Optimal-Time Mapping in Run-Length Compressed PBWT

Paola Bonizzoni, Davide Cozzi, Younan Gao

University of Milano-Bicocca, Italy

DSB 2026, Venice, Italy

Biological Context: Haplotypes and Bi-allelic Panels

- **Haplotypes**: A haplotype refers to a set of genetic variants (alleles) that are inherited together from a single parent.

S_1	0	1	0	1	1

Biological Context: Haplotypes and Bi-allelic Panels

- **Haplotypes**: A haplotype refers to a set of genetic variants (alleles) that are inherited together from a single parent.
- **Variant Sites (`w`)**: Genomic locations where sequences vary across a population are called variant sites.

Haplotype	Site 1	Site 2	Site 3	Site 4	Site 5
S_1	0	1	0	1	1

Biological Context: Haplotypes and Bi-allelic Panels

- **Haplotypes**: A haplotype refers to a set of genetic variants (alleles) that are inherited together from a single parent.
- **Variant Sites (w)**: Genomic locations where sequences vary across a population are called variant sites.
- **Bi-allelic Panels**: In most genomic studies, these sites are "bi-allelic," meaning they only exhibit two possible states, typically represented as 0 or 1.

Haplotype	Site 1	Site 2	Site 3	Site 4	Site 5
S_1	0	1	0	1	1

Biological Context: Haplotypes and Bi-allelic Panels

- **Haplotypes**: A haplotype refers to a set of genetic variants (alleles) that are inherited together from a single parent.
- **Variant Sites (w)**: Genomic locations where sequences vary across a population are called variant sites.
- **Bi-allelic Panels**: In most genomic studies, these sites are "bi-allelic," meaning they only exhibit two possible states, typically represented as 0 or 1.
- **The Matrix Representation**: Large-scale genomic data is often stored as an $h \times w$ binary matrix, where:

Haplotype	Site 1	Site 2	Site 3	Site 4	Site 5
S_1	0	1	0	1	1
S_2	0	0	1	1	0
S_3	0	0	1	1	0
S_4	0	1	1	1	1
S_5	1	0	0	0	1

A Typical Application: Positional Search

- **Positional Search**: Given a query pattern P and a specific variant site j, find the **longest prefix** $P[1..m']$ that appears in the population starting at site j.

$P[1..m]$		**0**	**1**	**1**	0
Haplotype	**Site 1**	**Site 2**	**Site 3**	**Site 4**	**Site 5**
S_1	0	1	0	1	1
S_2	0	0	1	1	0
S_3	0	0	1	1	0
S_4	0	1	1	1	1
S_5	1	0	0	0	1

Table: Alignment of Pattern P starting at Site 2 of Matrix M.

A Typical Application: Positional Search

- **Positional Search**: Given a query pattern P and a specific variant site j, find the **longest prefix** $P[1..m']$ that appears in the population starting at site j.
- **Identification**: Upon finding the match, either identify a representative haplotype or enumerate all haplotypes in the panel that contain the sequence.

$P[1..m]$		**0**	**1**	**1**	0
Haplotype	**Site 1**	**Site 2**	**Site 3**	**Site 4**	**Site 5**
S_1	0	1	0	1	1
S_2	0	0	1	1	0
S_3	0	0	1	1	0
S_4	0	1	1	1	1
S_5	1	0	0	0	1

Table: Alignment of Pattern P starting at Site 2 of Matrix M.

A Typical Application: Positional Search

Motivation

This query serves as the core building block for complex genomic tasks, such as computing **matching statistics** and identifying **Set-Maximal Exact Matches (SMEMs)**.

A Typical Application: Positional Search

Motivation

This query serves as the core building block for complex genomic tasks, such as computing **matching statistics** and identifying **Set-Maximal Exact Matches (SMEMs)**.

To **facilitate** these queries at scale, the **Positional Burrows–Wheeler Transform (PBWT)** is employed, enabling high-speed prefix matching across massive populations.

Positional Burrows–Wheeler Transform

- **Clustering Identical Alleles**: At any variant site j, the PBWT sorts haplotypes based on the co-lexicographic order of their sequences from site 1 through site $j-1$.

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

co-lex		
0	0	1
1	0	1
1	0	1
1	0	0
0	1	1

PBWT_3

PA	
	2
	1
...	4
	5
	3

PA_3

Positional Burrows–Wheeler Transform

- **Clustering Identical Alleles**: At any variant site j, the PBWT sorts haplotypes based on the co-lexicographic order of their sequences from site 1 through site $j-1$.
- **The Prefix Array (PA)**: In each column j, the PA is a permutation of $\{1, \ldots, h\}$ that records the co-lexicographical order of the haplotypes.

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

co-lex

0	0	1
1	0	1
1	0	1
1	0	0
0	1	1

PBWT_3

PA	
	2
	1
...	4
	5
	3

PA_3

Positional Burrows–Wheeler Transform

- **Creating "Runs"**: This sorting places haplotypes with similar prefixes in adjacent rows, causing identical alleles to cluster together.

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

Positional Burrows–Wheeler Transform

- **Creating "Runs"**: This sorting places haplotypes with similar prefixes in adjacent rows, causing identical alleles to cluster together.
- **Defining a Run**: A maximal contiguous block of identical symbols in a PBWT column is called a "run".

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

Positional Burrows–Wheeler Transform

- μ-PBWT **Compression**: By storing the starting positions, the lengths and values of these runs rather than every individual allele, the μ-PBWT variant compresses massive populations into $O(\tilde{r})$ space, where $\tilde{r}$ is the total number of runs.

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT	1	2	3	4	5
	1	0	1	0	1
	0	1	1	1	1
	0	0	1	1	0
	1	0	0	1	0
	1	0	1	0	1

μ-PBWT	1	2	3	4	5
	(1,1)	(1,0)	(1,1)	(1,0)	(1,1)
	(2,0)	(2,1)	(4,0)	(2,1)	(3,0)
	(4,1)	(3,0)	(5,1)	(5,0)	(5,1)

Navigating the Panel: Fore and Back Stepping

- **Fore Query (Forward Stepping)**
  - **Question:** If a haplotype is in row i at site j, what is its row index at site $j+1$?

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

Navigating the Panel: Fore and Back Stepping

- **Fore Query (Forward Stepping)**
  - **Question:** If a haplotype is in row i at site j, what is its row index at site $j+1$?

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

- **Back Query (Backward Stepping)**
  - **Question:** Given a row index at site $j+1$, what was the rank of that same haplotype at the previous site j?

Navigating the Panel: Fore and Back Stepping

- **Fore Query (Forward Stepping)**
  - **Question:** If a haplotype is in row i at site j, what is its row index at site $j+1$?

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

- **Back Query (Backward Stepping)**
  - **Question:** Given a row index at site $j+1$, what was the rank of that same haplotype at the previous site j?

Connection to Text Indexing

Both operations are conceptually similar to the **backward query** (LF-mapping) over the classic **Burrows–Wheeler Transform (BWT)**.

Related Works and Motivation

- **Uncompressed PBWT**: Traditional structures achieve **constant-time** `fore` and `back` stepping but require $O(\mathtt{h} \cdot \mathtt{w})$ words, which is prohibitive for massive datasets.

Related Works and Motivation

- **Uncompressed PBWT**: Traditional structures achieve **constant-time** `fore` and `back` stepping but require $O(\mathtt{h} \cdot \mathtt{w})$ words, which is prohibitive for massive datasets.
- **Standard Compressed PBWT**: Existing run-length encoded structures (μ-PBWT) reduce space to $O(\tilde{r})$ words, but prior to this work, each mapping step required $O(\log \log_w \mathtt{h})$ time.

Related Works and Motivation

- **Uncompressed PBWT**: Traditional structures achieve **constant-time** `fore` and `back` stepping but require $O(\mathtt{h} \cdot \mathtt{w})$ words, which is prohibitive for massive datasets.
- **Standard Compressed PBWT**: Existing run-length encoded structures (μ-`PBWT`) reduce space to $O(\tilde{r})$ words, but prior to this work, each mapping step required $O(\log \log_w \mathtt{h})$ time.
- **Move Data Structures**: Recent advancements in text indexing introduced the "move structure," which supports constant-time backward queries over the `BWT` using $O(r)$ words.

Limitation of the Standard Move Structure

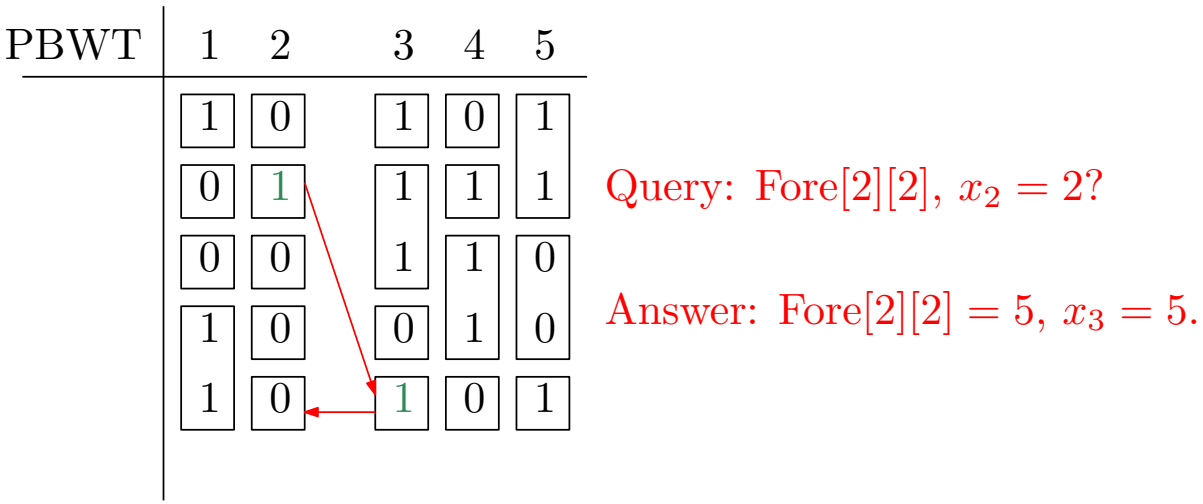
- **Iterative Stepping**: In `PBWT` applications, `fore` and `back` queries are performed sequentially (e.g., stepping from site $j \to j+1 \to j+2$).

Limitation of the Standard Move Structure

- **Iterative Stepping**: In `PBWT` applications, `fore` and `back` queries are performed sequentially (e.g., stepping from site $j \to j+1 \to j+2$).
- Simply applying move structures independently at each site fails because they do not "pass the torch"—the index needed for the next iteration is lost.

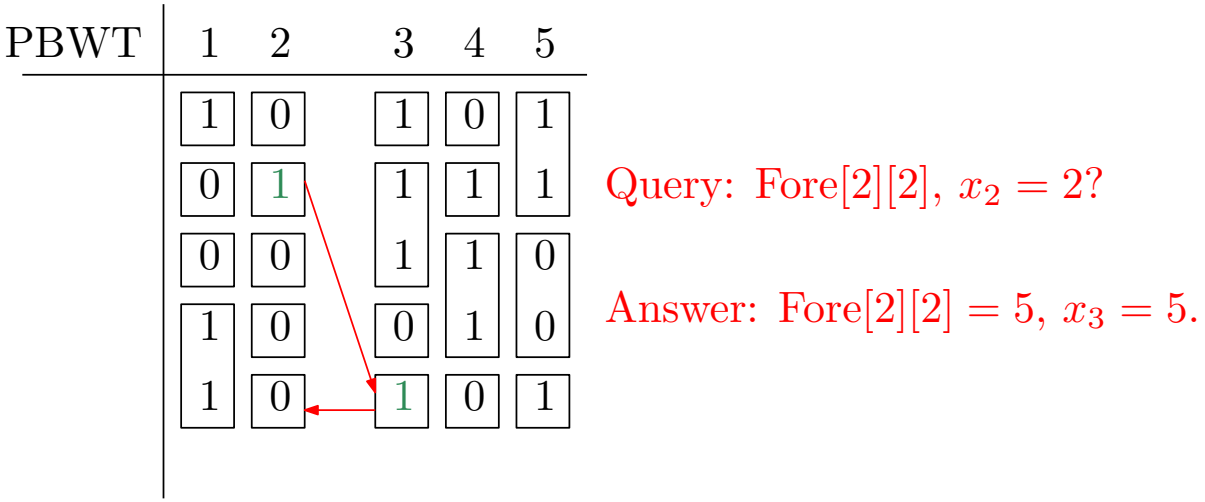
Limitation of the Standard Move Structure

- **Iterative Stepping**: In `PBWT` applications, `fore` and `back` queries are performed sequentially (e.g., stepping from site $j \to j+1 \to j+2$).
- Simply applying move structures independently at each site fails because they do not "pass the torch"—the index needed for the next iteration is lost.
  - The classic move structure returns the sub-index that contains `fore`$[i][j]$ at site j.



Limitation of the Standard Move Structure

- **Iterative Stepping**: In `PBWT` applications, `fore` and `back` queries are performed sequentially (e.g., stepping from site $j \to j+1 \to j+2$).
- Simply applying move structures independently at each site fails because they do not "pass the torch"—the index needed for the next iteration is lost.
  - The classic move structure returns the sub-index that contains `fore`$[i][j]$ at site j.
  - However, to perform the *next* step from $j+1$ to $j+2$, the algorithm requires the **sub-run index** at site $j+1$.



Summary of Our Results

Theoretical Bounds on $\tilde{r}$

Let $\mathtt{h}''$ be the number of adjacent pairs (S_i, S_{i+1}) such that $S_i \neq S_{i+1}$.

- **Bounds:** $\mathtt{h}'' + 1 \leq \tilde{r} \leq \mathtt{w}(\mathtt{h}'' + 1)$.
- **Corollary:** $\tilde{r}$ is at least the number of distinct haplotypes.

Summary of Our Results

Theoretical Bounds on $\tilde{r}$

Let $\mathtt{h}''$ be the number of adjacent pairs (S_i, S_{i+1}) such that $S_i \neq S_{i+1}$.

- **Bounds:** $\mathtt{h}'' + 1 \leq \tilde{r} \leq \mathtt{w}(\mathtt{h}'' + 1)$.
- **Corollary:** $\tilde{r}$ is at least the number of distinct haplotypes.

Our Contribution: Constant-Time Navigation

Inspired by the move structure, we achieve **constant-time** `fore` and `back` stepping while maintaining $O(\tilde{r})$ word space.

Summary of Our Results

Theoretical Bounds on $\tilde{r}$

Let $\mathtt{h}''$ be the number of adjacent pairs (S_i, S_{i+1}) such that $S_i \neq S_{i+1}$.

- **Bounds:** $\mathtt{h}'' + 1 \leq \tilde{r} \leq \mathtt{w}(\mathtt{h}'' + 1)$.
- **Corollary:** $\tilde{r}$ is at least the number of distinct haplotypes.

Our Contribution: Constant-Time Navigation

Inspired by the move structure, we achieve **constant-time** `fore` and `back` stepping while maintaining $O(\tilde{r})$ word space.

- **Applications**: *Prefix Searches* and *Haplotype Retrieval*

Summary of Our Results

Theoretical Bounds on $\tilde{r}$

Let $\mathtt{h}''$ be the number of adjacent pairs (S_i, S_{i+1}) such that $S_i \neq S_{i+1}$.

- **Bounds:** $\mathtt{h}'' + 1 \leq \tilde{r} \leq \mathtt{w}(\mathtt{h}'' + 1)$.
- **Corollary:** $\tilde{r}$ is at least the number of distinct haplotypes.

Our Contribution: Constant-Time Navigation

Inspired by the move structure, we achieve **constant-time** `fore` and `back` stepping while maintaining $O(\tilde{r})$ word space.

- **Applications**: *Prefix Searches* and *Haplotype Retrieval*
- **Multi-Allelic Support**: Our solution naturally generalizes to **multi-allelic panels** without additional overhead.

Our Results: High-Level Idea

- **Sub-run Partitioning**: We divide the original runs of the PBWT into smaller **sub-runs**.

Our Results: High-Level Idea

- **Sub-run Partitioning**: We divide the original runs of the PBWT into smaller **sub-runs**.
- **The Three-Overlap Constraint**: Each sub-run is constructed such that it "involves" (overlaps with) **at most three** sub-runs in the adjacent columns.

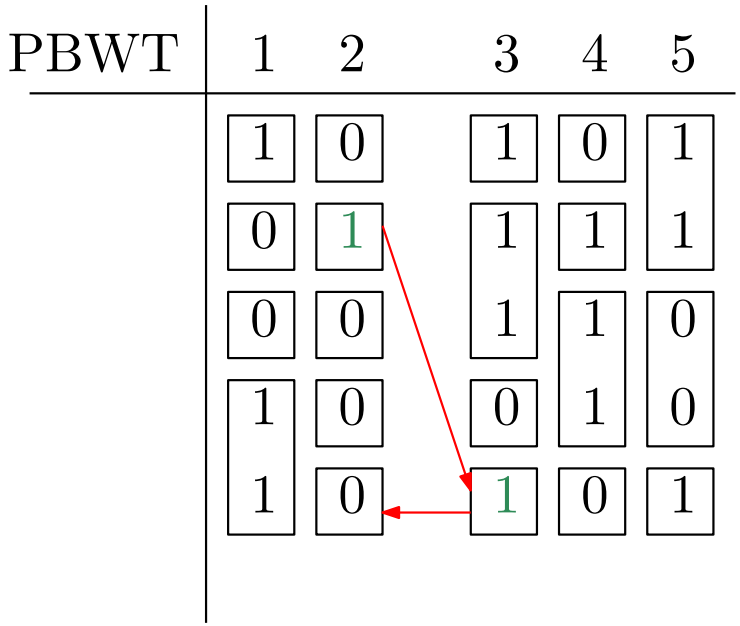
Our Results: High-Level Idea

- **Sub-run Partitioning**: We divide the original runs of the PBWT into smaller **sub-runs**.
- **The Three-Overlap Constraint**: Each sub-run is constructed such that it "involves" (overlaps with) **at most three** sub-runs in the adjacent columns.
- **Tabulated Mapping**: We use a lookup table to store this involvement information.

Our Results: High-Level Idea

- **Sub-run Partitioning**: We divide the original runs of the PBWT into smaller **sub-runs**.
- **The Three-Overlap Constraint**: Each sub-run is constructed such that it "involves" (overlaps with) **at most three** sub-runs in the adjacent columns.
- **Tabulated Mapping**: We use a lookup table to store this involvement information.
- **Constant-Time Navigation**:
  - Given (i, j) and the index x of the sub-run containing entry (i, j), we compute $\texttt{fore}(i, j)$ and the sub-run index at site $j + 1$ in **constant time**.
  - Given (i, j) and the same index x, we compute $\texttt{back}(i, j)$ and the sub-run index at site $j - 1$ in **constant time**.

Our Results: High-Level Idea



Query: Fore[2][2], $x_2 = 2$?

Old Answer: Fore[2][2] $= 5$, $x_3 = 5$.

New Answer: Fore[2][2] $= 5$, $x_3 = 4$.

The Three-Overlap Constraint

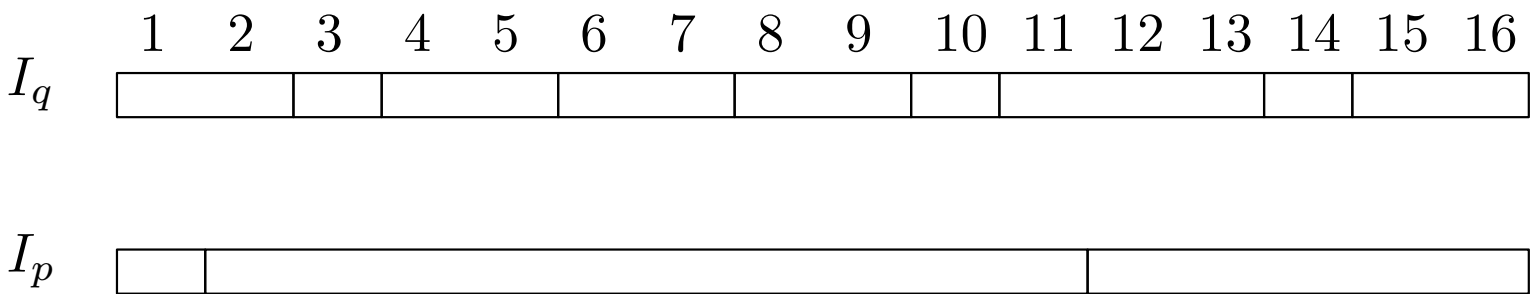
Definition

A partition I_p satisfies the **3-overlap constraint** with respect to I_q if every interval in I_p overlaps at most three intervals in I_q.

The Three-Overlap Constraint

Definition

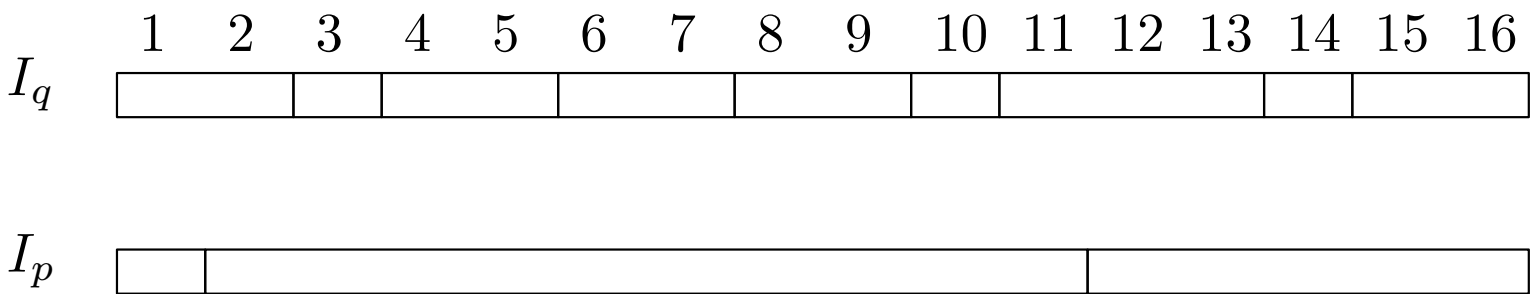
A partition I_p satisfies the **3-overlap constraint** with respect to I_q if every interval in I_p overlaps at most three intervals in I_q.



The Three-Overlap Constraint

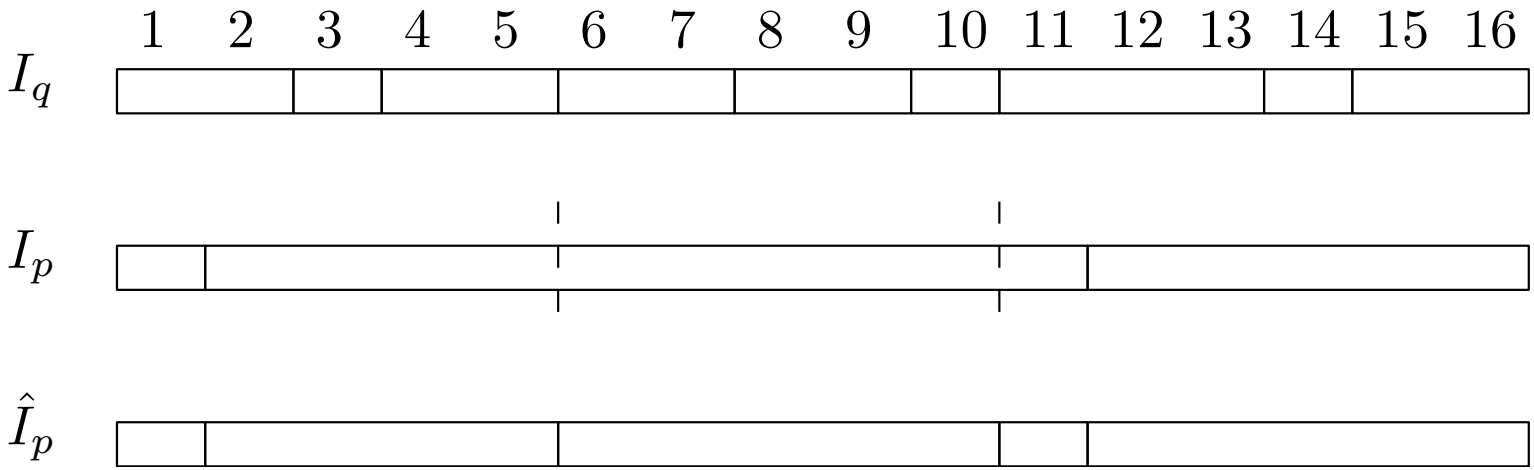
Definition

A partition I_p satisfies the **3-overlap constraint** with respect to I_q if every interval in I_p overlaps at most three intervals in I_q.



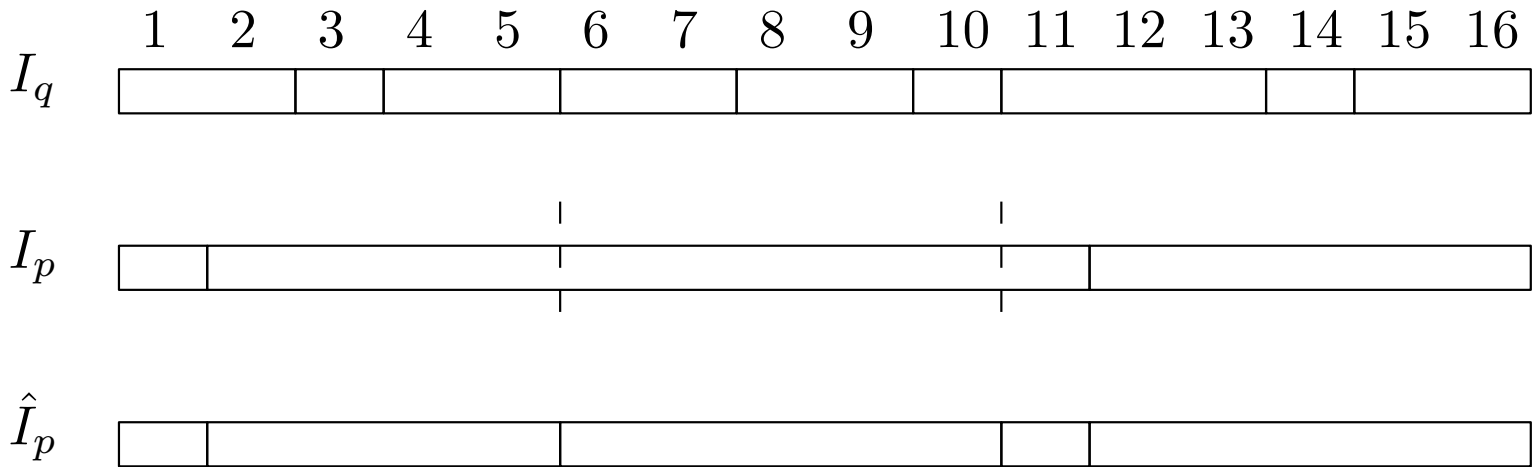
The 3-overlap constraint for I_p is violated.

The Three-Overlap Constraint



- **Normalization Algorithm**: We can transform any partition I_p into a new partition $\hat{I}_p$ that satisfies the constraint in $O(|I_p| + |I_q|)$ time.

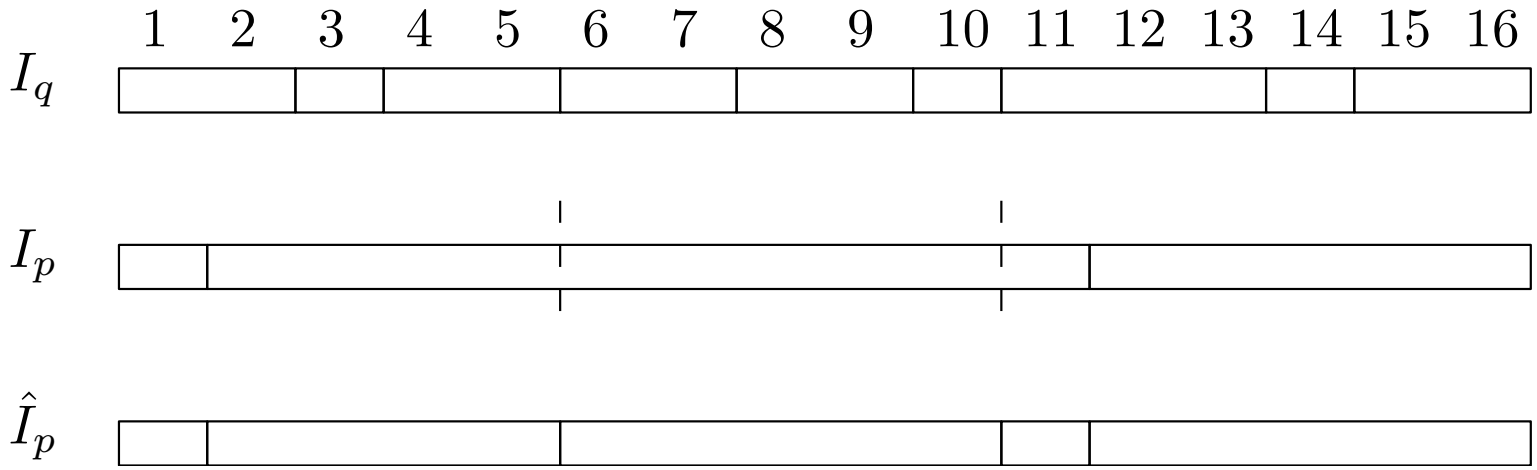
The Three-Overlap Constraint



- **Normalization Algorithm**: We can transform any partition I_p into a new partition $\hat{I}_p$ that satisfies the constraint in $O(|I_p| + |I_q|)$ time.
- **Size Bound**: The number of the sub-intervals in $\hat{I}_p$ is bounded by:

$$|\hat{I}_p| \leq |I_p| + \left\lfloor \frac{|I_q|}{2} \right\rfloor$$

The Three-Overlap Constraint



- **Normalization Algorithm**: We can transform any partition I_p into a new partition $\hat{I}_p$ that satisfies the constraint in $O(|I_p| + |I_q|)$ time.
- **Size Bound**: The number of the sub-intervals in $\hat{I}_p$ is bounded by:

$$|\hat{I}_p| \leq |I_p| + \left\lfloor \frac{|I_q|}{2} \right\rfloor$$

- **Implication**: This bound ensures that while we increase the number of intervals to satisfy the constraint, the total space remains linear in terms of the number of runs $O(\tilde{r})$.

Constructing Sub-Runs $\texttt{SubIB}'s$: Defining the Bijection

For each column j, we define a bijection $\texttt{foreL}_j$ to construct the list of sub-runs $\texttt{SubIB}_{j+1}$ required for constant-time `back` queries.

Constructing Sub-Runs SubIB′s: Defining the Bijection

- **The Mapping Function**: Given a set of intervals $L = \{[b_\tau, e_\tau]\}_{1 \le \tau \le |L|}$ at site j, the bijection $\texttt{foreL}_j(L)$ maps them to site $j+1$: $\texttt{foreL}_j(L) = \{[\texttt{fore}(b_\tau, j), \texttt{fore}(e_\tau, j)] \mid 1 \le \tau \le |L|\}$

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

$\text{Fore}_3(\{[1,3], [4, 4], [5, 5]\}) = \{[1,1],[2,4],[5, 5]\}$

Constructing Sub-Runs SubIB′s: Defining the Bijection

- **The Mapping Function**: Given a set of intervals $L = \{[b_\tau, e_\tau]\}_{1 \le \tau \le |L|}$ at site j, the bijection $\texttt{foreL}_j(L)$ maps them to site $j+1$: $\texttt{foreL}_j(L) = \{[\texttt{fore}(b_\tau, j), \texttt{fore}(e_\tau, j)] \mid 1 \le \tau \le |L|\}$
- **Ordered Output**: sorted in increasingly order by left endpoints.

PBWT				
1	0	1	0	1
0	1	1	1	1
0	0	1	1	0
1	0	0	1	0
1	0	1	0	1

PA				
1	2	2	5	5
2	3	1	2	3
3	1	4	1	2
4	4	5	4	1
5	5	3	3	4

$\text{Fore}_3(\{[1,3], [4, 4], [5, 5]\}) = \{[1,1], [2,4], [5, 5]\}$

Constructing Sub-Runs SubIB: The Algorithm

The list of sub-runs $\texttt{SubIB}_j$ is constructed inductively for each column $j = 1 \ldots \texttt{w}$:

Constructing Sub-Runs $\texttt{SubIB}$: The Algorithm

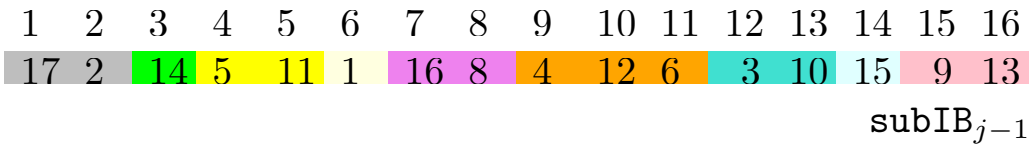
The list of sub-runs $\texttt{SubIB}_j$ is constructed inductively for each column $j = 1 \ldots \texttt{w}$:

- **Base Case ($j = 1$)**: Set $\texttt{SubIB}_1 = \texttt{intervals}_1$ (the original runs of the first column).

Constructing Sub-Runs SubIB: The Algorithm

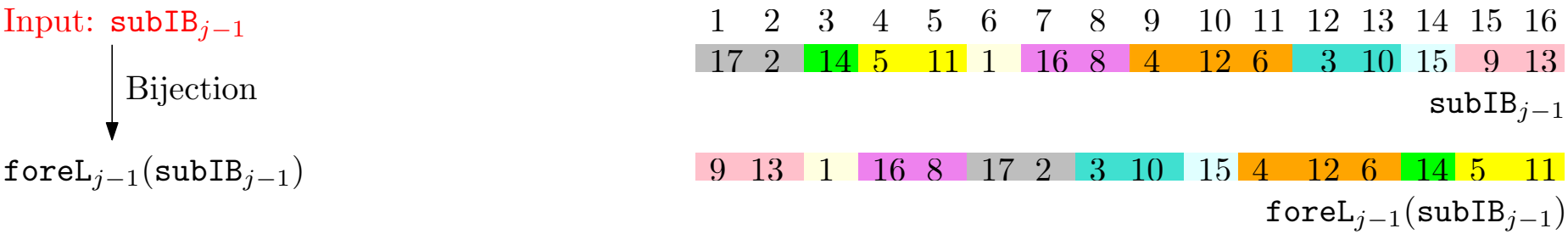
- **Inductive Step ($j > 1$)**: To construct $\texttt{SubIB}_j$, we reconcile the list $\texttt{intervals}_j$ of the runs at column j with $\texttt{SubIB}_{j-1}$:

Input: $\texttt{subIB}_{j-1}$



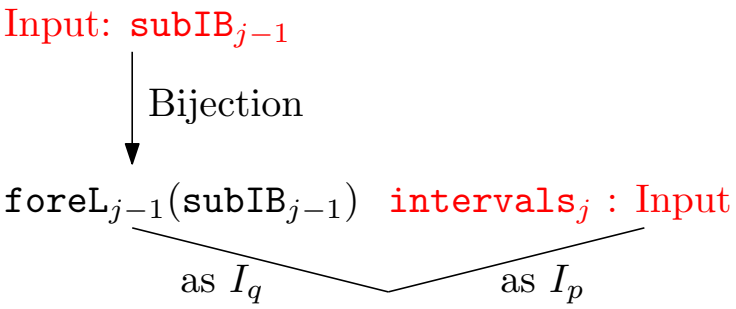
Constructing Sub-Runs SubIB: The Algorithm

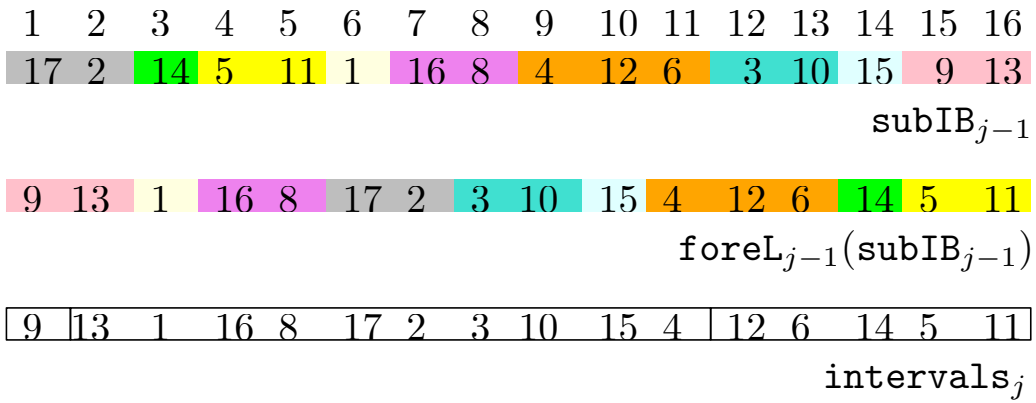
- **Inductive Step** ($j > 1$): To construct $\texttt{SubIB}_j$, we reconcile the list $\texttt{intervals}_j$ of the runs at column j with $\texttt{SubIB}_{j-1}$:
  1. **Map**: Project the previous sub-runs forward: $I_q = \texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$.



Constructing Sub-Runs SubIB: The Algorithm

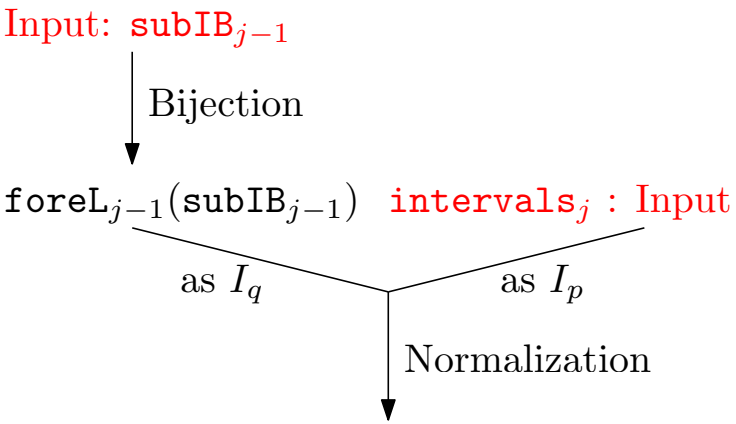
- **Inductive Step** ($j > 1$): To construct $\mathtt{SubIB}_j$, we reconcile the list $\mathtt{intervals}_j$ of the runs at column j with $\mathtt{SubIB}_{j-1}$:
  1. **Map**: Project the previous sub-runs forward: $I_q = \mathtt{foreL}_{j-1}(\mathtt{SubIB}_{j-1})$.

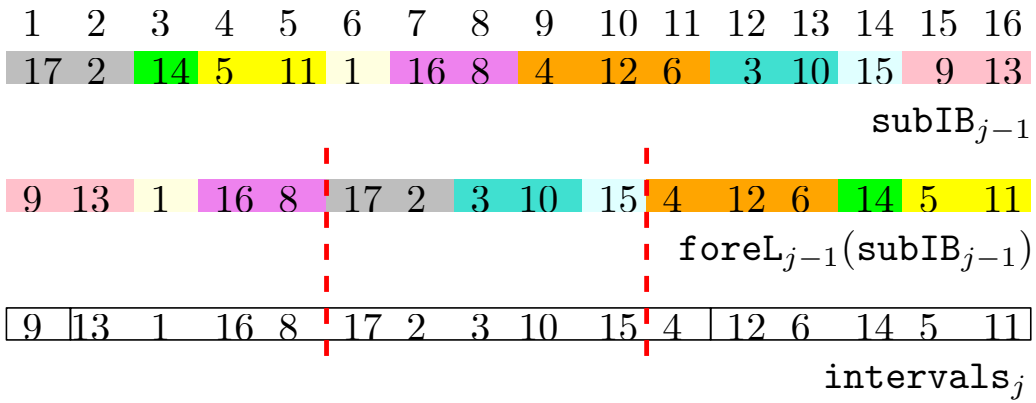




Constructing Sub-Runs SubIB: The Algorithm

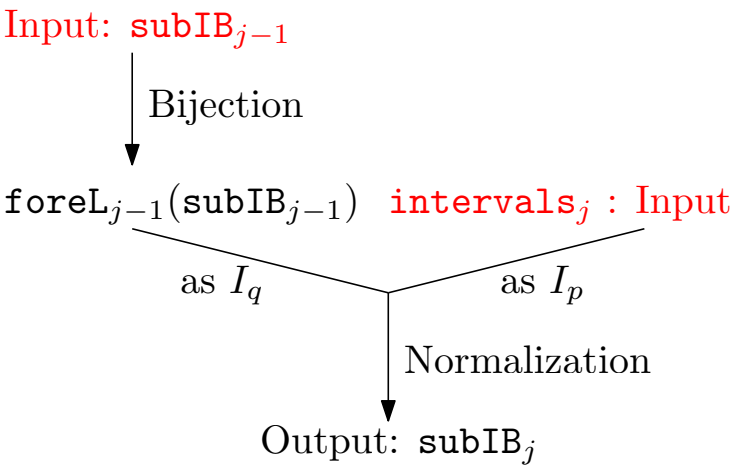
- **Inductive Step** ($j > 1$): To construct $\mathtt{SubIB}_j$, we reconcile the list $\mathtt{intervals}_j$ of the runs at column j with $\mathtt{SubIB}_{j-1}$:
  1. **Map**: Project the previous sub-runs forward: $I_q = \mathtt{foreL}_{j-1}(\mathtt{SubIB}_{j-1})$.
  2. **Normalize**: Apply $\mathtt{normalization}(I_p, I_q)$ where $I_p = \mathtt{intervals}_j$.

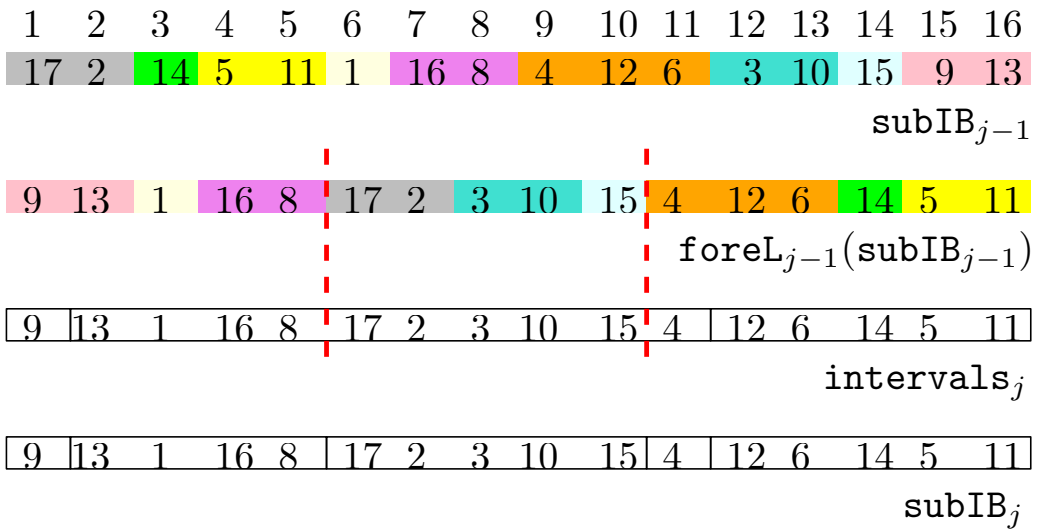




Constructing Sub-Runs SubIB: The Algorithm

- **Inductive Step** ($j > 1$): To construct $\mathtt{SubIB}_j$, we reconcile the list $\mathtt{intervals}_j$ of the runs at column j with $\mathtt{SubIB}_{j-1}$:
  1. **Map**: Project the previous sub-runs forward: $I_q = \mathtt{foreL}_{j-1}(\mathtt{SubIB}_{j-1})$.
  2. **Normalize**: Apply $\mathtt{normalization}(I_p, I_q)$ where $I_p = \mathtt{intervals}_j$.
  3. **Assign**: The resulting partition becomes $\mathtt{SubIB}_j$.

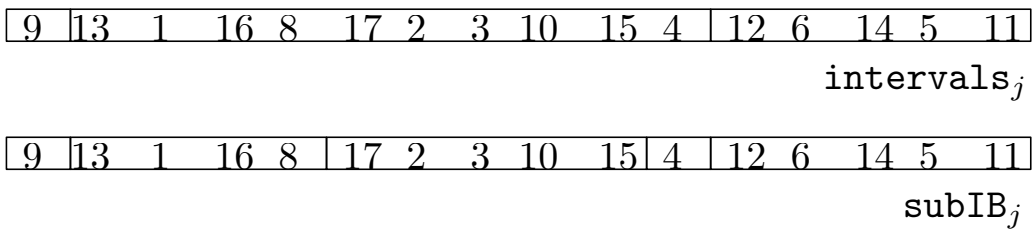




Constructing Sub-Runs SubIB: The Algorithm

Key Properties of the Construction

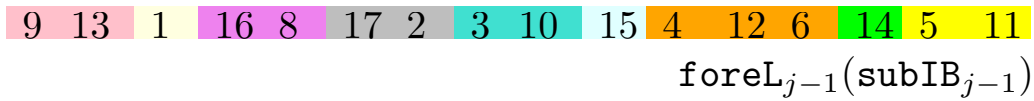
- Every interval in $\texttt{SubIB}_j$ is a **sub-interval** of an interval in $\texttt{intervals}_j$ (and thus corresponds to a **sub-run** at column j).



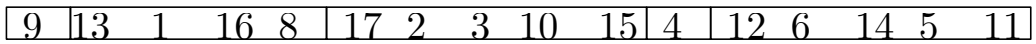
Constructing Sub-Runs SubIB: The Algorithm

Key Properties of the Construction

- Every interval in $\texttt{SubIB}_j$ is a **sub-interval** of an interval in $\texttt{intervals}_j$ (and thus corresponds to a **sub-run** at column j).
- Every interval in $\texttt{SubIB}_j$ overlaps **at most three** intervals in $\texttt{foreL}_{j-1}(\texttt{SubIB}_{j-1})$.



$\texttt{foreL}_{j-1}(\texttt{subIB}_{j-1})$



$\texttt{subIB}_j$

Space Complexity: Bounding the Number of Sub-Runs

Lemma

The total number of sub-runs is linearly bounded by $2\tilde{r}$.

Space Complexity: Bounding the Number of Sub-Runs

Lemma

The total number of sub-runs is linearly bounded by $2\tilde{r}$.

- **Recursive Step**: From the Normalization Lemma, we have:

$$|\mathtt{SubIB}_j| \leq r_j + \frac{|\mathtt{SubIB}_{j-1}|}{2}$$

Space Complexity: Bounding the Number of Sub-Runs

Lemma

The total number of sub-runs is linearly bounded by $2\tilde{r}$.

- **Recursive Step**: From the Normalization Lemma, we have:

$$|\mathtt{SubIB}_j| \leq r_j + \frac{|\mathtt{SubIB}_{j-1}|}{2}$$

- **Geometric Expansion**: Solving this recursion reveals a geometric sequence:

$$|\mathtt{SubIB}_j| \leq \sum_{1 \leq \tau \leq j} \frac{r_\tau}{2^{j-\tau}} = r_j + \frac{r_{j-1}}{2} + \frac{r_{j-2}}{4} + \dots$$

Space Complexity: Bounding the Number of Sub-Runs

Lemma

The total number of sub-runs is linearly bounded by $2\tilde{r}$.

- **Recursive Step**: From the Normalization Lemma, we have:

$$|\mathtt{SubIB}_j| \leq r_j + \frac{|\mathtt{SubIB}_{j-1}|}{2}$$

- **Geometric Expansion**: Solving this recursion reveals a geometric sequence:

$$|\mathtt{SubIB}_j| \leq \sum_{1 \leq \tau \leq j} \frac{r_\tau}{2^{j-\tau}} = r_j + \frac{r_{j-1}}{2} + \frac{r_{j-2}}{4} + \dots$$

- **Total Summation**: By summing across all columns j:

$$\sum_{1 \leq j \leq \mathtt{w}} |\mathtt{SubIB}_j| \leq \sum_{1 \leq j \leq \mathtt{w}} \sum_{1 \leq \tau \leq j} \frac{r_\tau}{2^{j-\tau}} < 2 \sum_{j=1}^{\mathtt{w}} r_j = 2\tilde{r}$$

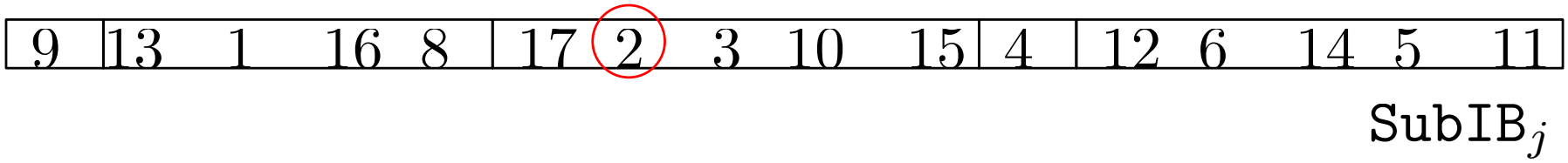
Main Result: Constant-Time Navigation

Theorem

There exists a data structure of $O(\tilde{r})$ words that supports `fore`, `back`, *and character retrieval in $O(1)$* ***time*** *per step, without accessing the original matrix or PBWT.*

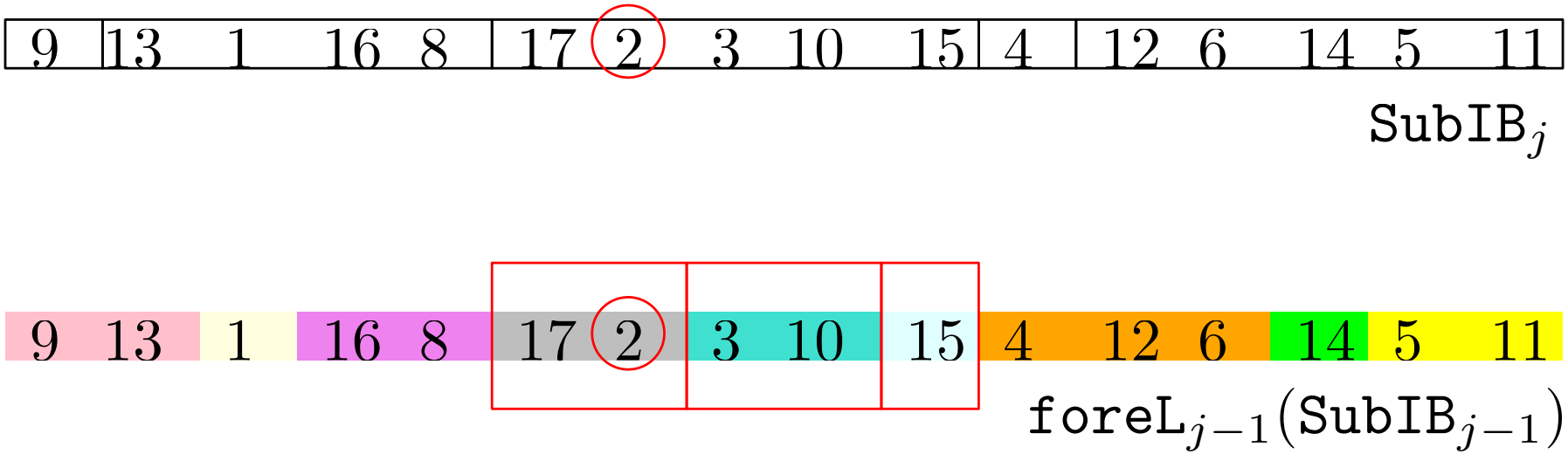
Main Result: Constant-Time Navigation

$\texttt{back}[7][j], x_j = 3$



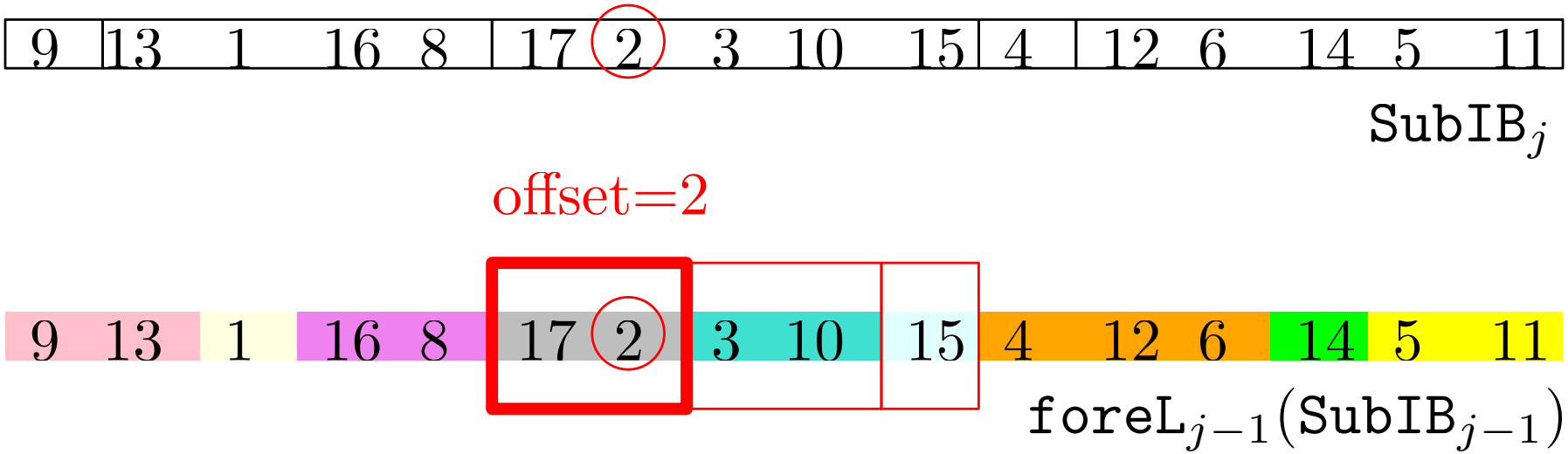
Main Result: Constant-Time Navigation

$\texttt{back}[7][j], x_j = 3$



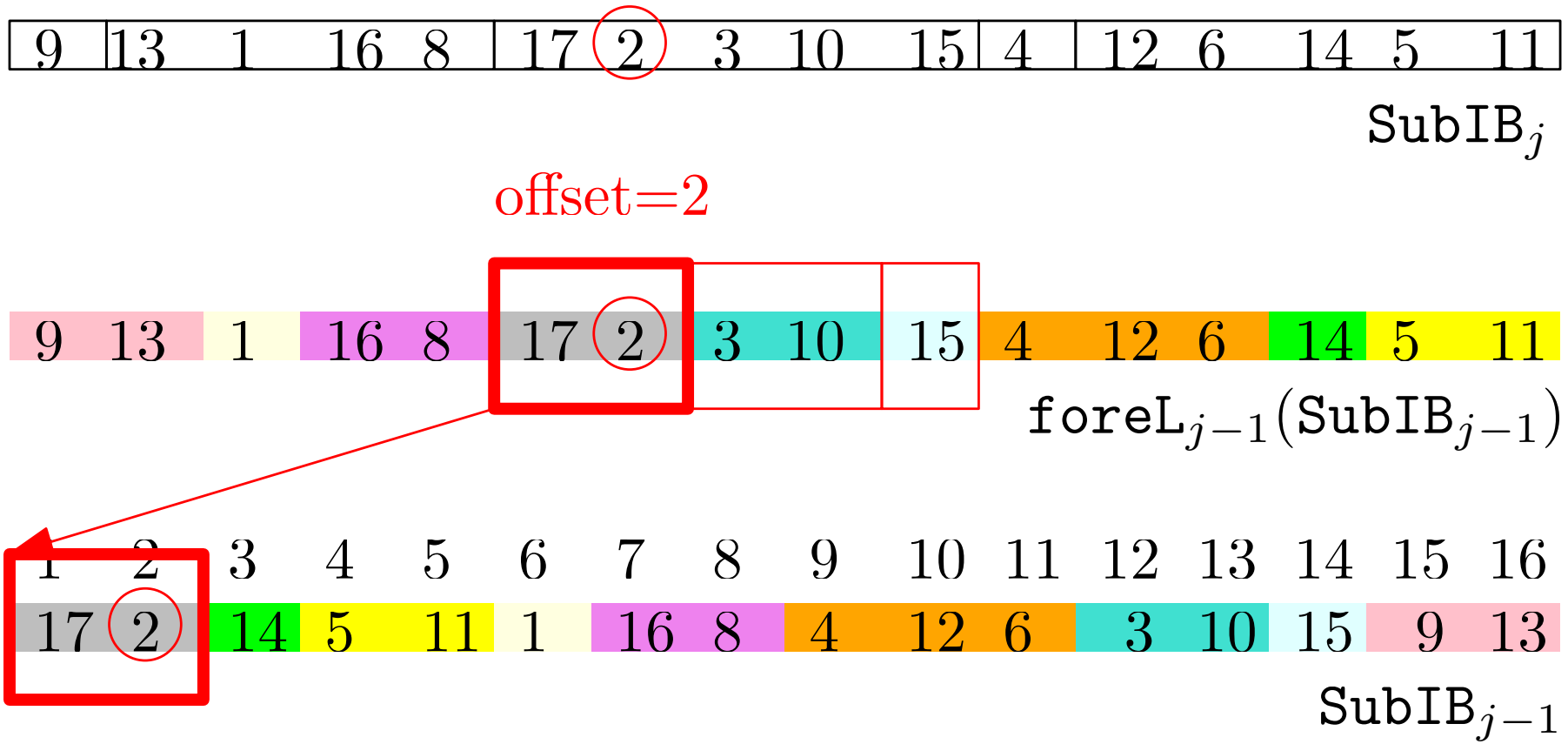
Main Result: Constant-Time Navigation

$\texttt{back}[7][j], x_j = 3$



Main Result: Constant-Time Navigation

$\texttt{back}[7][j] = 2, x_{j-1} = 1$



Application: Haplotype Retrieval

Theorem

Any S_i can be retrieved in $O(w + \log \log_w h)$ time using $O(\tilde{r})$ words.

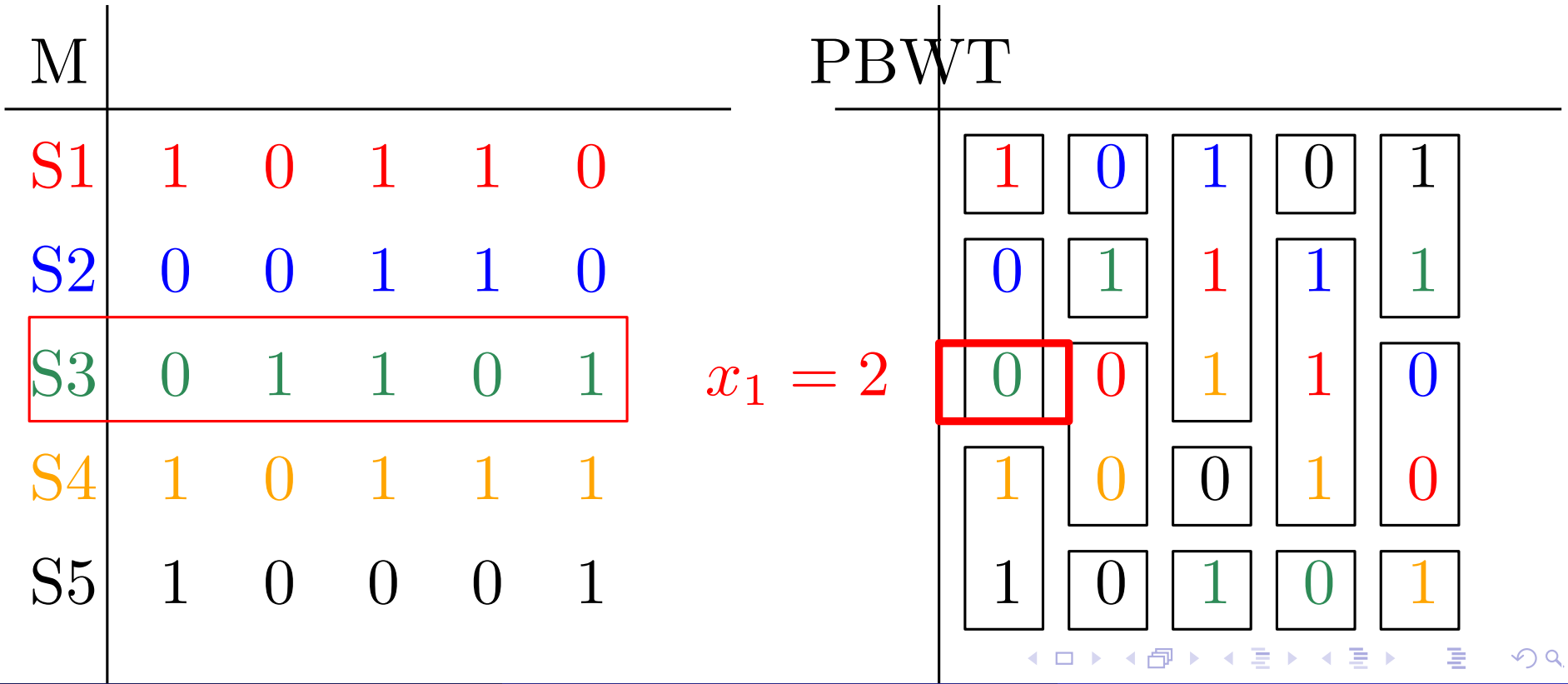
M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

Application: Haplotype Retrieval

Theorem

Any S_i can be retrieved in $O(w + \log\log_w h)$ time using $O(\tilde{r})$ words.

1. Predecessor query to find initial sub-run ($O(\log\log_w \mathtt{h})$).



Application: Haplotype Retrieval

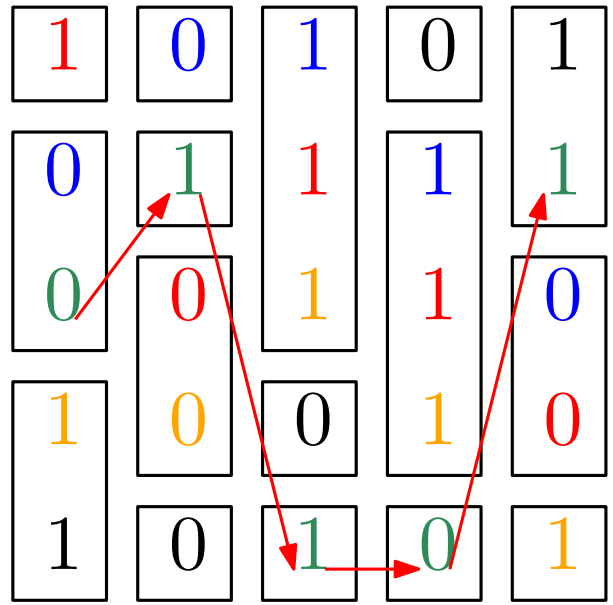
Theorem

Any S_i can be retrieved in $O(w + \log\log_w h)$ time using $O(\tilde{r})$ words.

1. Predecessor query to find initial sub-run ($O(\log\log_w \mathtt{h})$).
2. $O(1)$ forward stepping invoked $\mathtt{w}$ times.

M					
S1	1	0	1	1	0
S2	0	0	1	1	0
S3	0	1	1	0	1
S4	1	0	1	1	1
S5	1	0	0	0	1

PBWT



Application: Prefix Search

- **Goal:** Find longest common prefix $P[1..m']$ and list occ haplotypes.
- **Complexity:** $O(m' \log \log_w \sigma + \mathrm{occ})$ time in $O(\mathtt{h} + \tilde{r})$ space.
- **Variable Lengths:** Handled via terminal symbol #.

	1	2	3	4	5	6	7	8	9
S_1	a	b	a	a	b	a	#		
S_2	a	b	a	a	b	b	#		
S_3	a	b	a	c	#				
S_4	b	c	b	c	a	b	a	#	
S_5	b	c	b	c	a	b	b	#	
S_6	b	c	b	c	b	b	a	#	
S_7	b	c	b	c	b	b	b	a	#

(a) Arbitrary Lengths

	1	2	3	4	5	6	7	8	9
1	a	b	a	a	b	b	#	#	#
2	a	b	a	a	b	b	a	#	
3	a	b	a	c	#	a	b	#	
4	b	c	b	c	a	b	#	a	
5	b	c	b	c	a	b	a		
6	b	c	b	c	b	b	b		
7	b	c	b	c	b				

(b) Extended PBWT

	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	4	1	4	7
2	2	2	2	2	2	5	4	6	
3	3	3	3	3	3	1	5	5	
4	4	4	4	4	4	2	2	7	
5	5	5	5	5	5	6	6		
6	6	6	6	6	6	7	7		
7	7	7	7	7	7				

(c) Extended PA

Figure: Handling varying haplotype lengths.

Conclusions

- **Optimal Time:** Constant-time mapping achieved in run-length compressed PBWT space.
- **Lower and Upper Bounds on $\tilde{r}$:** Established relationship between $\tilde{r}$ and non-identical consecutive haplotype pairs.
- **Practical Utility:** Revisted haplotype retrieval and prefix search applications.
- **Future Work:** Accelerating SMEM and MPSC computations.

Thank You for Your Attention!

Questions?

Younan Gao
University of Milano-Bicocca
younan.gao@unimib.it