Faster Algorithms for Computing Longest Common Increasing Subsequences

Gerth Stølting Brodal

Irit Katriel

BRICS, University of Aarhus
Århus, Denmark

Kanela Kaligosi

Martin Kutz

Max-Planck Institut für Informatik
Saarbrücken, Germany

The Longest-Commmon-Subsequence Problem

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some alphabet Σ

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

The Longest-Commmon-Subsequence Problem

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some alphabet Σ

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a longest subsequence that occurs in both sequences, a *longest common subsequence (LCS)*

The Longest-Commmon-Subsequence Problem

Given: two sequences $A = (a_1, \dots, a_m)$, $B = (b_1, \dots, b_n)$ over some alphabet Σ

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a longest subsequence that occurs in both sequences, a *longest common subsequence (LCS)*

The Longest-Commmon-Subsequence Problem

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some alphabet Σ

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a longest subsequence that occurs in both sequences, a *longest common subsequence (LCS)*

The Longest-Commmon-Subsequence Problem

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some alphabet Σ

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a longest subsequence that occurs in both sequences, a *longest common subsequence (LCS)*

Note: letters may occur repeatedly in the subsequence

The Longest-Increasing-Subsequence Problem

Given: a sequence $A = (a_1, \ldots, a_m)$
over an ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \epsilon\}$

$$\gamma \quad \alpha \quad \delta \quad \beta \quad \delta \quad \alpha \quad \gamma \quad \beta \quad \epsilon \quad \delta$$

The Longest-Increasing-Subsequence Problem

Given: a sequence $A = (a_1, \ldots, a_m)$
over an ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \epsilon\}$

$$\gamma \quad \alpha \quad \delta \quad \beta \quad \delta \quad \alpha \quad \gamma \quad \beta \quad \epsilon \quad \delta$$

Task: Find a *longest increasing subsequence (LIS)* in A

The Longest-Increasing-Subsequence Problem

Given: a sequence $A = (a_1, \ldots, a_m)$
over an ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \epsilon\}$

$$\gamma \quad \alpha \quad \delta \quad \beta \quad \delta \quad \alpha \quad \gamma \quad \beta \quad \epsilon \quad \delta$$

Task: Find a *longest increasing subsequence (LIS)* in A

The Longest-Increasing-Subsequence Problem

Given: a sequence $A = (a_1, \dots, a_m)$
over an ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \epsilon\}$

$$\gamma \quad \alpha \quad \delta \quad \beta \quad \delta \quad \alpha \quad \gamma \quad \beta \quad \epsilon \quad \delta$$

Task: Find a *longest increasing subsequence (LIS)* in A

Important: here, letters may not occur repeatedly
(strictly increasing subsequence)

Classical Results

- LCS can be computed in $O(mn)$ time by dynamic programming [Wagner & Fischer, 1974]

  (and by divide-&-conquer in $O(n)$ space [Hirschberg, 1975])

Classical Results

- LCS can be computed in $O(mn)$ time by dynamic programming [Wagner & Fischer, 1974]

  (and by divide-&-conquer in $O(n)$ space [Hirschberg, 1975])

- $\Theta(\log n)$-time speed-up possible [Masek & Paterson, 1980]

Classical Results

- LCS can be computed in $O(mn)$ time by dynamic programming [Wagner & Fischer, 1974]

  (and by divide-&-conquer in $O(n)$ space [Hirschberg, 1975])

- $\Theta(\log n)$-time speed-up possible [Masek & Paterson, 1980]

- important parameter: $r = \#$matches (pairs (i,j) with $a_i = b_j$)

  LCS in $O(r \log n)$ time [Hunt & Szymanski, 1977]
  (assuming $r \geq m, n$)

Classical Results

- LCS can be computed in $O(mn)$ time by dynamic programming [Wagner & Fischer, 1974]

  (and by divide-&-conquer in $O(n)$ space [Hirschberg, 1975])

- $\Theta(\log n)$-time speed-up possible [Masek & Paterson, 1980]

- important parameter: $r = \#$matches (pairs (i, j) with $a_i = b_j$)

  LCS in $O(r \log n)$ time [Hunt & Szymanski, 1977]
  (assuming $r \geq m, n$)

- LIS in $O(n \log n)$ time [Fredman, 1975]

  (also as corollary of $O(r \log n)$-time algorithm above)

Longest Commmon Increasing Subsequences

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$
over some ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \cdots\}$

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Longest Commmon Increasing Subsequences

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \cdots\}$

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a *longest increasing* subsequence that occurs in both sequences, a *longest common increasing subsequence (LCIS)*

Longest Commmon Increasing Subsequences

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \cdots\}$

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a *longest increasing* subsequence that occurs in both sequences, a *longest common increasing subsequence (LCIS)*

Longest Commmon Increasing Subsequences

Given: two sequences $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ over some ordered alphabet $\Sigma = \{\alpha < \beta < \gamma < \delta < \cdots\}$

$$\alpha \quad \gamma \quad \gamma \quad \beta \quad \epsilon \quad \alpha \quad \beta \quad \beta \quad \alpha \quad \gamma \quad \delta$$

$$\gamma \quad \beta \quad \alpha \quad \gamma \quad \epsilon \quad \beta \quad \delta \quad \delta \quad \alpha \quad \beta \quad \delta$$

Task: Find a *longest increasing* subsequence that occurs in both sequences, a *longest common increasing subsequence (LCIS)*

Quite recently introduced by Yang, Huang, and Chao (IPL, 2005): They compute LCIS in $\Theta(mn)$ time and space.

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]
- parametrized: $O\left(\min\{r\log|\Sigma|, m|\Sigma|+r\}\log\log m + \mathit{Sort}_\Sigma(m)\right)$
  [Chan et al., ISAAC 2005]

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]
- parametrized: $O\left(\min\{r \log|\Sigma|, m|\Sigma| + r\} \log\log m + \mathit{Sort}_\Sigma(m)\right)$
  (essentially $O(r \cdot \log|\Sigma|)$) [Chan et al., ISAAC 2005]

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]
- parametrized: $O\left(\min\{r \log|\Sigma|, m|\Sigma| + r\} \log\log m + Sort_{\Sigma}(m) \right)$

  (essentially $O(r \cdot \log|\Sigma|)$) [Chan et al., ISAAC 2005]

  remember: r might be $\Omega(mn)$
  but it could also be much smaller in certain important cases
  (when A, B are permutations, for example)

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]
- parametrized: $O\left(\min\{r \log|\Sigma|, m|\Sigma| + r\} \log\log m + \mathit{Sort}_\Sigma(m)\right)$

  (essentially $O(r \cdot \log|\Sigma|)$) [Chan et al., ISAAC 2005]

  remember: r might be $\Omega(mn)$
  but it could also be much smaller in certain important cases
  (when A, B are permutations, for example)

New Result:

An LCIS for a length-m and a length-n sequence can be computed in $O\left((m + n\ell) \log\log|\Sigma| + \mathit{Sort}_\Sigma(m)\right)$ time, where $\ell =$ length of LCIS.

LCIS Results

- LCIS in $\Theta(mn)$ time and space [Yang et al., IPL 2005]
- parametrized: $O\left(\min\{r \log|\Sigma|, m|\Sigma| + r\} \log\log m + \mathrm{Sort}_\Sigma(m)\right)$

  (essentially $O(r \cdot \log|\Sigma|)$) [Chan et al., ISAAC 2005]

  remember: r might be $\Omega(mn)$
  but it could also be much smaller in certain important cases
  (when A, B are permutations, for example)

New Result:

An LCIS for a length-m and a length-n sequence can be computed in $O\left((m + n\ell) \log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\right)$ time, where $\ell =$ length of LCIS.

(essentially $O(n\ell)$) $(n \geq m)$

We *"usually"* expect quite small ℓ. So it's a *"good"* parameter!

LCIS Results

New Result:

An LCIS for a length-m and a length-n sequence can be computed in $O\Big((m + n\ell) \log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time, where $\ell =$ length of LCIS.

(essentially $O(n\ell)$) $(n \geq m)$

You *"usually"* expect quite small ℓ. So it's a *"good"* parameter!

LCIS Results

New Result:

An LCIS for a length-m and a length-n sequence can be computed in $O\Big((m+n\ell)\log\log|\Sigma| + \mathcal{S}ort_{\Sigma}(m)\Big)$ time, where $\ell =$ length of LCIS.

(essentially $O(n\ell)$) $(n \geq m)$

You *"usually"* expect quite small ℓ. So it's a *"good"* parameter!

Even $O(m)$ space possible using randomized data structures; then it's *expected* running time.

(uses Willard's y-fast tries)

Weakly-Increasing Subsequences

Both, LIS and LCIS consider strictly increasing subsequences.

What about the “weak” ($\leq$ instead of $<$) variant?

Weakly-Increasing Subsequences

Both, LIS and LCIS consider strictly increasing subsequences.

What about the “weak” ($\leq$ instead of $<$) variant:

LCWIS: longest common weakly increasing subsequence

(of two sequences over an ordered alphabet)

Weakly-Increasing Subsequences

Both, LIS and LCIS consider strictly increasing subsequences.

What about the “weak” ($\leq$ instead of $<$) variant:

LCWIS: longest common weakly increasing subsequence

(of two sequences over an ordered alphabet)

Our new result also applies (just replace $<$ by $\leq$ everywhere) but . . .

Weakly-Increasing Subsequences

Both, LIS and LCIS consider strictly increasing subsequences.

What about the “weak” ($\leq$ instead of $<$) variant:

LCWIS: longest common weakly increasing subsequence

(of two sequences over an ordered alphabet)

Our new result also applies (just replace $<$ by $\leq$ everywhere) but . . .

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

Weakly-Increasing Subsequences

Both, LIS and LCIS consider strictly increasing subsequences.

What about the “weak” ($\leq$ instead of $<$) variant:

LCWIS: longest common weakly increasing subsequence

(of two sequences over an ordered alphabet)

Our new result also applies (just replace $<$ by $\leq$ everywhere) but . . .

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

Why should this be interesting?

Weakly-Increasing Subsequences

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

Weakly-Increasing Subsequences

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

- For LCS the 2-letter case seems to be as hard as the general problem already.

Weakly-Increasing Subsequences

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

- For LCS the 2-letter case seems to be as hard as the general problem already.
- For LCIS the bounded-alphabet case can be done in near-linear time (our algorithm)

Weakly-Increasing Subsequences

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

- For LCS the 2-letter case seems to be as hard as the general problem already.
- For LCIS the bounded-alphabet case can be done in near-linear time (our algorithm)
- Complexity of LCWIS seems to lie somehow between the two

Weakly-Increasing Subsequences

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

- For LCS the 2-letter case seems to be as hard as the general problem already.
- For LCIS the bounded-alphabet case can be done in near-linear time (our algorithm)
- Complexity of LCWIS seems to lie somehow between the two

4-letter LCWIS remains open

Applications

Our LCIS algorithm

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell)\log\log|\Sigma| + Sort_\Sigma(m)\Big)$ time.

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell) \log\log |\Sigma| + \mathrm{Sort}_{\Sigma}(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell)\log\log|\Sigma| + \mathit{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell) \log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell)\log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell)\log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$

$L_1[1] = 8$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell) \log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$

$L_1[1] = 8$

$L_1[9] = 8$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell)\log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$

$L_1[1] = 8$

$L_1[9] = 8$

$L_2[4] = 9$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell) \log\log|\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$
$L_1[1] = 8$
$L_1[9] = 8$
$L_2[4] = 9$
$L_2[5] = 2$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Theorem. An LCIS for a length-m seq. A and a length-n seq. B can be computed in $O\Big((m + n\ell) \log\log |\Sigma| + \mathrm{Sort}_\Sigma(m)\Big)$ time.

A dynamic-programming approach, but not over the $A \times B$ table.

Instead, evaluate arrays $L_i[j]$: minimal index κ in B such that there exists lenght-i CIS on $A[1..i]$ and $B[1..\kappa]$ ending on a_i.

$L_1[4] = 3$
$L_1[1] = 8$
$L_1[9] = 8$
$L_2[4] = 9$
$L_2[5] = 2$
$L_3[8] = 6$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \ldots m]$ from $L_{i-1}[1 \ldots m]$

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \ldots m]$ from $L_{i-1}[1 \ldots m]$

"New" data structure: *Bounded Heaps*

combine McCreight's priority search tree with van Emde Boas trees

	1	2	3	4	5	6	7	8	9
$A:$	γ	α	α	β	δ	α	β	ϵ	γ
$B:$	β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \ldots m]$ from $L_{i-1}[1 \ldots m]$

“New” data structure: *Bounded Heaps*

combine McCreight’s priority search tree with van Emde Boas trees

- maintain collection of items, each with a *key* and a *priority*

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \dots m]$ from $L_{i-1}[1 \dots m]$

“New” data structure: *Bounded Heaps*

combine McCreight’s priority search tree with van Emde Boas trees

- maintain collection of items, each with a *key* and a *priority*
- query (k): minimum-priority item with key $< k$

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \ldots m]$ from $L_{i-1}[1 \ldots m]$

“New” data structure: *Bounded Heaps*

combine McCreight’s priority search tree with van Emde Boas trees

- maintain collection of items, each with a *key* and a *priority*
- query (k): minimum-priority item with key $< k$
- insert (item,key,priority) and decrease_key (item,key)

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \dots m]$ from $L_{i-1}[1 \dots m]$

"New" data structure: *Bounded Heaps*

combine McCreight's priority search tree with van Emde Boas trees

- maintain collection of items, each with a *key* and a *priority*
- query (k): minimum-priority item with key $< k$
- insert (item,key,priority) and decrease_key (item,key)

items: length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)

key: the letter $a_h = b_k$

priority: the index k (in B)

Our LCIS algorithm

Evaluate arrays L_i one after another:

compute $L_i[1 \ldots m]$ from $L_{i-1}[1 \ldots m]$

"New" data structure: *Bounded Heaps*

combine McCreight's priority search tree with van Emde Boas trees

- maintain collection of items, each with a *key* and a *priority*
- query (k): minimum-priority item with key $< k$
- insert (item,key,priority) and decrease_key (item,key)

items: length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)

key: the letter $a_h = b_k$

priority: the index k (in B)

Each operation in $O(\log \log |\Sigma|)$ time

Our LCIS algorithm

query (k): minimum-priority item with key $< k$

items: length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)

key: the letter $a_h = b_k$

priority: the index k (in B)

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

query (k):	minimum-priority item with key $< k$
items:	length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)
key:	the letter $a_h = b_k$
priority:	the index k (in B)

Example: want to compute $L_3[8]$

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

query (k):	minimum-priority item with key $< k$
items:	length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)
key:	the letter $a_h = b_k$
priority:	the index k (in B)

Example: want to compute $L_3[8]$

- query (ϵ) *"where does longest length-2 sequence with last letter* $< \epsilon$ *end in* B*?"*

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

query (k):	minimum-priority item with key $< k$
items:	length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)
key:	the letter $a_h = b_k$
priority:	the index k (in B)

Example: want to compute $L_3[8]$

- query (ϵ) *"where does longest length-2 sequence with last letter $< \epsilon$ end in B?"*
- answer: *at position 2*

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

query (k): minimum-priority item with key $< k$

items: length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)

key: the letter $a_h = b_k$

priority: the index k (in B)

Example: want to compute $L_3[8]$

- query (ϵ) *"where does longest length-2 sequence with last letter $< \epsilon$ end in B?"*
- answer: *at position 2*
- find next occurence of ϵ after position 2 in B: 6

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

Our LCIS algorithm

query (k):	minimum-priority item with key $< k$
items:	length-$(i-1)$ CIS ending on $a_h = b_k$ (in A resp. B)
key:	the letter $a_h = b_k$
priority:	the index k (in B)

Example: want to compute $L_3[8]$

- query (ϵ) *"where does longest length-2 sequence with last letter $< \epsilon$ end in B?"*
- answer: *at position 2*
- find next occurence of ϵ after position 2 in B: 6
- set new value $L_3[8] := 6$

1	2	3	4	5	6	7	8	9
γ	α	α	β	δ	α	β	ϵ	γ
β	δ	β	α	α	ϵ	δ	γ	β

LCWIS with Two Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

LCWIS with Two Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

2-Letter case is simple:

- every potential solution is of the form $\alpha^r \beta^s$

LCWIS with Two Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

2-Letter case is simple:

- every potential solution is of the form $\alpha^r \beta^s$
- for every $r \leq m$ do
  - find leftmost occurence of α^r in A and B

LCWIS with Two Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

2-Letter case is simple:

- every potential solution is of the form $\alpha^r \beta^s$
- for every $r \leq m$ do
  - find leftmost occurence of α^r in A and B
  - fill up to the right with maximum number of β's

LCWIS with Two Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

2-Letter case is simple:

- every potential solution is of the form $\alpha^r \beta^s$
- for every $r \leq m$ do
  - find leftmost occurence of α^r in A and B
  - fill up to the right with maximum number of β's
- take the best result over all r

$\longrightarrow O(m)$ time

LCWIS with Three Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

LCWIS with Three Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

3-Letter case is *not* so simple!

LCWIS with Three Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

3-Letter case is *not* so simple!

- every potential solution is of the form $\alpha^r \beta^s \gamma^t$

LCWIS with Three Letters

Theorem.

We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

3-Letter case is *not* so simple!

- every potential solution is of the form $\alpha^r \beta^s \gamma^t$
- naive implementation would require quadratic time

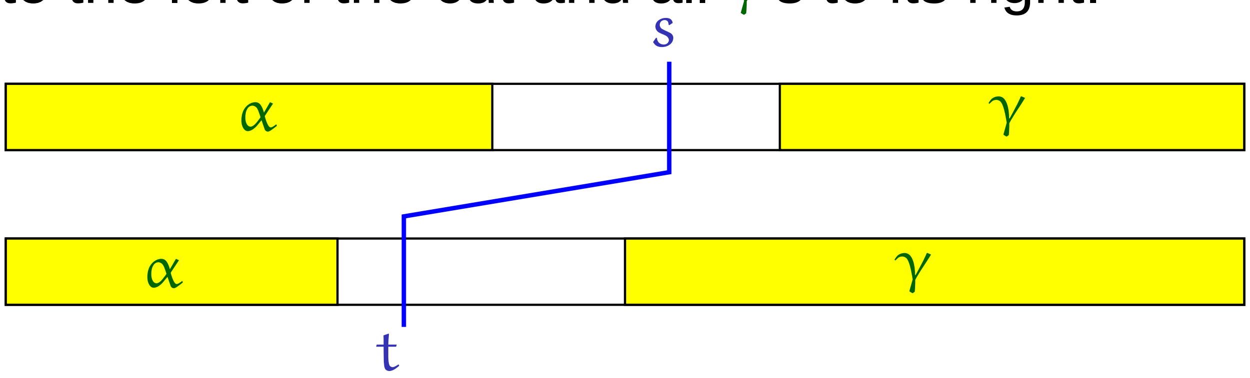
LCWIS with Three Letters

Theorem.
We can compute an LCWIS over a 2-letter alphabet in linear time, and over a 3-letter alphabet in $O(m + n \log n)$ time.

3-Letter case is *not* so simple!

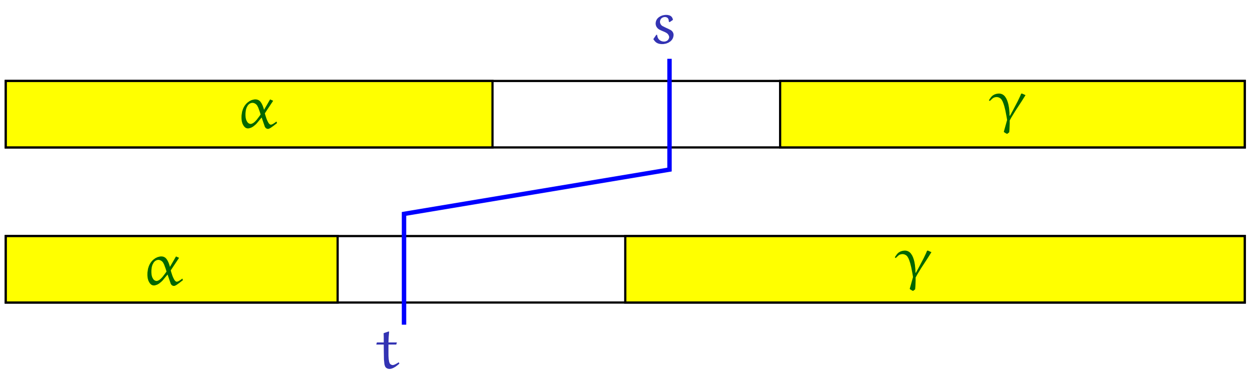
- every potential solution is of the form $\alpha^r \beta^s \gamma^t$
- naive implementation would require quadratic time

Idea: Guess a *cut* $(s, t) \in A \times B$ and consider only solutions with all α's to the left of the cut and all γ's to its right.



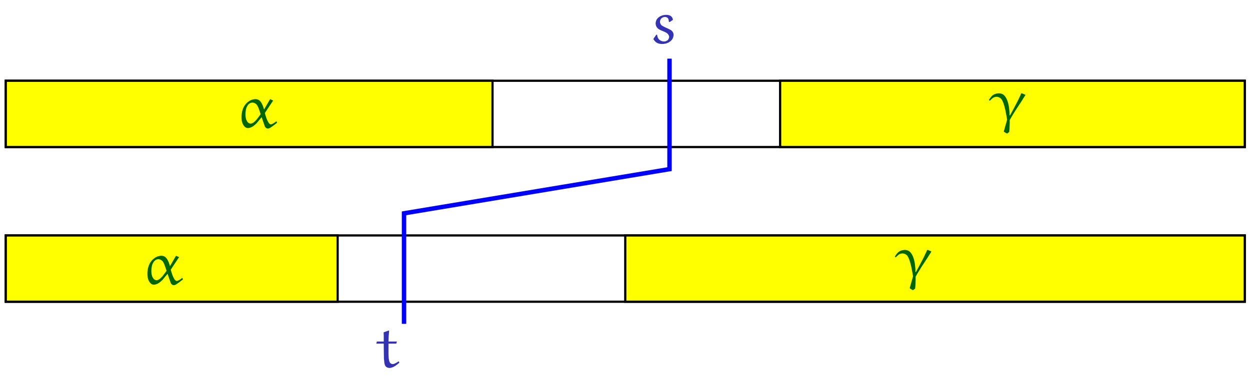
LCWIS with Three Letters

Idea: Guess a *cut* $(s,t) \in A \times B$ and consider only solutions with all α's to the left of the cut and all γ's to its right.



LCWIS with Three Letters

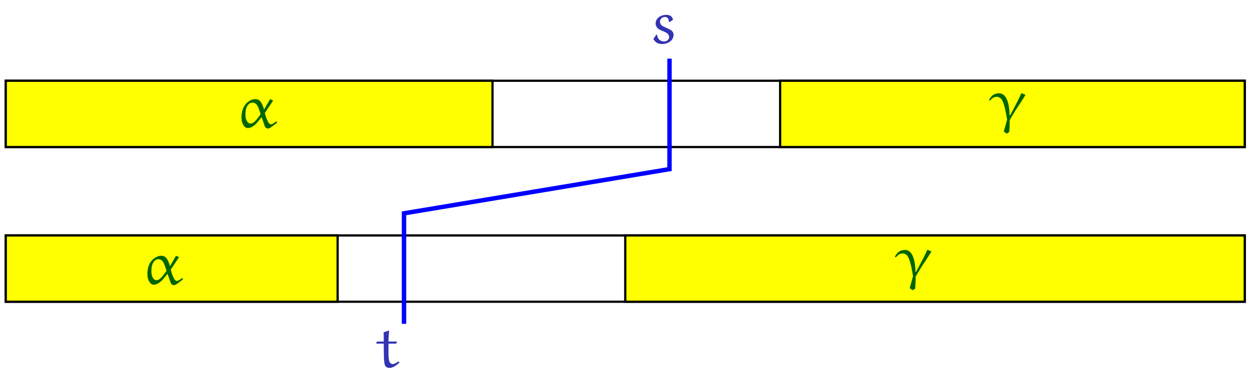
Idea: Guess a *cut* $(s, t) \in A \times B$ and consider only solutions with all α's to the left of the cut and all γ's to its right.



Now enter all "α-*information*" into the cut in linear time
and then check all "γ-*information*" against the cut in linear time.

LCWIS with Three Letters

Idea: Guess a *cut* $(s, t) \in A \times B$ and consider only solutions with all α's to the left of the cut and all γ's to its right.

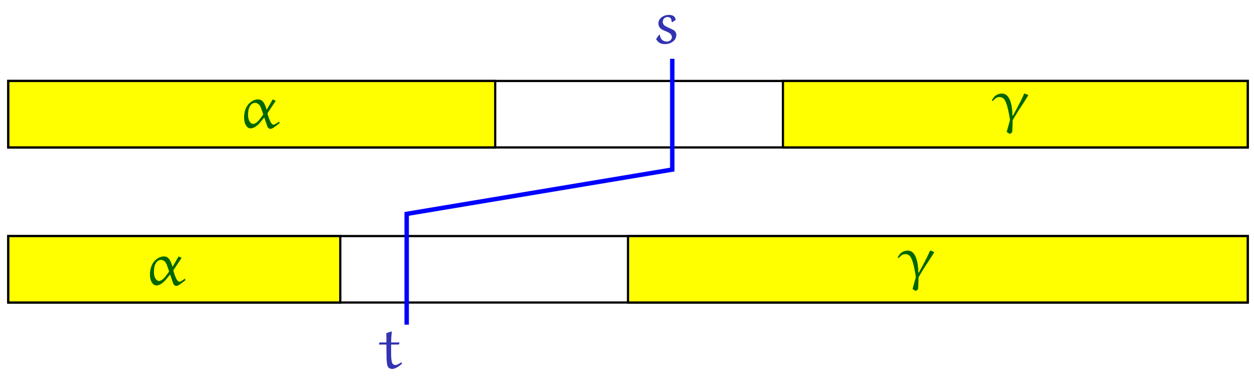


Now enter all "α-*information*" into the cut in linear time
and then check all "γ-*information*" against the cut in linear time.

Gives linear time per cut $\longrightarrow$ cubic total time!

LCWIS with Three Letters

Idea: Guess a *cut* $(s, t) \in A \times B$ and consider only solutions with all α's to the left of the cut and all γ's to its right.



Now enter all "α-*information*" into the cut in linear time and then check all "γ-*information*" against the cut in linear time.

Gives linear time per cut $\longrightarrow$ cubic total time!

A hierarchical distribution of information reduces all information storage to $O(m + n \log n)$ time.

Multiple Sequences

Theorem.

An LCIS or LCWIS of k length-n sequences can be computed in $O(r \log^{k-1} \log\log r)$ time, where $r =$ # of match vectors.

Multiple Sequences

Theorem.

An LCIS or LCWIS of k length-n sequences can be computed in $O(r \log^{k-1} \log\log r)$ time, where $r = \#$ of match vectors.

$\longrightarrow$

4 Multiple Sequences

In this section we consider the problem of finding an LCIS of k length-n sequences, for $k \geq 3$. We will denote the sequences by $A^1 = (a_1^1, \ldots, a_n^1)$, $A^2 = (a_1^2, \ldots, a_n^2)$, ..., $A^k = (a_1^k, \ldots, a_n^k)$. A *match* is a vector $(i_1, i_2, \ldots, i_k)$ of indices such that $a_{i_1}^1 = a_{i_2}^2 = \cdots = a_{i_k}^k$. Let r be the number of matches. Chan et al. [4] showed that an LCIS can be found in $O(\min(kr^2, kr \log\sigma \log^{k-1} r) + kSort_\Sigma(n))$ time (they present two algorithms, each corresponding to one of the terms in the min). We present a simpler solution which replaces the second term by $O(r \log^{k-1} r \log\log r)$.

We denote the ith coordinate of a vector v by $v[i]$, and the alphabet symbol corresponding to the match described by a vector v will be denoted $s(v)$. A vector v *dominates* a vector v' if $v[i] > v'[i]$ for all $1 \leq i \leq k$, and we write $v' < v$. Clearly, an LCIS corresponds to a sequence $v_1, \ldots, v_\ell$ of matches such that $v_1 < v_2 < \cdots < v_\ell$ and $s(v_1) < s(v_2) < \cdots < s(v_\ell)$.

To find an LCIS, we use a data structure by Gabow et al. [6, Theorem 3.3], which stores a fixed set of n vectors from $\{1, \ldots, n\}^k$. Initially all vectors are *inactive*. The data structure supports the following two operations:

1. *Activate* a vector with an associated priority.
2. A query of the form "what is the maximum priority of an active vector that is dominated by a vector p?"

A query takes $O(\log^{k-1} n \log\log n)$ time and the total time for at most n activations is $O(n \log^{k-1} n \log\log n)$. The data structure requires $O(n \log^{k-1} n)$ preprocessing time and space.

Each of the r matches $v = (v_1, \ldots, v_k)$ corresponds to a vector. The priority of v will be the length of the longest LCIS that ends at the match v. We will consider the matches by non-decreasing order of their symbols. For each symbol s of the alphabet, we first compute the priority of every match v with $s(v) = s$. This is equal to 1 plus the maximum priority of a vector dominated by v. Then, we activate these vectors in the data structure with the priorities we have computed; they should be there when we compute the priorities for matches v with $s(v) > s$.

The algorithm applies to the case of a common weakly-increasing subsequence by the following modification: The matches will be considered by non-decreasing order of $s(v)$ as before, but within each symbol also in non-decreasing lexicographic order of v. For each match, we compute its priority and immediately activate it in the data structure (so that it is active when considering other matches with the same symbol). The lexicographic order ensures that if $v > v'$ then v' is in the data structure when v is considered.

Theorem 4. *An LCIS or LCWIS of k length-n sequences can be computed in $O(r \log^{k-1} r \log\log r)$ time, where r counts the number of match vectors.*

5 Outlook

The central question about the LCS problems is, whether it can be solved in $O(n^{2-\epsilon})$ time in general. It seems that with LCIS we face the same frontier. Our

Open Problems

- Can you do the Four-Russians Trick for LCIS?
  (get something like $O(n^2 \log\log n / \log n)$)

Open Problems

- Can you do the Four-Russians Trick for LCIS?
  (get something like $O(n^2 \log\log n / \log n)$)
- Can you extend the near-linear running time for LCWIS to 4,5,. . . -letter alphabets?

Open Problems

- Can you do the Four-Russians Trick for LCIS?
  (get something like $O(n^2 \log\log n / \log n)$)
- Can you extend the near-linear running time for LCWIS
  to 4,5,...-letter alphabets?
- With LCS, is the 2-letter case as hard as the general problem?