CPM 2018

Computing longest common square subsequences

Takafumi Inoue[1], Shunsuke Inenaga[1],
Heikki Hyyrö[2], Hideo Bannai[1], Masayuki Takeda[1]

[1]Kyushu University
[2]University of Tampere

Longest Common Subsequence (LCS)

LCS Problem

Input: two strings A and B of length n each

Output: (length of) LCS of A and B

- LCS is a classical measure for string comparison.
- Standard DP solves this in $O(n^2)$ time.

E.g.) $A = \mathtt{aacaabad}$

vs

$B = \mathtt{cacbcbbd}$

Longest Common Subsequence (LCS)

LCS Problem

Input: two strings A and B of length n each

Output: (length of) LCS of A and B

- LCS is a classical measure for string comparison.
- Standard DP solves this in $O(n^2)$ time.

E.g.) $A = \texttt{\textbf{a}a\textbf{c}aa\textbf{b}a\textbf{d}}$

vs

$B = \texttt{c\textbf{ac}bcb\textbf{bd}}$

Constrained/Restricted LCS

- Variants of LCS problem where the solution must satisfy pre-determined constraints.
- Attempt to reflect user's a-priori knowledge to the solutions.
  - STR-IC-LCS, STR-EC-LCS, SEQ-IC-LCS, SEQ-EC-LCS
    LCS of A and B that includes (excludes) given pattern P as a substring (subsequence).
    (See [Kuboi et al, CPM 2017] and references therein)
  - Longest common *palindromic* subsequence (LCPS)
    [Chowdhury et al. 2014, Inenaga & Hyyrö 2018, Bae & Lee 2018]

Longest Common Square Subseq. (LCSS)

- This work considers new variant of LCS, called LCSS, where the solution has to be *square*.
- Square (a.k.a. tandem repeat) is string of form xx.
  - `aabaab`
  - `abababab`
  - `abcbbabcbb`

Longest Common Square Subseq. (LCSS)

LCSS Problem

Input: two strings A and B of length n each

Output: (length of) LCSS of A and B

E.g.)

$A = \texttt{monsterstrike}$

vs

$B = \texttt{fourstringmasters}$

Longest Common Square Subseq. (LCSS)

LCSS Problem

Input: two strings A and B of length n each

Output: (length of) LCSS of A and B

E.g.)

$A =$ `mon`**`sterstr`**`ike`

vs

$B =$ `four`**`str`**`ingma`**`ster`**`s`

Our Results

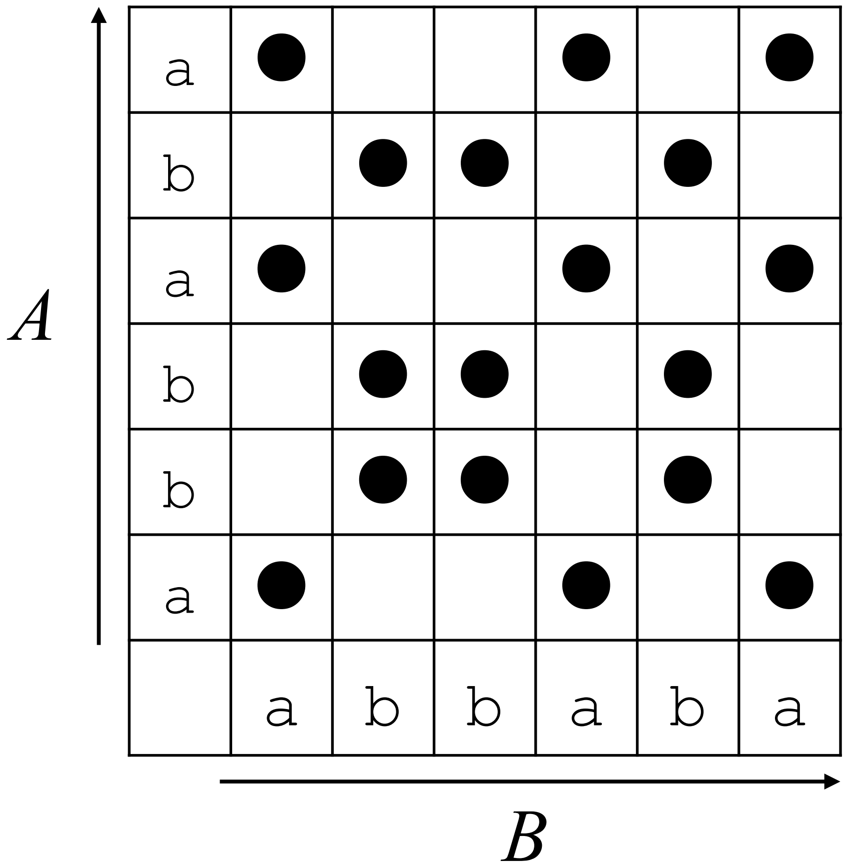
Upper bounds (algorithms) for LCSS

algorithm	time	space
Naïve	$O(n^6)$	$O(n^4)$
Simple	$O(Mn^4)$	$O(n^4)$
Matching rectangle 1	$O(\sigma M^3+n)$	$O(M^2+n)$
Matching rectangle 2	$O(M^3 \log^2 n \log\log n + n)$	$O(M^3+n)$

- n is the length of the input strings.
- M is the number of matching points,
  i.e., $M = |\{(i, j) \mid A[i] = B[j], 1 \le i, j \le n\}|$.
- σ is the alphabet size.

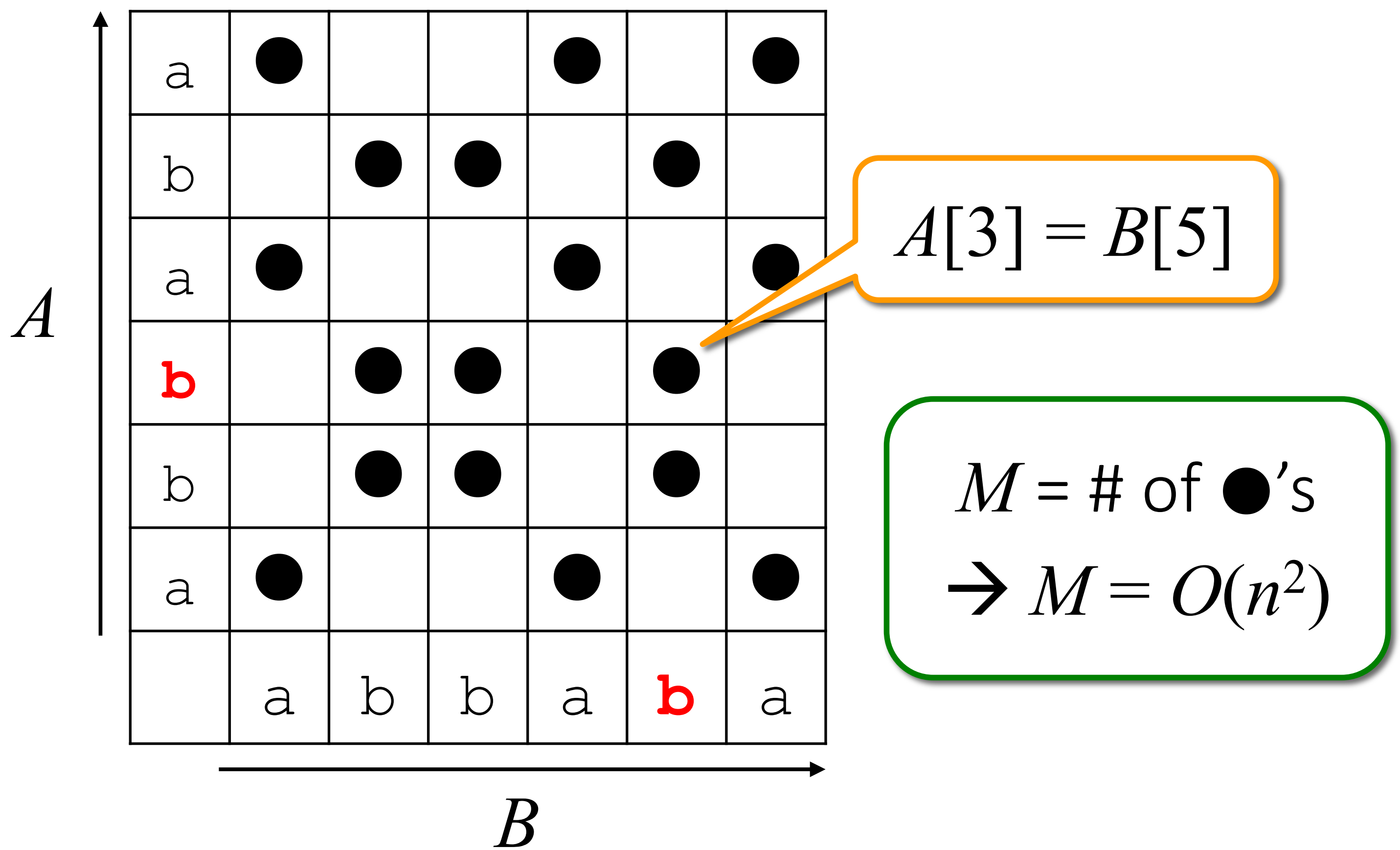
Matching Points

- M is the number of matching points, i.e., $M = |\{(i, j) \mid A[i] = B[j],\ 1 \le i, j \le n\}|$.



Matching Points

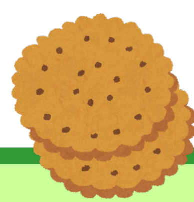
- M is the number of matching points, i.e., $M = |\{(i, j) \mid A[i] = B[j], 1 \leq i, j \leq n\}|$.



Matching Points [Cont.]

- But M can be much smaller than $O(n^2)$ in many cases

e							
i		●				●	
k							
o							
o							
c				●			
	b	i	s	c	u	i	t



Our Results

Upper bounds (algorithms) for LCSS

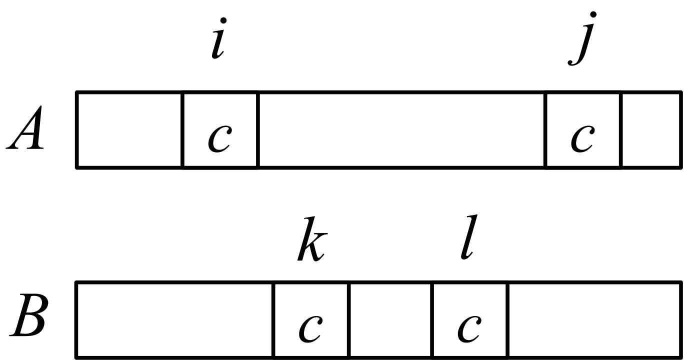
algorithm	time	space
Naïve	$O(n^6)$	$O(n^4)$
Simple	$O(Mn^4)$	$O(n^4)$
Matching rectangle 1	$O(\sigma M^3+n)$	$O(M^2+n)$
Matching rectangle 2	$O(M^3 \log^2 n \log\log n + n)$	$O(M^3+n)$

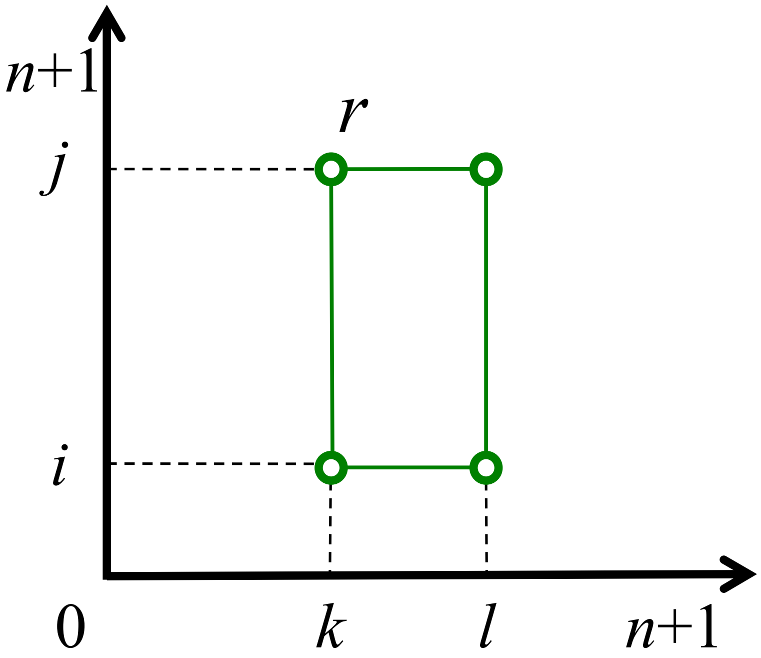
M is at most $O(n^2)$ and can be much smaller

- n is the length of the input strings.
- M is the number of matching points, i.e., $M = |\{(i, j) \mid A[i] = B[j], 1 \leq i, j \leq n\}|$.
- σ is the alphabet size.

Matching Rectangles

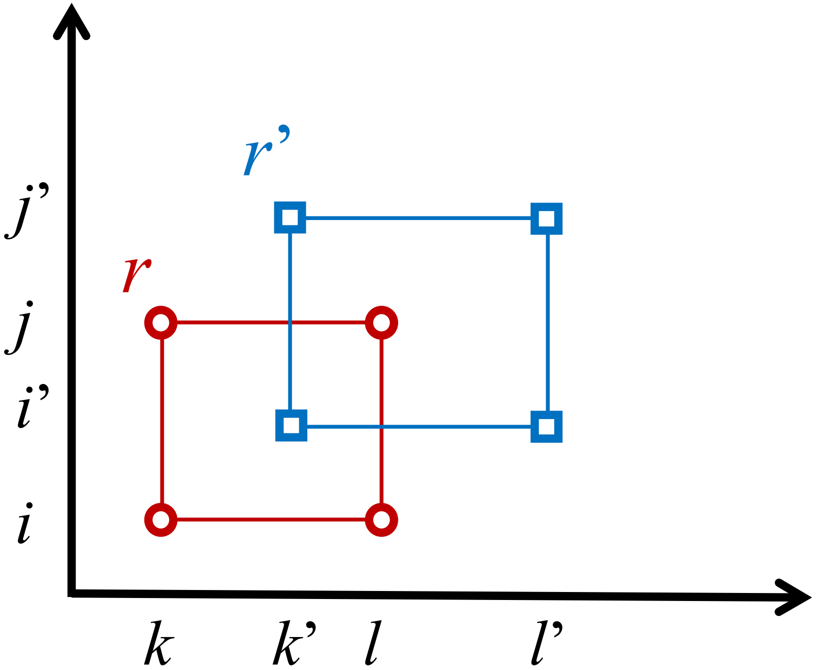
- Tuple $r = (i, j, k, l)$ is called *matching rectangle* if $A[i] = A[j] = B[k] = B[l]$.

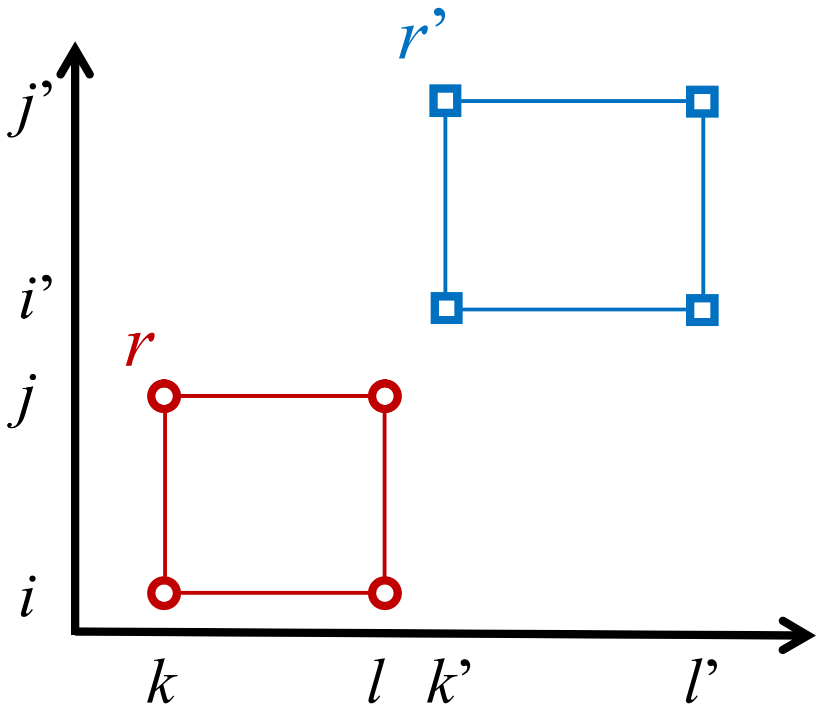




Partial Order of Matching Rectangles

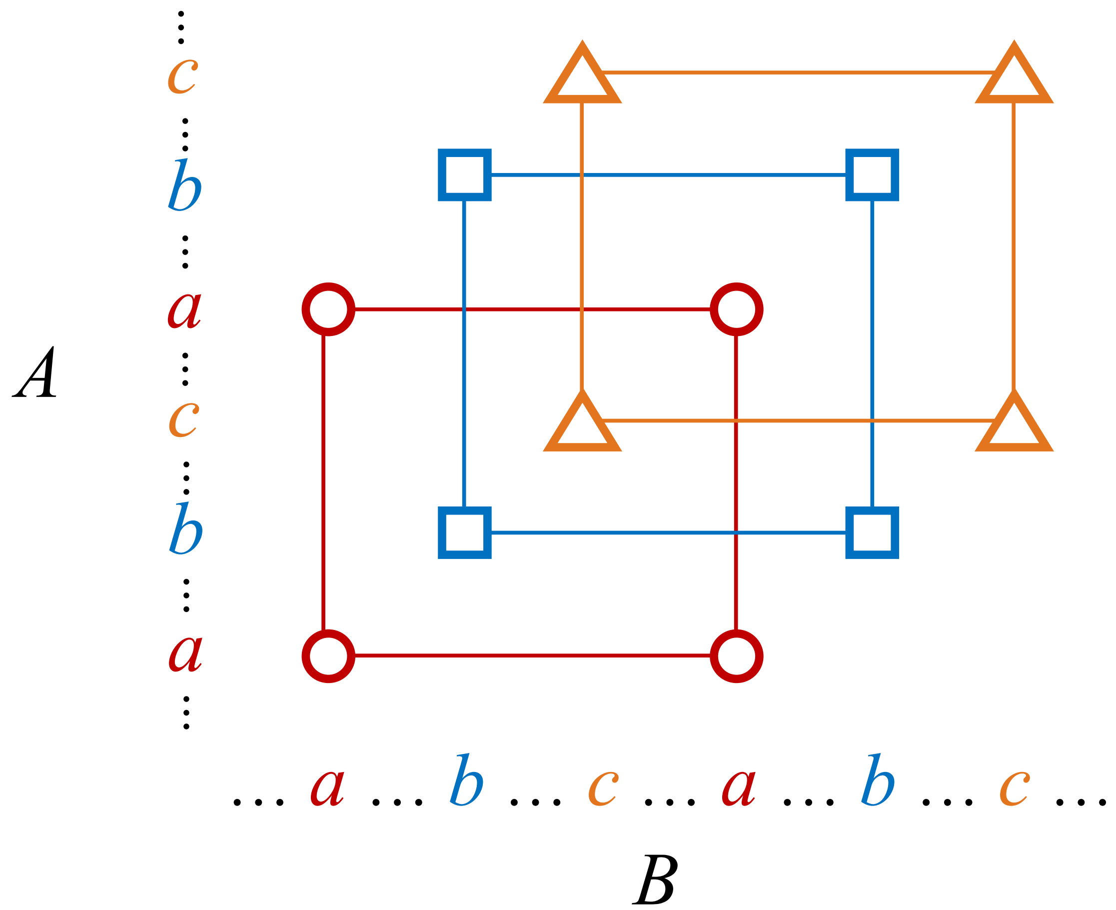
- For matching rectangles $r = (i, j, k, l)$ and $r' = (i', j', k', l')$, $r < r'$ iff $i < i'$, $j < j'$, $k < k'$, and $l < l'$.
  Namely, $r < r'$ iff r lies strictly more left-lower than r'.





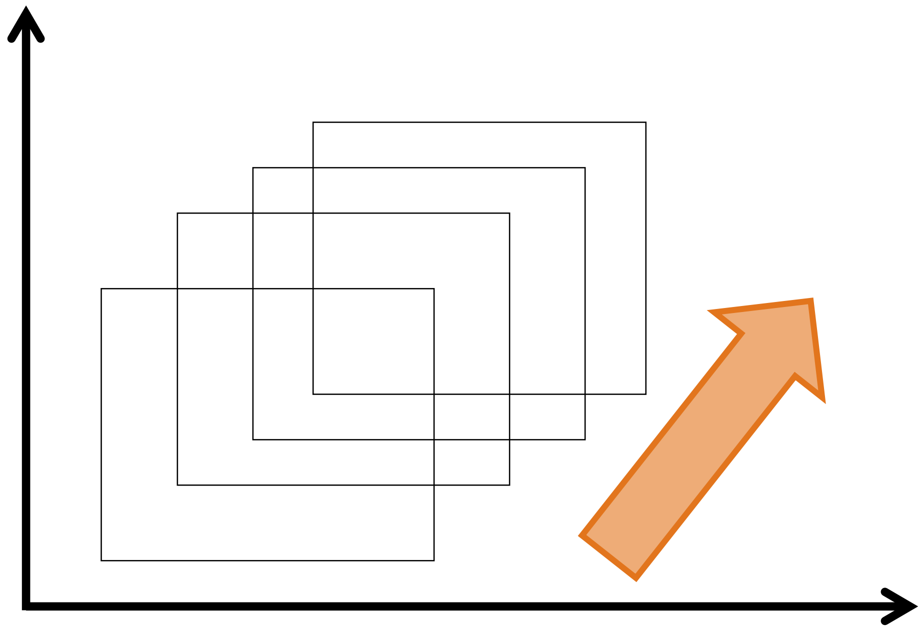
Observation

- Each common square subsequence has corresponding sequence of matching rectangles.



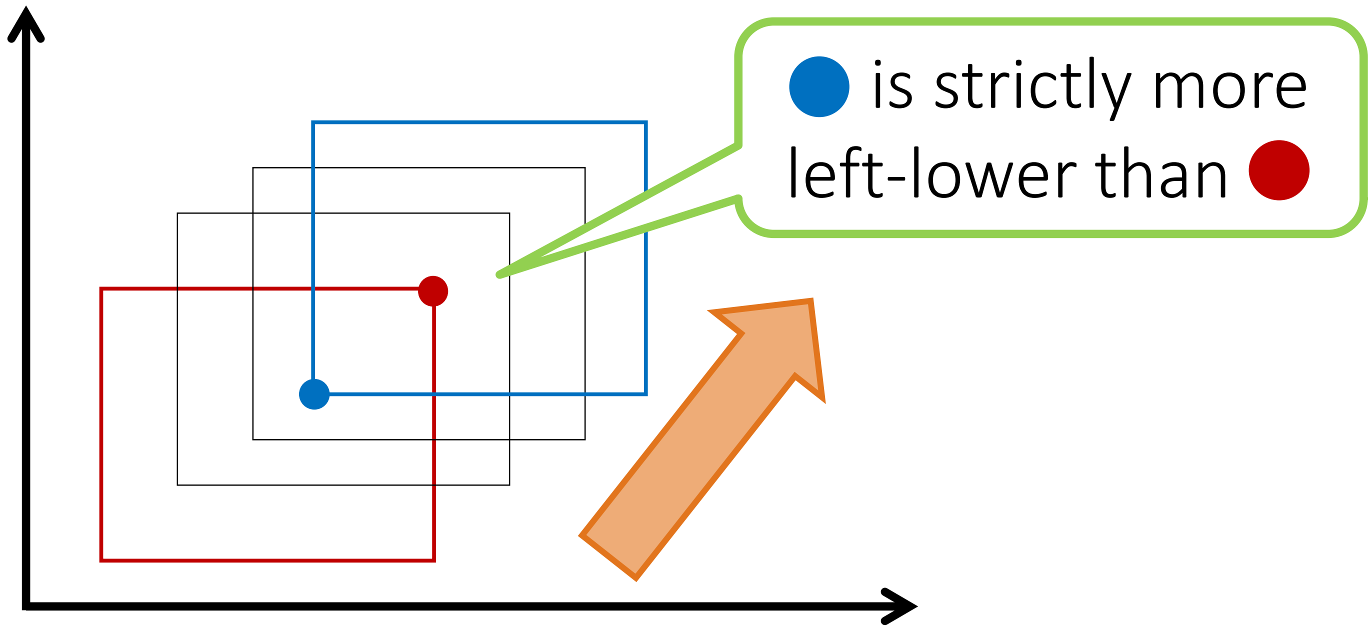
CSS and matching rectangle

- Sequence $r_1, \ldots, r_s$ of s matching rectangles represents CSS of length s iff
  - $\boxed{r_1 < r_2 \ldots < r_s}$
  - $\boxed{i_s < j_1, k_s < l_1}$ where $r_1 = (i_1, j_1, k_1, l_1)$, $r_s = (i_s, j_s, k_s, l_s)$



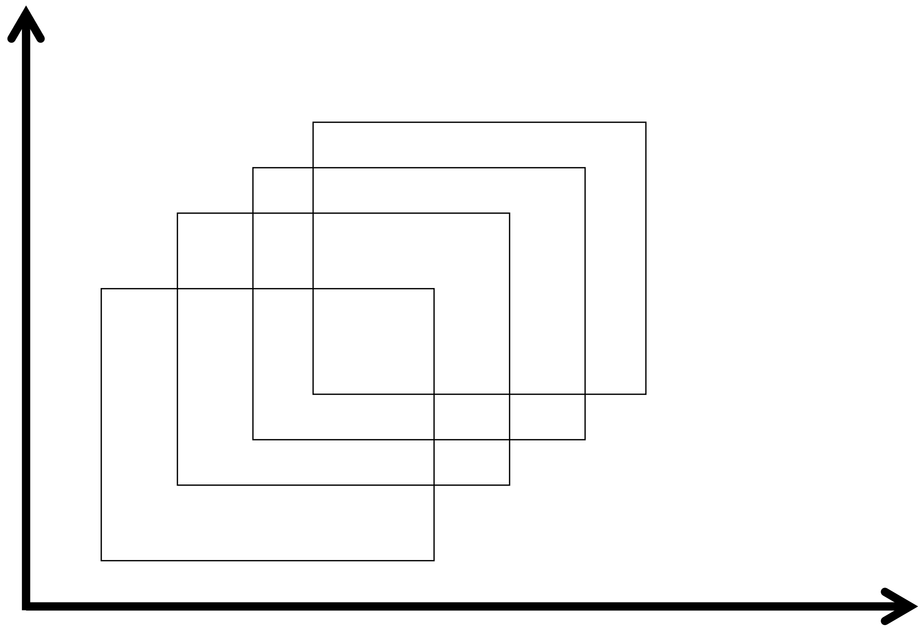
CSS and matching rectangle

- Sequence $r_1, \ldots, r_s$ of s matching rectangles represents CSS of length s iff
  - $\boxed{r_1 < r_2 \ldots < r_s}$
  - $\boxed{i_s < j_1, k_s < l_1}$ where $r_1 = (i_1, j_1, k_1, l_1)$, $r_s = (i_s, j_s, k_s, l_s)$



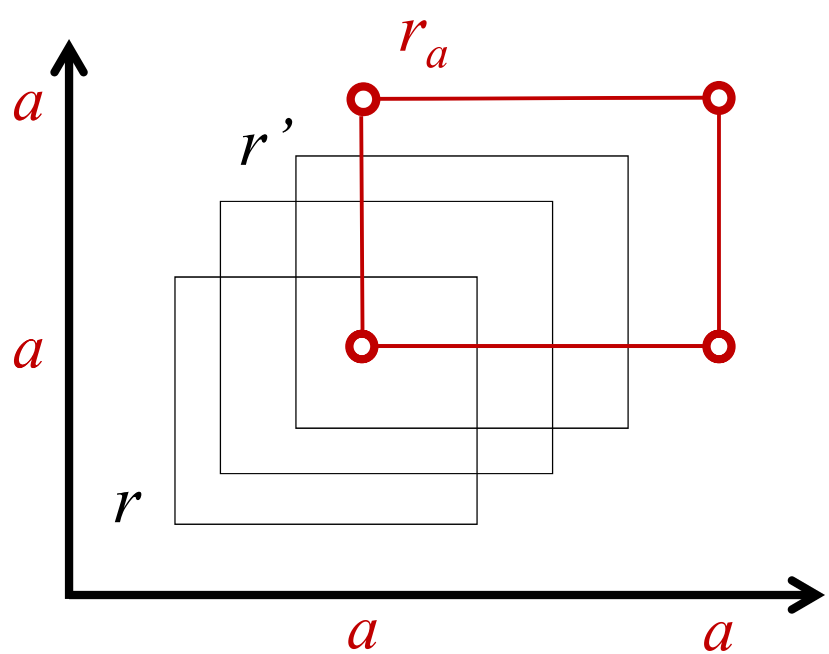
LCSS → Longest sequence of DOMRs

- Computing LCSS reduces to finding longest sequence of diagonally overlapping matching rectangles (DOMRs).



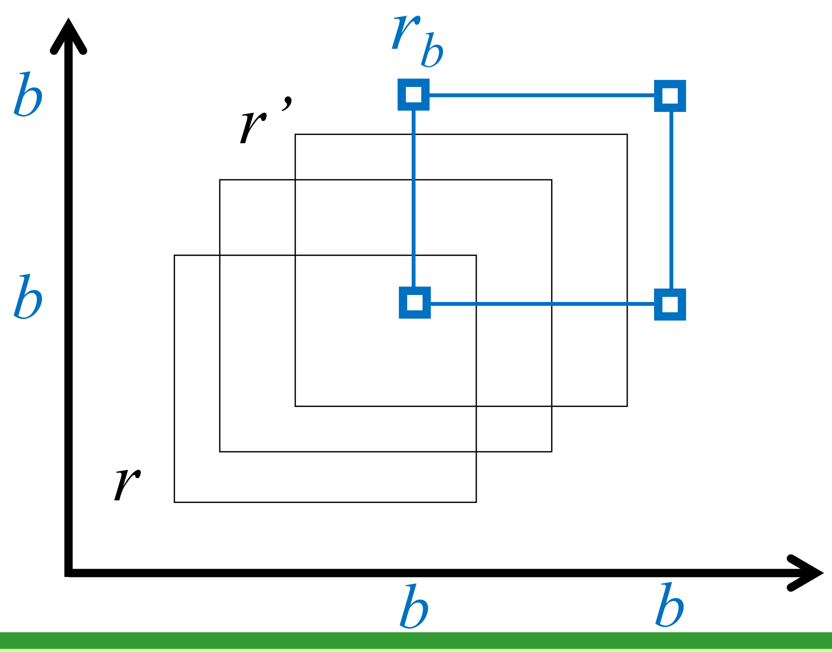
Basic Algorithm

- For each matching rectangle r, maintain DP table D_r of size M^2 such that $\mathrm{D}_r[r']$ stores length of longest sequence of DOMRs that begins with r and ends with r'.
- For each character c, find the "closest" matching rectangle r_c w.r.t. c that can be added after r'. Update $\mathrm{D}_r[r_c]$ if needed.



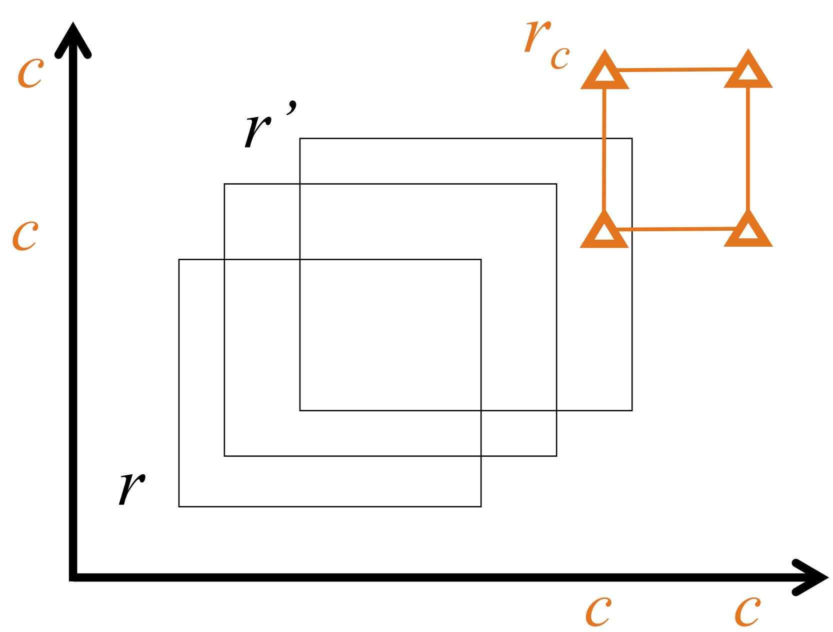
Basic Algorithm

- For each matching rectangle r, maintain DP table D_r of size M^2 such that $\mathrm{D}_r[r']$ stores length of longest sequence of DOMRs that begins with r and ends with r'.
- For each character c, find the "closest" matching rectangle r_c w.r.t. c that can be added after r'. Update $\mathrm{D}_r[r_c]$ if needed.



Basic Algorithm

- For each matching rectangle r, maintain DP table D_r of size M^2 such that $\mathrm{D}_r[r']$ stores length of longest sequence of DOMRs that begins with r and ends with r'.
- For each character c, find the "closest" matching rectangle r_c w.r.t. c that can be added after r'. Update $\mathrm{D}_r[r_c]$ if needed.



Basic Algorithm [Cont.]

- Let R be # of matching rectangles ($R = O(M^2)$).
- We compute $\mathrm{D}_r[r']$ for $R^2 = O(M^4)$ pairs of matching rectangles (r, r') .
- We test σ characters to extend the current sequence of DOMRs w.r.t. $\mathrm{D}_r[r']$.
- Each extension can be obtained in $O(1)$ time after suitable preprocessing.

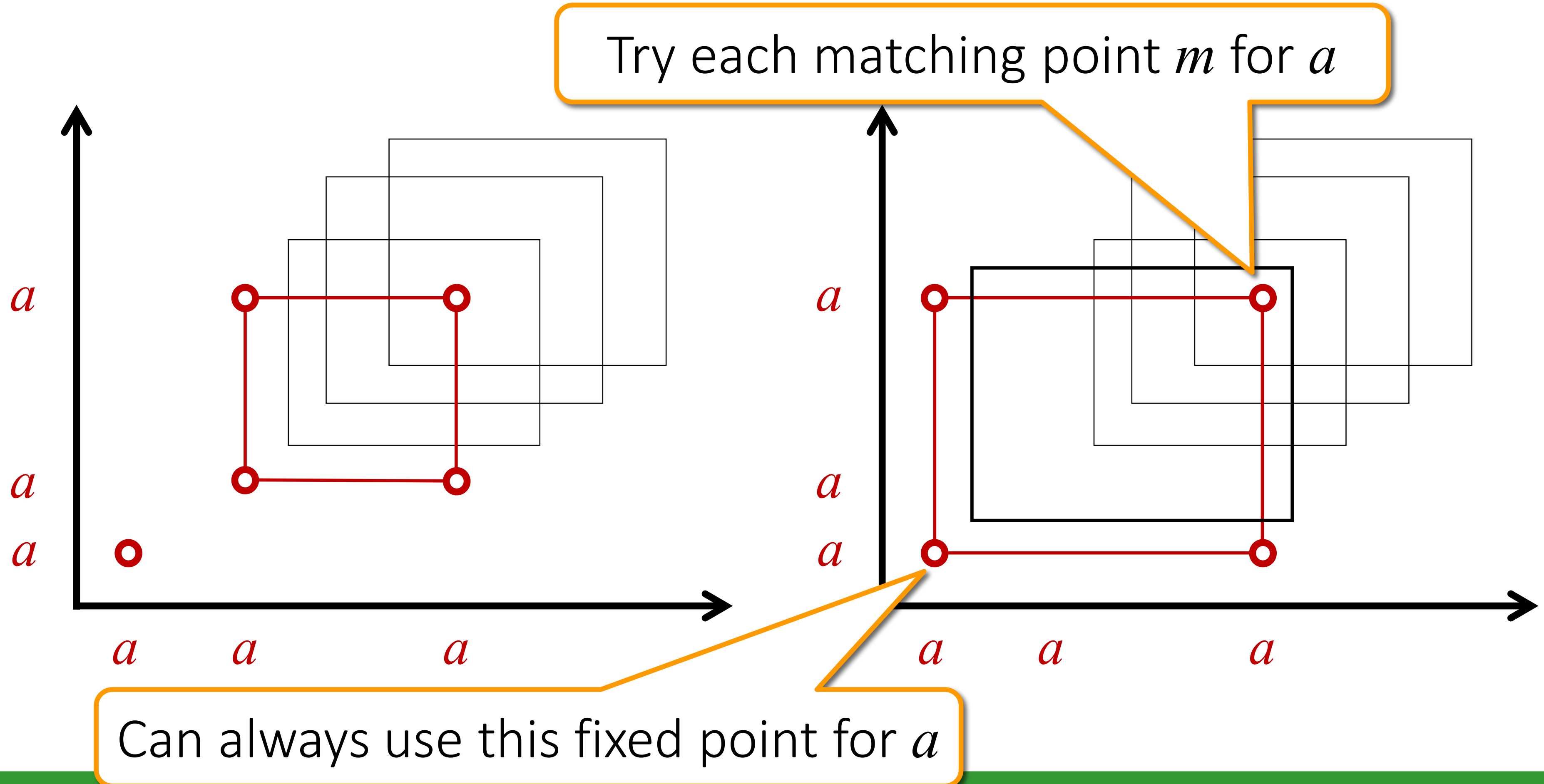
→ $O(\sigma R^2 + n) = O(\sigma M^4 + n)$ time... Slow?

⇩ Can be improved to

$O(\sigma MR + n) = O(\sigma M^3 + n)$ time

On Start Matching Rectangle

- Always better to use a start matching rectangle that has the "smallest" left-lower corner for each character.



Improved Algorithm

- We compute $\mathrm{D}_m[r']$ for $MR = O(M^3)$ pairs of matching points and matching rectangles (m, r') .
- We test σ characters to extend the current sequence of DOMRs.
- Each extension can be obtained in $O(1)$ time after suitable preprocessing.

→ $O(\sigma MR + n) = O(\sigma M^3 + n)$ time!

Improved Algorithm [Cont.]

Theorem

The LCSS problem can be solved in $O(\sigma MR + n) = O(\sigma M^3 + n)$ time with $O(M^2 + n)$ space.

Corollary

The *expected* running time of this algorithm is $O(n^6/\sigma^3)$.

- For random text $M \approx n^2/\sigma$ and $R \approx M^2/\sigma \approx n^4/\sigma^3$.

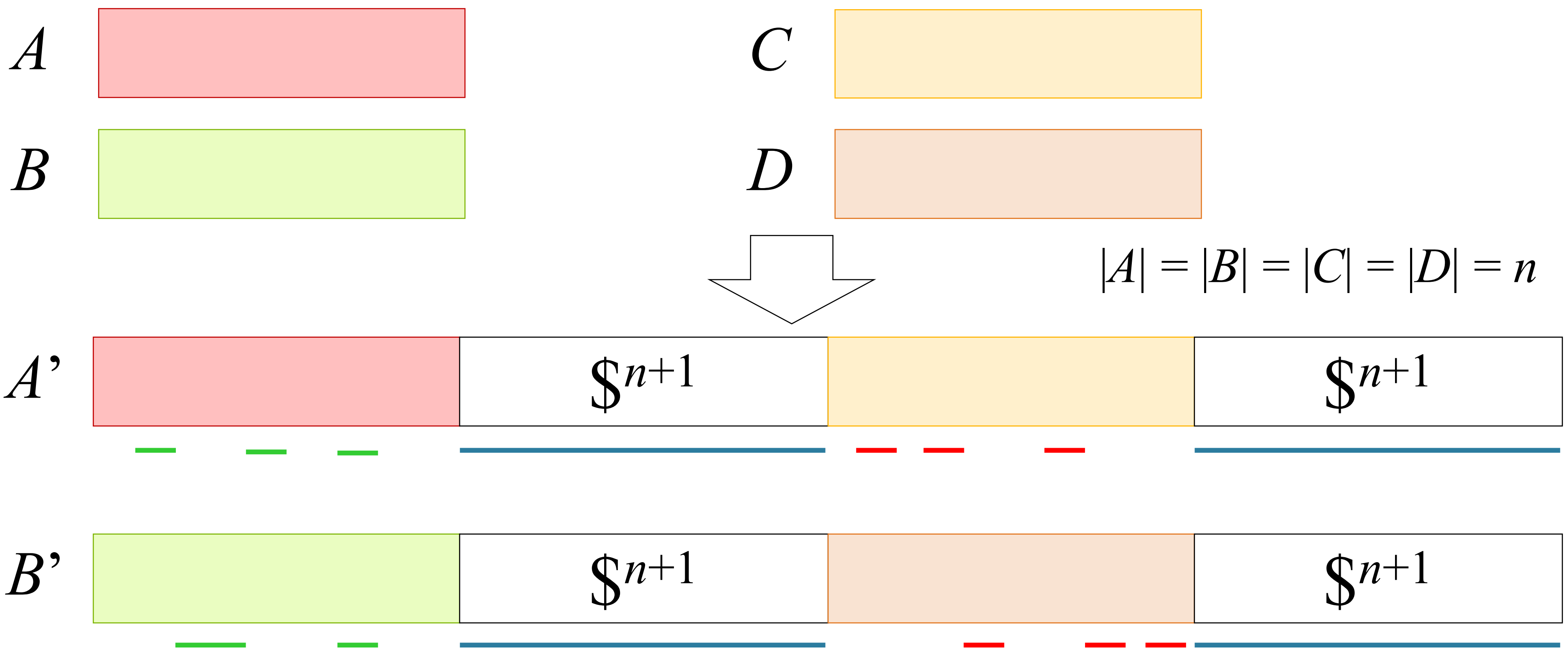
Hardness of LCSS

Lemma

LCSS for two strings is at least as hard as LCS for four strings.

4-LCS → 2-LCSS

Computing LCS for A, B, C, D of length n each reduces to computing LCSS of A', B' of length $4n+2$ each.



Conditional Lower Bound for LCSS

Lemma [Abboud et al. 2015]

There is no algorithm which solves the LCS problem for k strings in $O(n^{k-\varepsilon})$ time with constant $\varepsilon > 0$, unless the strong exponential time hypothesis (SETH) fails.

Corollary

There is no algorithm which solves the LCSS problem for two strings in $O(n^{4-\varepsilon})$ time with constant $\varepsilon > 0$, unless SETH fails.

Conclusions & Open Problem

$M = O(n^2)$

Upper bounds for LCSS

algorithm	time	space
Naïve	$O(n^6)$	$O(n^4)$
Simple	$O(Mn^4)$	$O(n^4)$
Matching rectangle 1	$O(\sigma M^3+n)$	$O(M^2+n)$
Matching rectangle 2	$O(M^3 \log^2 n \log\log n + n)$	$O(M^3+n)$

Conditional Lower bound for LCSS

$O(n^{4-\varepsilon})$-time solution (with constant $\varepsilon > 0$) is unlikely to exist

How can we close this (almost) quadratic gap?

Strong Exponential Time Hypothesis (SETH)

- Let s_k be the greatest lower bound (infimum) of real numbers δ such that k-SAT can be solved in $O(2^{\delta n})$ time, where n = # of variables.
- The *exponential time hypothesis* (*ETH*) is a conjecture that $s_k > 0$ for any $k \geq 3$.
- Clearly $s_3 \leq s_4 \leq s_5$ …
  The *strong ETH* (*SETH*) is a conjecture that the limit of s_k when k approaches ∞ is 1.