!) consecutive periods: then result correct on whole b

Overall time $O(pq + \log k \cdot q \log q)$

Further string comparison
Periodic LCS

Doubly-periodic LCS: set as Problem L in Petrozavodsk Programming Camp 2023

ICPC (International Collegiate Programming Contest): top annual event for competitive programming

Petrozavodsk Programming Camp: long-standing conference/experimental ground for ICPC problem setters

- high geographical coverage
- stands out by problems' complexity and originality
- used to propose/test new ideas and approaches

Further string comparison
Periodic LCS

Initial model solution: standard DP with optimisations

$O(\max(p,q)^3 \log(pk+ql))$; solving up to $p \leq 50$ (time limit 20 s)

Refined in succesive versions using sticky braids

Version with cubic $\boxdot$-mult via generic $\odot$-mult

$O(pq + q^3 \log k)$; solving $p \leq 50$ within 1 s; $p = 500$ out of time

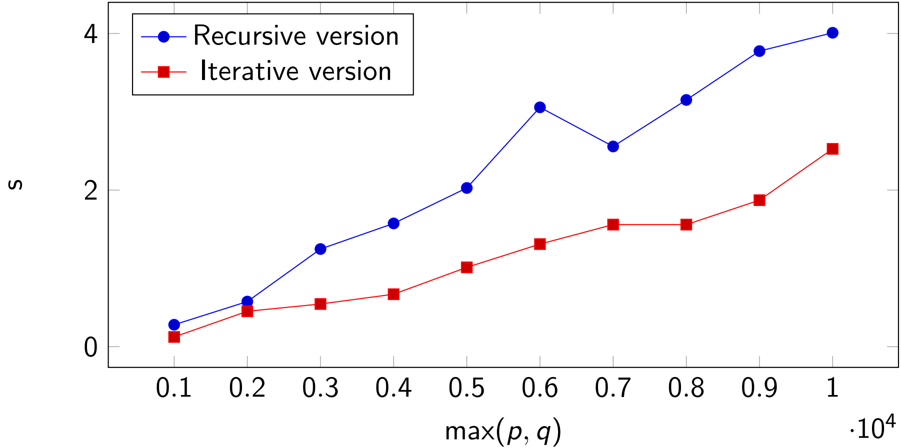
Version with quadratic $\boxdot$-mult via Monge $\odot$-multiplication (Knuth)

$O(pq + q^2 \log k)$; solving $p \leq 500$ within 10 s; $p = 1000$ out of time

Set as challenge benchmark for contestants

Version with quasilinear $\boxdot$-mult (Steady Ant): $O(pq + q \log q \log k)$

Version with recursion-free Steady Ant: another speedup by approx $\times 2$



Conclusions

Deep connection between classical combinatorial algorithms (LCS) and algebra (the Hecke monoid)

Powerful due to fast $\boxdot$-multiplication (Steady Ant)

New application: doubly-periodic LCS

Periodic structures are important, occur frequently e.g. in bioinformatics

New highly engineered implementation

Tested in a competitive ICPC-style environment

Further work:

- general-purpose library for semi-local LCS
- further applications
- popularising the Hecke monoid in the competitive programming community

Conclusions