SimdMinimizers: Computing random minimizers, *fast*

Ragnar Groot Koerkamp, **Igor Martayan**

@curiouscoding.nl @imartayan

SEA 2025, July 24, Venice

martayan.org/slides-sea-25.pdf

ETH zürich
Université de Lille
CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

(Random) Minimizers

Given a **window of `w` consecutive *k*-mers**, select the **"smallest" *k*-mer**.

Random minimizer: select the *k*-mer with the **smallest hash**.

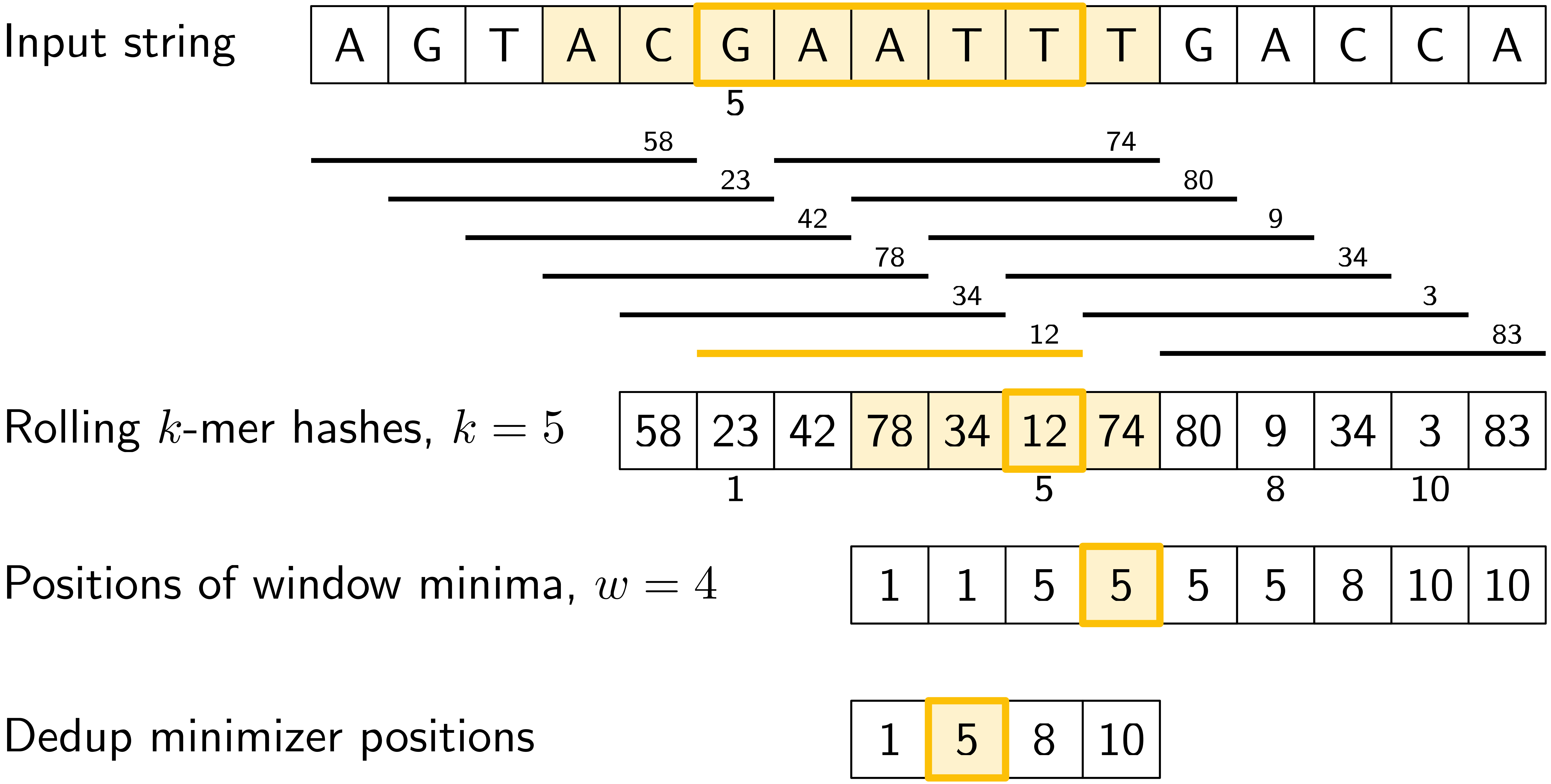
```
TGACATCGACGTT

TGACAT (TGA GAC ACA CAT)
 GACATC       w=4  k=3
  ACATCG
   CATCGA
    ATCGAC
     TCGACG
      CGACGT
       GACGTT
```

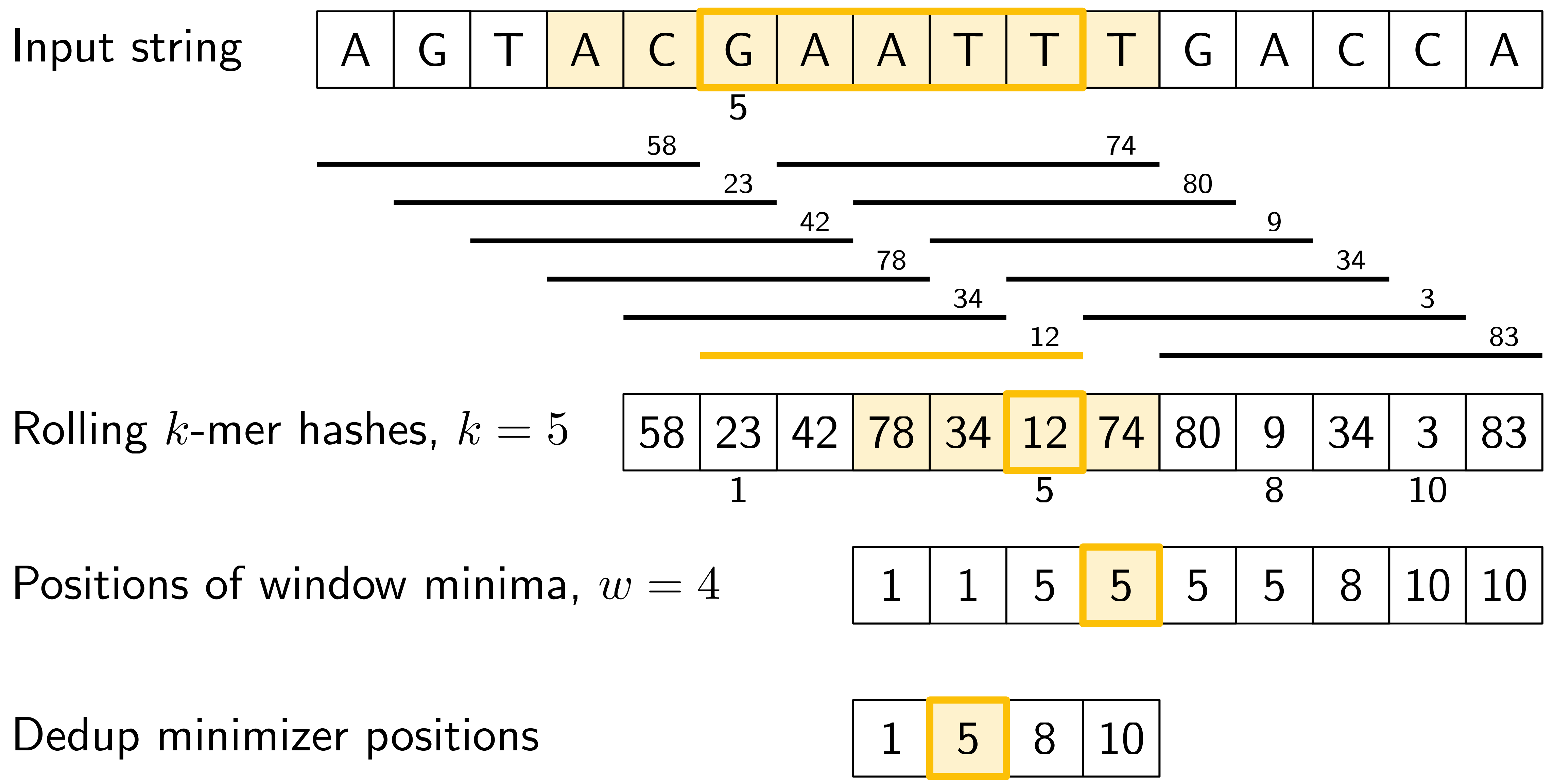
Used everywhere in bioinformatics:
indexing, counting, mapping, assembling...

Problem statement:
Given a (large) DNA sequence, return the starting positions of its random minimizers.

Overview of the problem



Designing a vectorized version



Two constraints for vectorization:

1. Avoid **sequential dependencies** between lanes

2. Avoid **branches** when computing sliding window minima

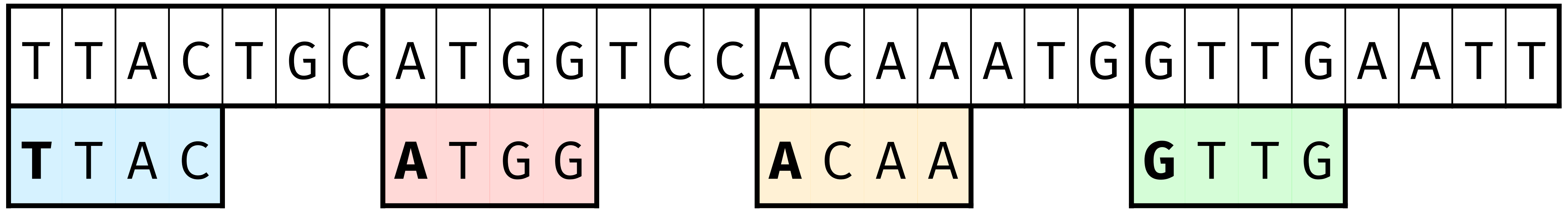
Iterating packed input

T	T	A	C	T	G	C	A	T	G	G	T	C	C	A	C	A	A	A	T	G	G	T	T	G	A	A	T	T

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped

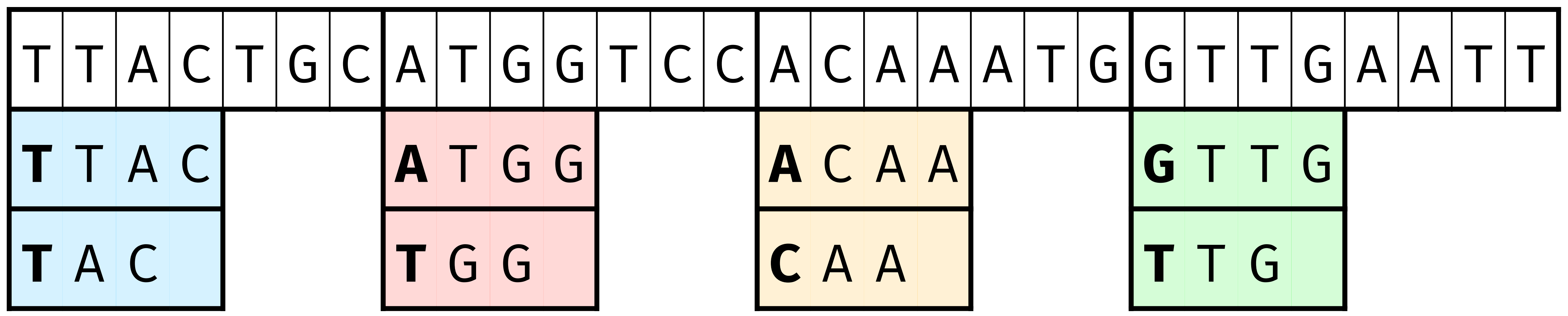
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



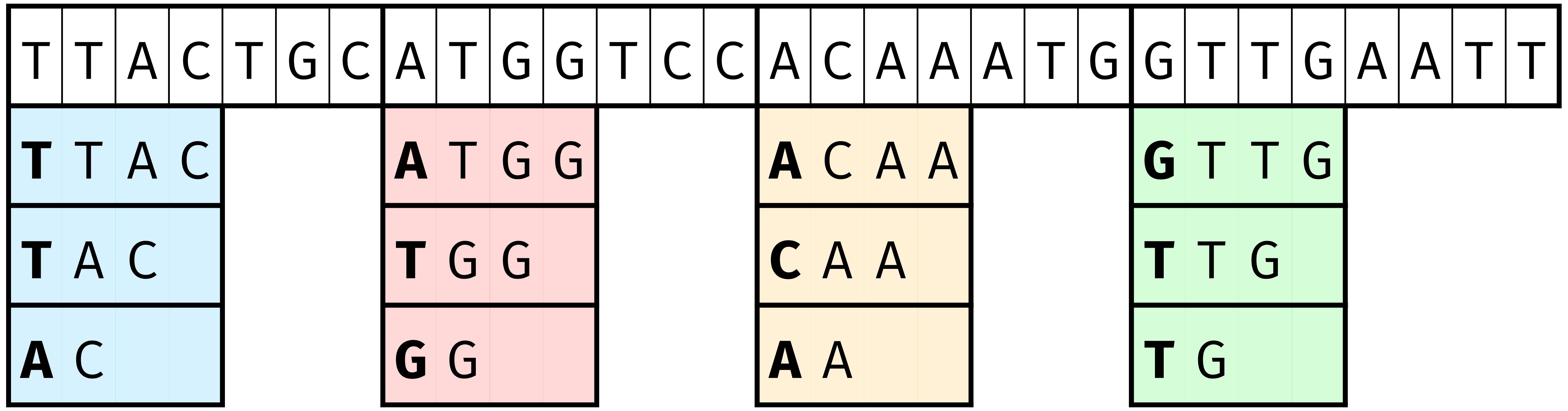
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



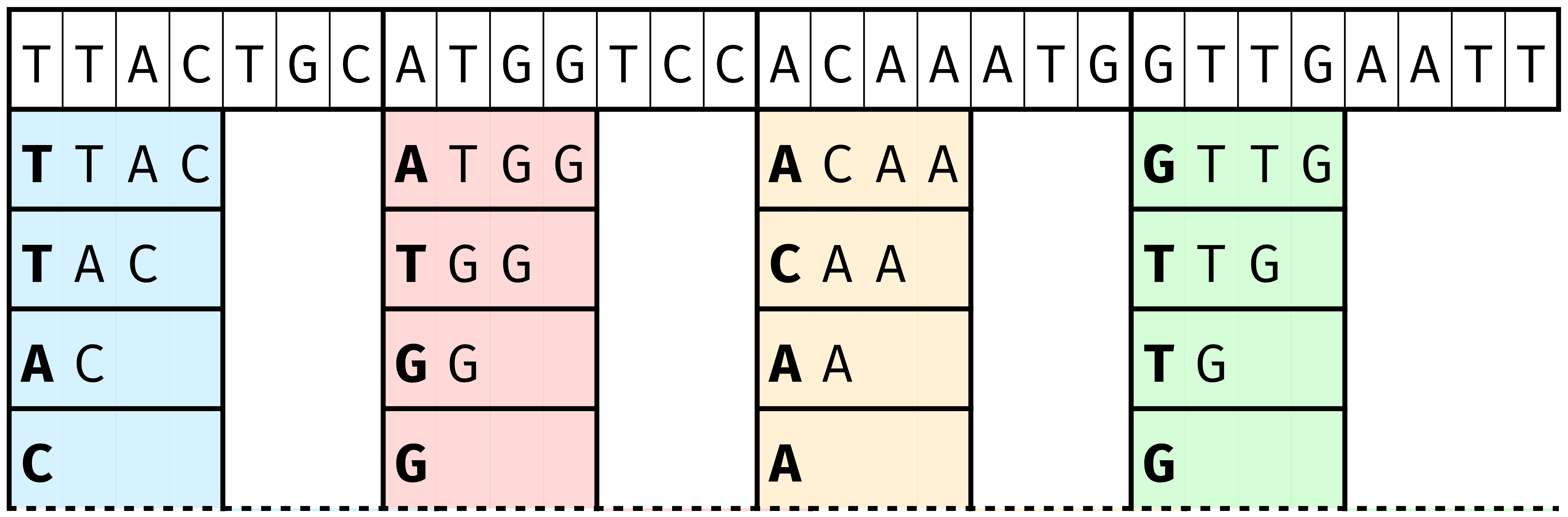
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



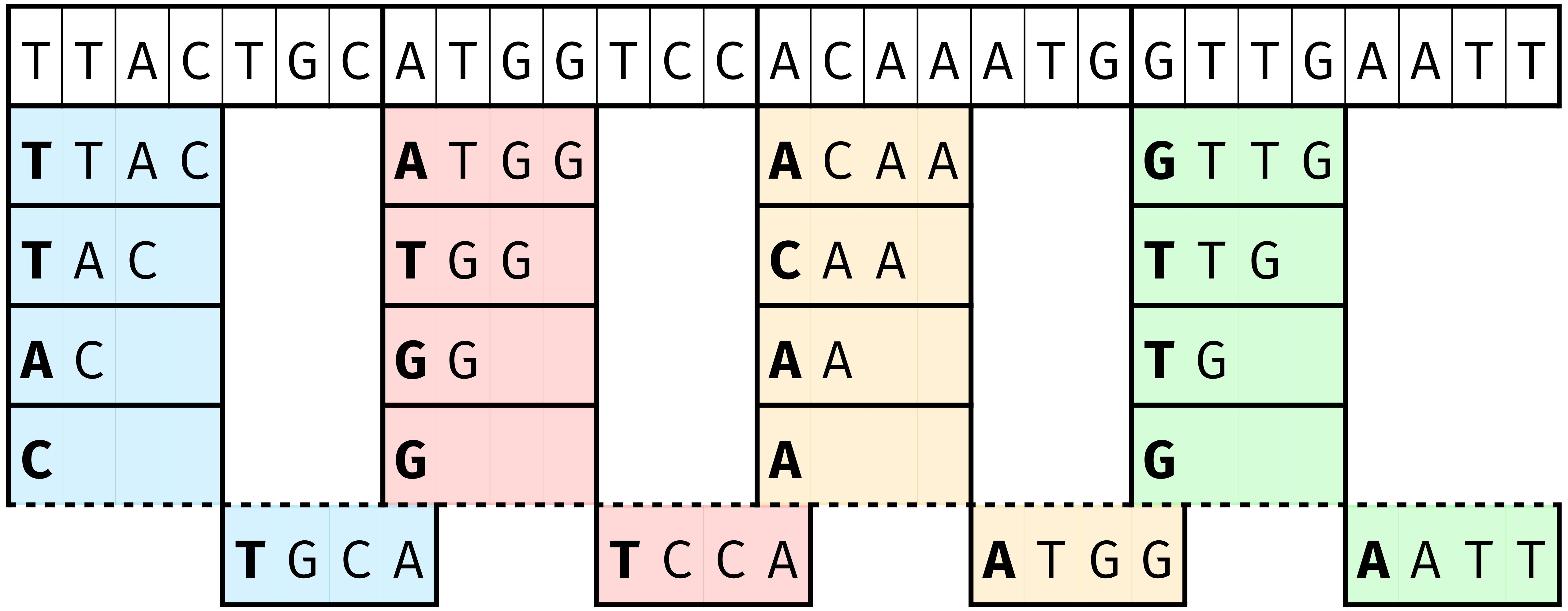
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



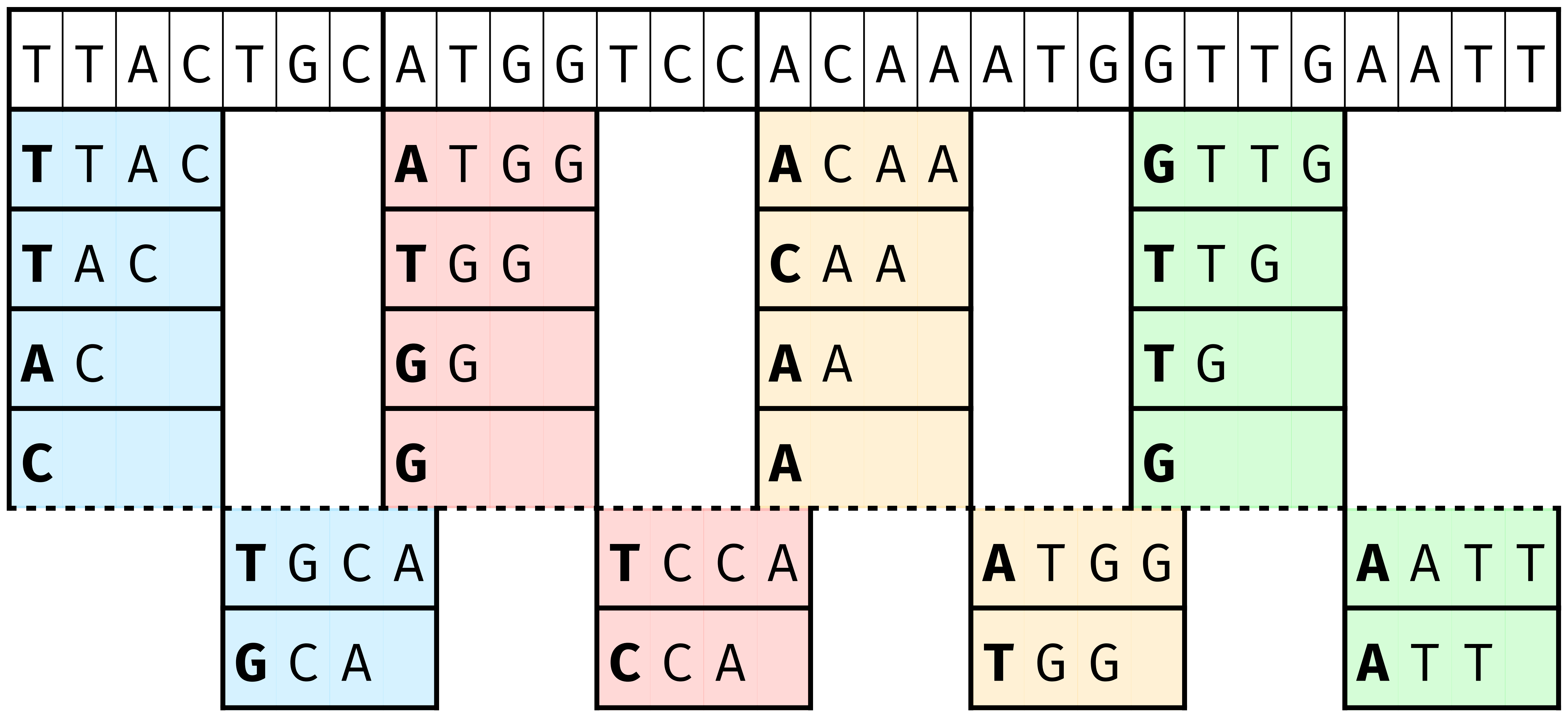
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



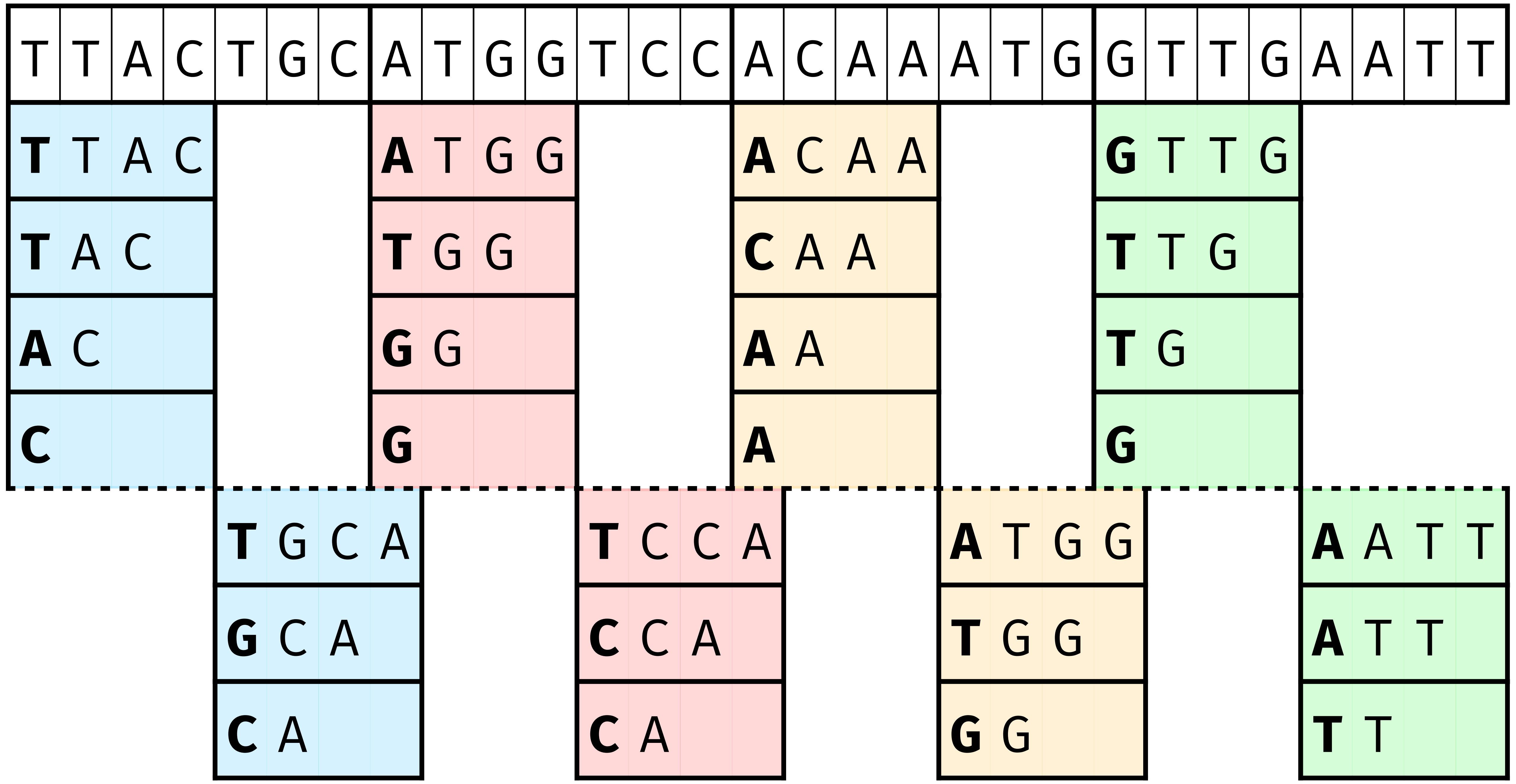
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



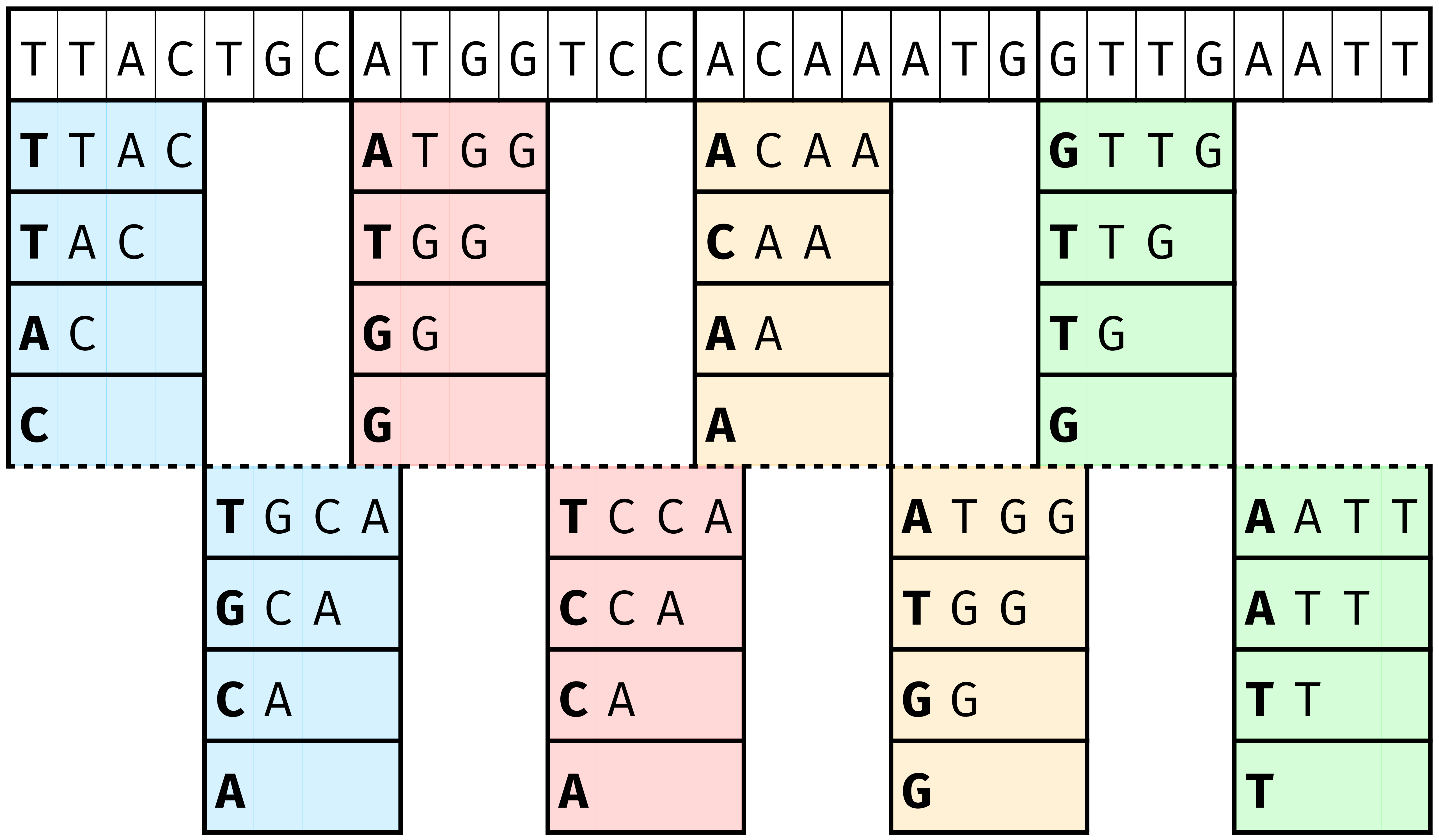
Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



Iterating packed input

- Split input in **L** overlapping chunks
- Each lane contains packed bases from one of the chunks
- Chunks overlap by $w + k - 2$ bases so that no window is skipped



Rolling hash (NtHash)

…	G	G	C	A	T	G	…

hash of GGCAT	1	0	1	1	0	0	0	1
rotated left by 1	0	1	1	0	0	0	1	1
XOR								
h(G)	1	0	0	0	1	0	1	1
XOR								
h(G) rotated *k* times	0	1	1	1	0	0	0	1
hash of GCATG	1	0	0	1	1	0	0	1

In practice:

- Two iterators ***k*** positions apart for bases coming in and out
- Hashes stored in **L** lanes of 32 bits

Existing sliding min algorithms

- **Naive**: for each window, scan all values to find the minimum $\Rightarrow O(nw)$
- **Rescan**: only keep track of the minimum, if it goes out of scope rescan the whole window to find the new minimum $\Rightarrow O(nw)$ worst case, $O(n)$ average
- **Monotone queue**: store an increasing sequence of minima, discard values that are followed by a smaller one $\Rightarrow O(n)$

Naive	Rescan	Queue
$O(nw)$	$O(n)$ avg	$O(n)$
branchless	branch for rescan	branches for every update

A branchless sliding min algorithm

w=5	2	1	7	4	5	9	6	3	5	8	2	…

"Two stacks" algorithm:

Split into chunks of size `w`,

Given 2 chunks `[`x_1`,…,`x_w`] [`y_1`,…,`y_w`]`

- suffix stack: `S[i] = min[`x_i`,…,`x_w`]`
- prefix stack: `P[i] = min[`y_1`,…,`y_i`]`

`i`th window min: `min(S[i],P[i-1])`

$\Rightarrow$ $O(n)$ & branchless!

A branchless sliding min algorithm

"Two stacks" algorithm:

Split into chunks of size `w`,

Given 2 chunks $[x_1,\dots,x_w]$ $[y_1,\dots,y_w]$

- suffix stack: $S[i] = \min[x_i,\dots,x_w]$
- prefix stack: $P[i] = \min[y_1,\dots,y_i]$

i^{th} window min: $\min(S[i],P[i-1])$

$\Rightarrow$ $O(n)$ & branchless!

w=5												
	2	1	7	4	5	9	6	3	5	8	2	…
	1	1	4	4	5							

A branchless sliding min algorithm

"Two stacks" algorithm:

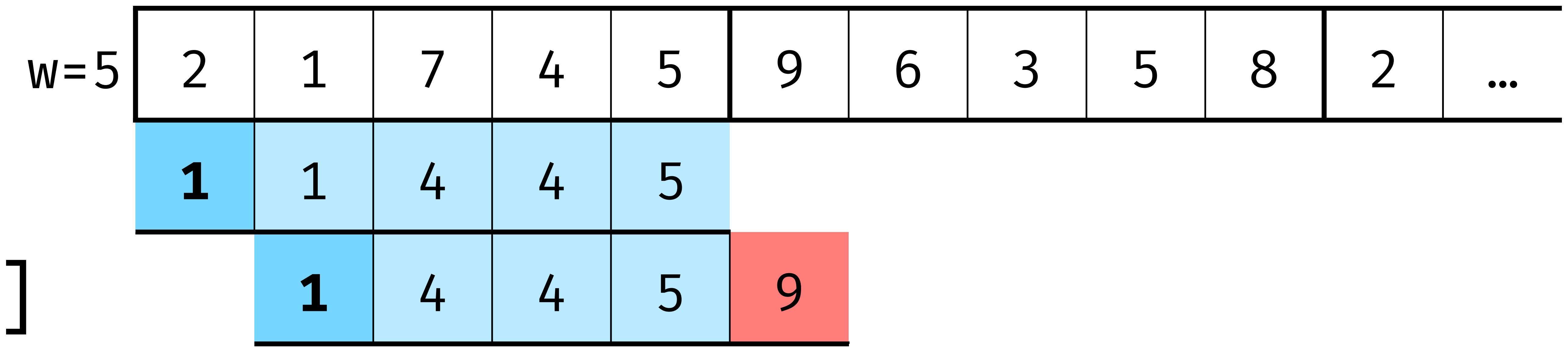
Split into chunks of size `w`,

Given 2 chunks $[x_1,\ldots,x_w]$ $[y_1,\ldots,y_w]$

- suffix stack: $S[i] = \min[x_i,\ldots,x_w]$
- prefix stack: $P[i] = \min[y_1,\ldots,y_i]$

i^{th} window min: $\min(S[i], P[i-1])$

$\Rightarrow$ $O(n)$ & branchless!



A branchless sliding min algorithm

"Two stacks" algorithm:

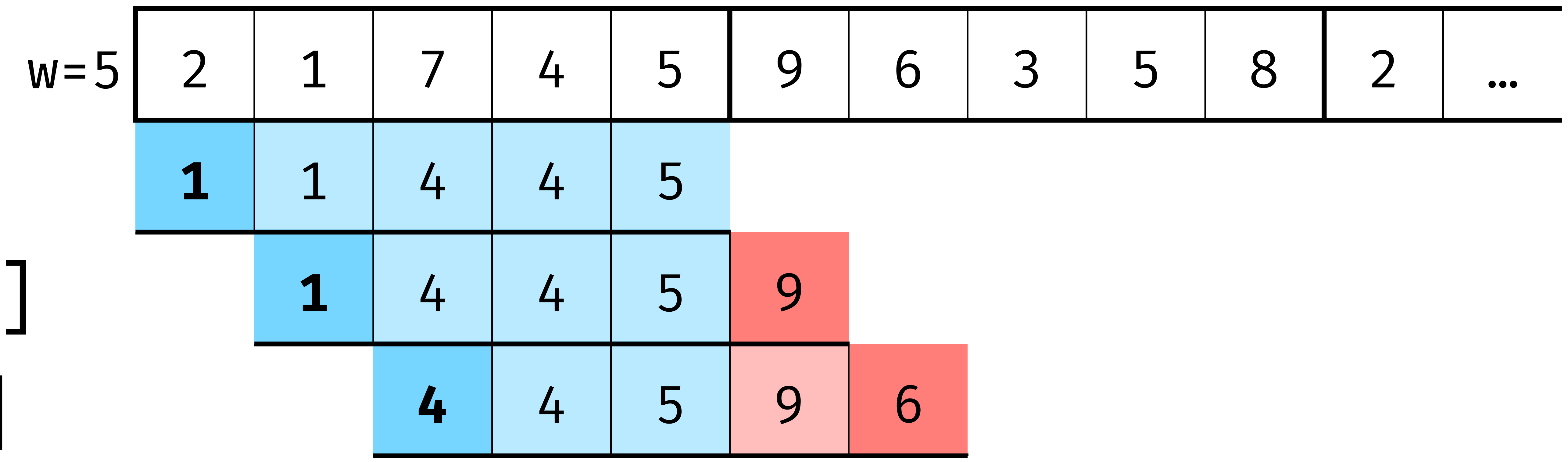
Split into chunks of size `w`,

Given 2 chunks $[x_1, \ldots, x_w]$ $[y_1, \ldots, y_w]$

- suffix stack: $S[i] = \min[x_i, \ldots, x_w]$
- prefix stack: $P[i] = \min[y_1, \ldots, y_i]$

i^{th} window min: $\min(S[i], P[i-1])$

⇒ $O(n)$ & branchless!



A branchless sliding min algorithm

"Two stacks" algorithm:

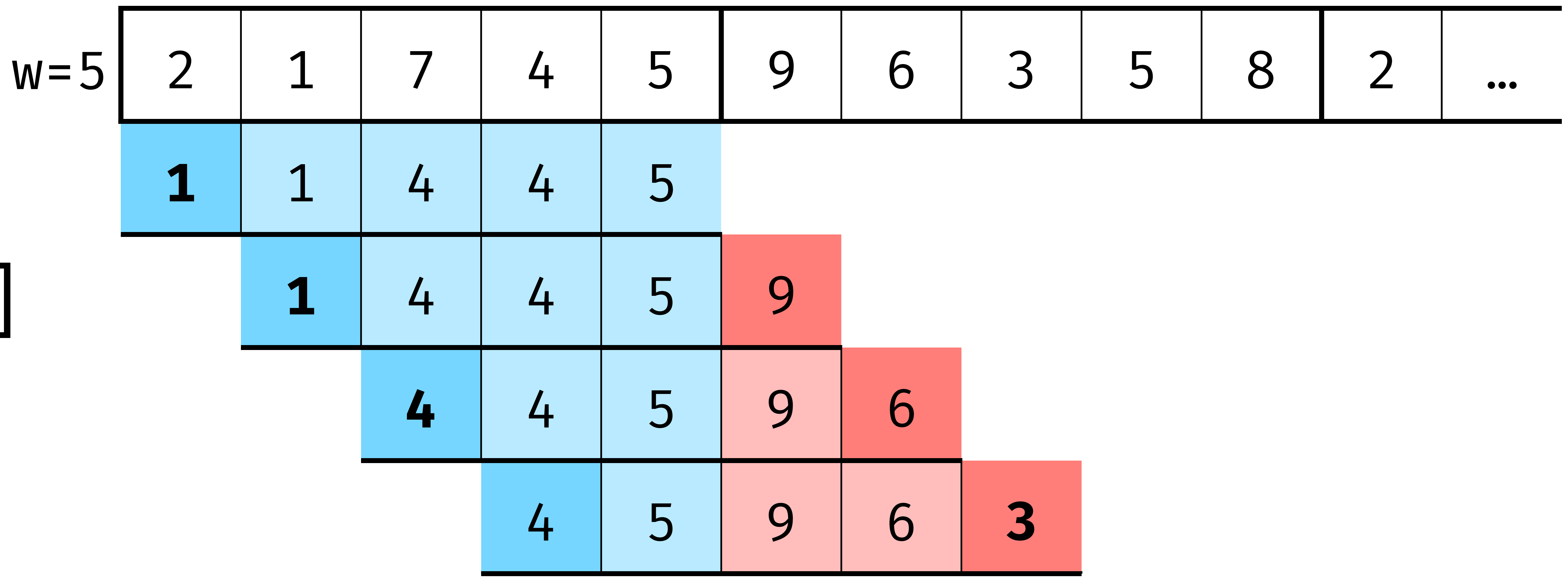
Split into chunks of size `w`,

Given 2 chunks $[x_1,\dots,x_w]$ $[y_1,\dots,y_w]$

- suffix stack: $S[i] = \min[x_i,\dots,x_w]$
- prefix stack: $P[i] = \min[y_1,\dots,y_i]$

i^{th} window min: $\min(S[i], P[i-1])$

$\Rightarrow$ $O(n)$ & branchless!



A branchless sliding min algorithm

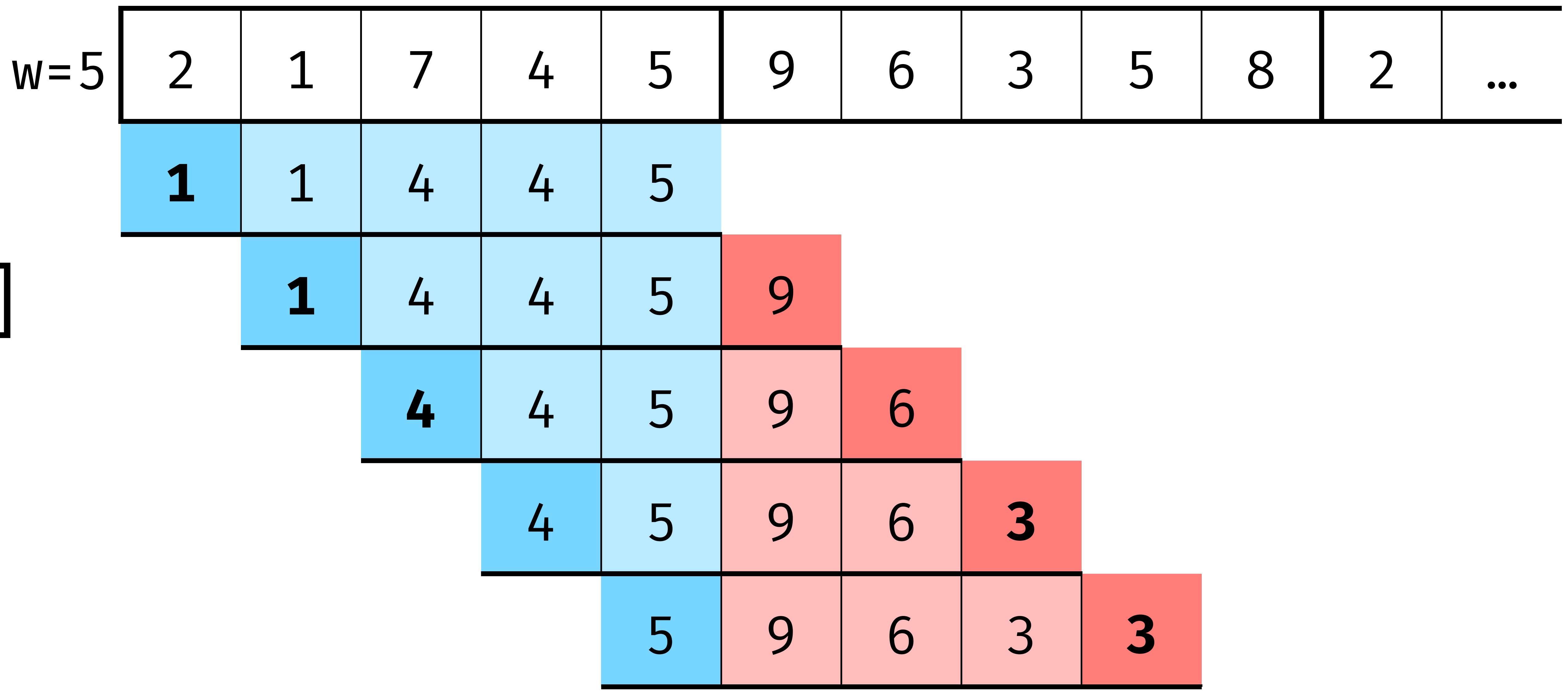
"Two stacks" algorithm:

Split into chunks of size `w`,

Given 2 chunks `[x₁,…,xw] [y₁,…,yw]`

- suffix stack: `S[i] = min[xi,…,xw]`
- prefix stack: `P[i] = min[y1,…,yi]`

`i`th window min: `min(S[i],P[i-1])`



$\Rightarrow$ $O(n)$ & branchless!

A branchless sliding min algorithm

"Two stacks" algorithm:

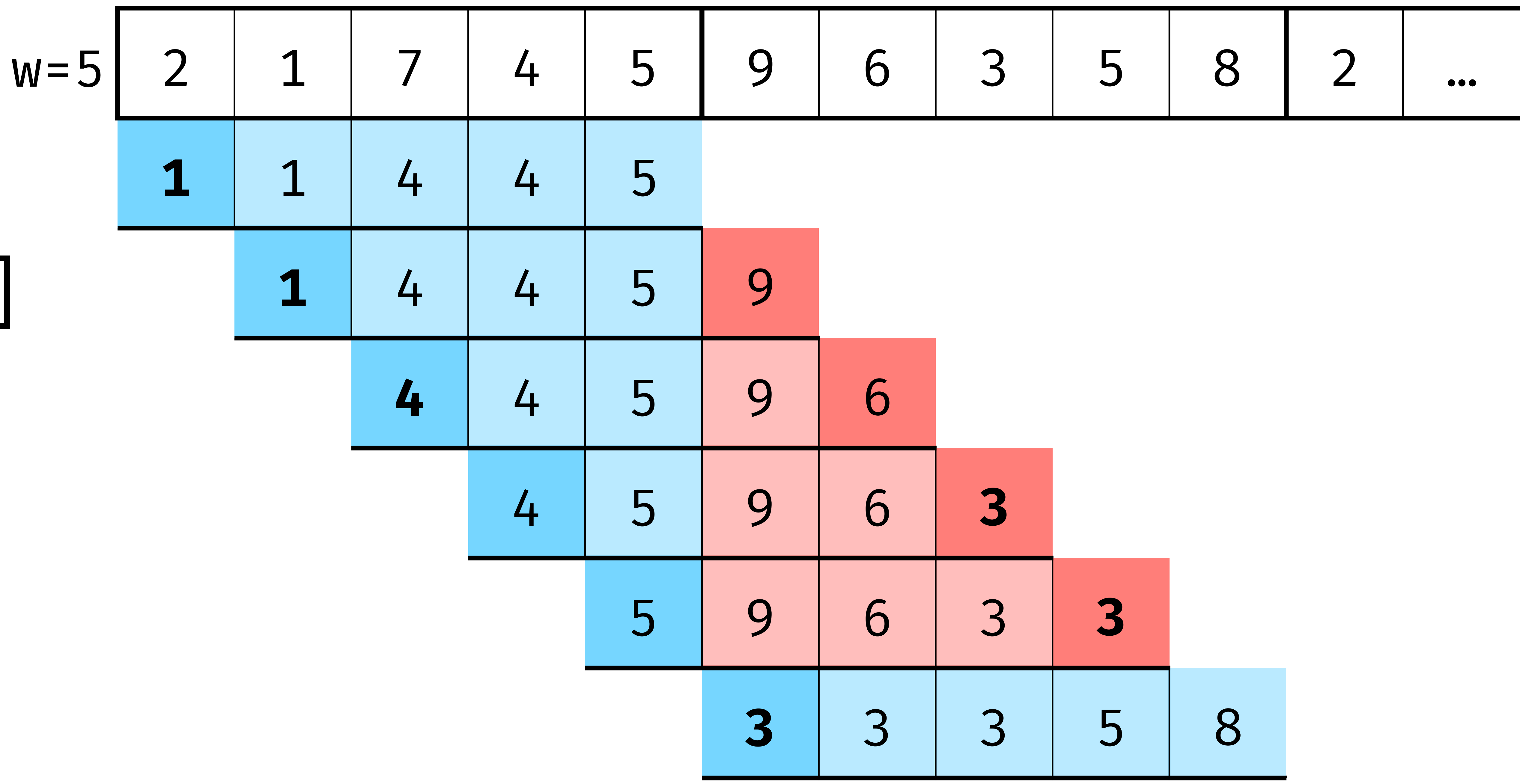
Split into chunks of size `w`,

Given 2 chunks $[x_1,\ldots,x_w]$ $[y_1,\ldots,y_w]$

- suffix stack: $S[i] = \min[x_i,\ldots,x_w]$
- prefix stack: $P[i] = \min[y_1,\ldots,y_i]$

i^{th} window min: $\min(S[i], P[i-1])$

$\Rightarrow O(n)$ & branchless!



A branchless sliding min algorithm

"Two stacks" algorithm:

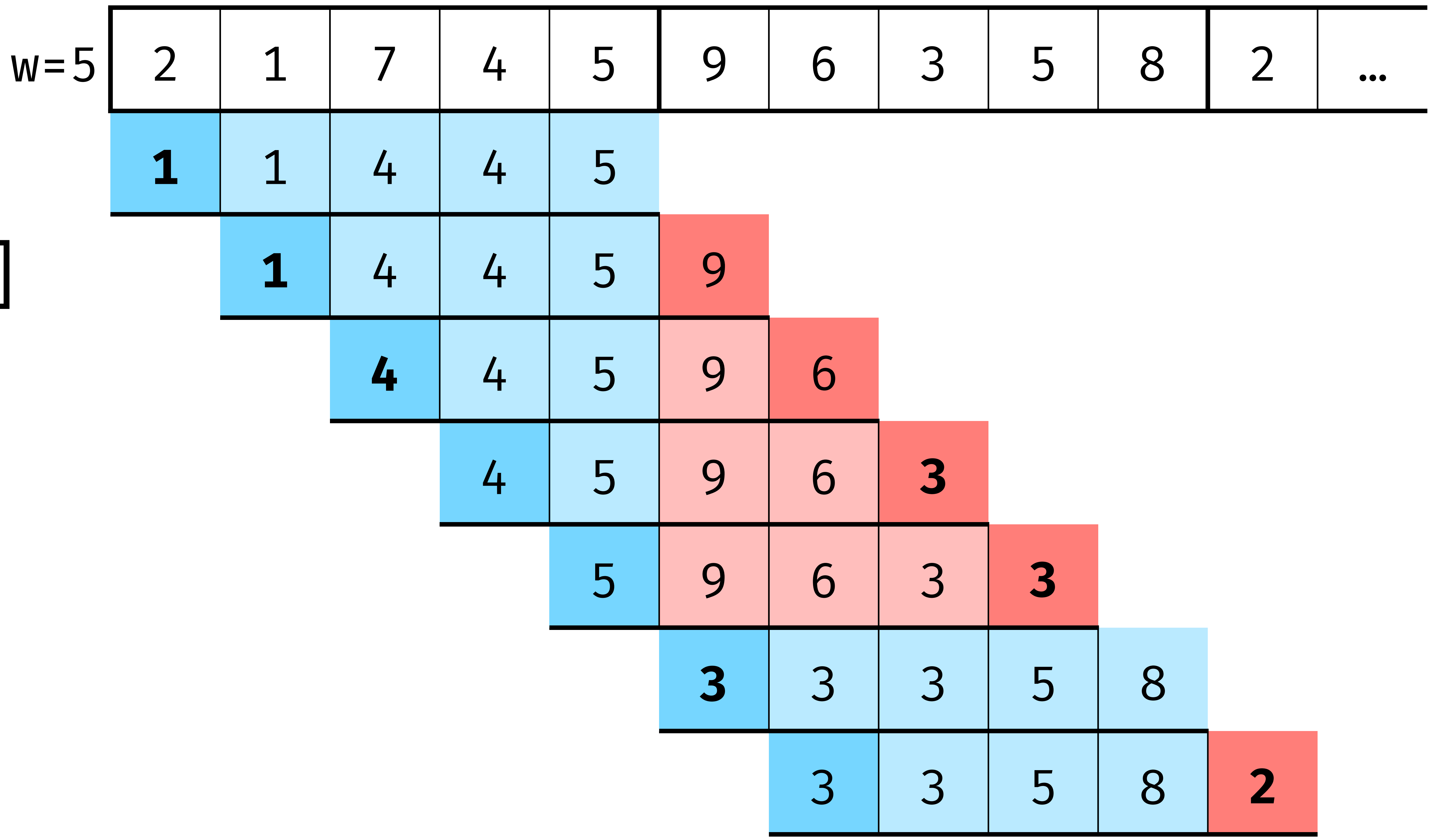
Split into chunks of size `w`,

Given 2 chunks $[x_1,\dots,x_w]$ $[y_1,\dots,y_w]$

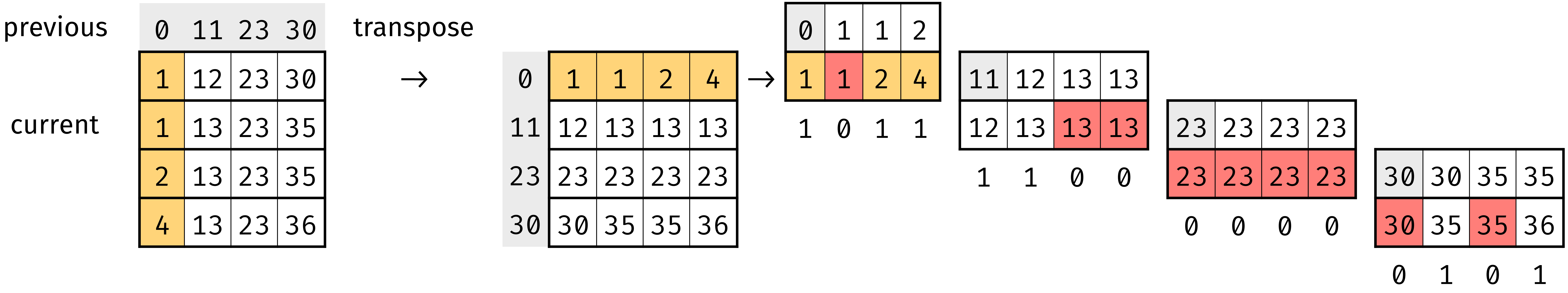
- suffix stack: $S[i] = \min[x_i,\dots,x_w]$
- prefix stack: $P[i] = \min[y_1,\dots,y_i]$

i^{th} window min: $\min(S[i], P[i-1])$

$\Rightarrow O(n)$ & branchless!



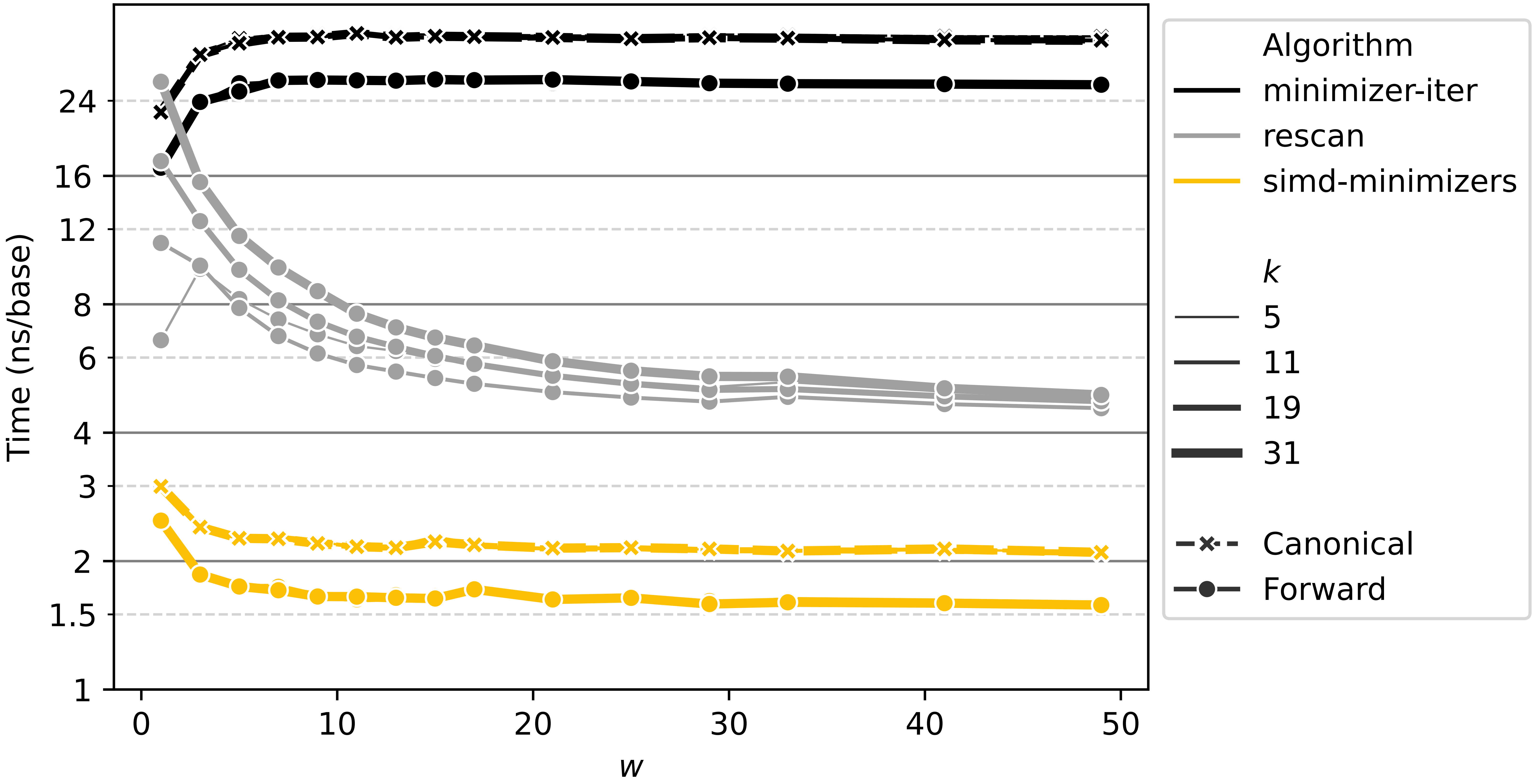
Deduplicating minimizer positions



Adapted from Daniel Lemire's blog

lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/

Performance of SimdMinimizers



Our SIMD algorithm takes **~1.6 ns/bp** (on an i7-10750H@2.6GHz).

Computing all minimizers in a human genome takes **~5s** (single core).

Take-home messages

- SimdMinimizers is 4-7x faster than existing methods
- Can be further parallelized on multiple threads
- Supports both AVX2 and NEON
- Already used in many Rust bioinformatics tools
- Future work: Python/C++ bindings, other minimizer schemes
- Try it at `github.com/rust-seq/simd-minimizers`

Paper

GitHub

Thank you!