UNIVERSITÄT
DES
SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Variation and Conservation in MSAs: Positional Prefix and Divergence Arrays

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Overview

Previously

Building **Multiple Sequence Alignments (MSAs)**

Today

How can we **use** multiple sequence alignments once we have them?

- MSAs have the potential to represent **genetic diversity in a population**
- **Calling variants** for each sample relative to a reference genome **implicitly constructs an MSA** with the reference genome at its center

Indexing a Collection of Sequences

In order to **use** an MSA, we need efficient ways of **searching** it.

Applications

- Searching for common sequences
- Compressing the whole collection / MSA
- Finding intervals that are conserved between many samples

Literature

Richard Durbin. *Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT)*. Bioinformatics, 30(9), pp. 1266-72, 2014.

Tools we have so far

Approach

Use generalized suffix tree, suffix array, or BWT/FM index

Reminder: Generalized Index

1. Concatenate sequences: $S = s_1\$_1 s_2\$_2 \ldots s_n\$_n$
2. Build your favorite index on S

Disadvantages

- Does not take alignment of sequences into account, i.e. no notion of a **common position** or a **common occurence** of a substring.
- Cannot exploit alignment for compression

Turning an MSA into a Haplotype Panel

```
AGATAGTTGCTTTAA-CCTAATGGACAGAT-CTGCTCTGTGACGCCGGT
AGATAGTTACTTTAAGCCTAATGGACAGAT-CTGCTCTGTGACGCCGGT
AGATAGTTACTTTAAGCCTAATGGACAGATGCTGCTCTGTGACGCCGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGT
AGATAGTTGCTTTAAGCCTAATGGACAGATGCTGCTCTGTGAAGCCGGT
```

Turning an MSA into a Haplotype Panel

```
AGATAGTTGCTTTAA-CCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAAGCCTAATGGACAGATGCTGCTCTGTGAAGCCGGGT
```

Turning an MSA into a Haplotype Panel

```
AGATAGTTGCTTTAA-CCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAAGCCTAATGGACAGATGCTGCTCTGTGAAGCCGGGT
```

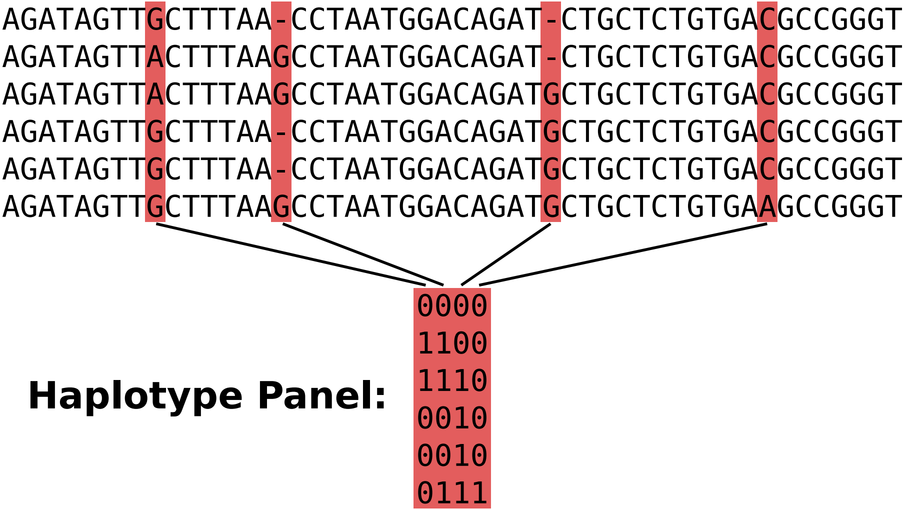
```
********G******-**************-***********C*******
********A******G**************-***********C*******
********A******G**************G***********C*******
********G******-**************G***********C*******
********G******-**************G***********C*******
********G******G**************G***********A*******
```

Turning an MSA into a Haplotype Panel

```
AGATAGTTGCTTTAA-CCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAAGCCTAATGGACAGATGCTGCTCTGTGAAGCCGGGT
```

```
        0      0              0           0
        1      1              0           0
        1      1              1           0
        0      0              1           0
        0      0              1           0
        0      1              1           1
```

Turning an MSA into a Haplotype Panel



```
AGATAGTTGCTTTAA-CCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGAT-CTGCTCTGTGACGCCGGGT
AGATAGTTACTTTAAGCCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAA-CCTAATGGACAGATGCTGCTCTGTGACGCCGGGT
AGATAGTTGCTTTAAGCCTAATGGACAGATGCTGCTCTGTGAAGCCGGGT
```

Haplotype Panel:

```
0000
1100
1110
0010
0010
0111
```

Set of Haplotype Sequences (Haplotype Panel)

Given

Set X of M haplotype sequences over $\Sigma = \{0, 1\}$, each of them with N sites

$$\begin{array}{ccccccccccc}
x_0 & 0 & 0 & 1 & 0 & \ldots & 1 & 1 & 0 & 0 \\
x_1 & 1 & 0 & 1 & 1 & \ldots & 1 & 0 & 1 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_{M-2} & 1 & 0 & 1 & 1 & \ldots & 1 & 1 & 0 & 0 \\
x_{M-1} & 0 & 1 & 0 & 0 & \ldots & 1 & 0 & 1 & 0
\end{array}$$

Note

Here we assume a binary alphabet, but the ideas extend to larger alphabets as well. (See Mäkinen and Norri, 2019, `https://doi.org/10.1016/j.ipl.2019.02.003`).

Reversed Prefix Sorting

Reminder: Suffix Array

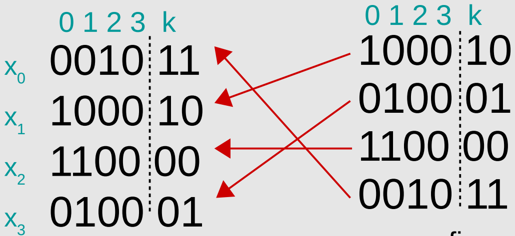
```
    0123456
s = banana$

r   pos[r]   s[pos[r]..]
0   6        $
1   5        a$
2   3        ana$
3   1        anana$
4   0        banana$
5   4        na$
6   2        nana$
```

The **suffix array** contains the starting positions of all suffixes in lexicographic order.

New Concept: Reversed Prefix Sorting

Pick a prefix length k.
Consider the k-prefixes and sort their **reverses** lexicographically.

Example for $k = 4$



	0 1 2 3	k
x_0	0010	11
x_1	1000	10
x_2	1100	00
x_3	0100	01

original set

0 1 2 3	k
1000	10
0100	01
1100	00
0010	11

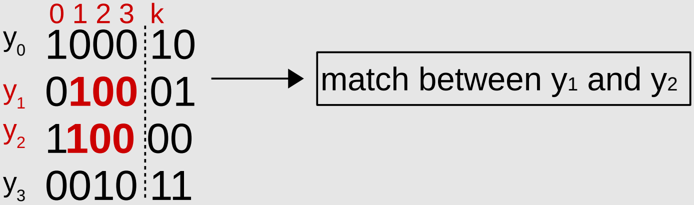
reverse prefix ordering

Reversed Prefix Sorting: Why?

Observations

- **Matches** between two sequences are **adjacent** in the sorting,
- … comparable to the sorted suffixes in the suffix array,
- … but here the matches are **positional/aligned**.

Example



0 1 2 3 k

y_0 1000 10

y_1 0**100** 01

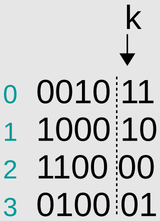
y_2 1**100** 00

y_3 0010 11

match between y_1 and y_2

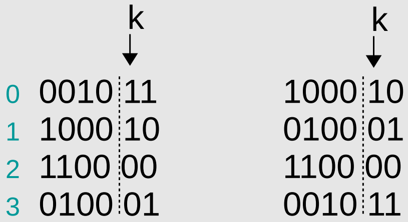
Positional Prefix Array by Example

unsorted:



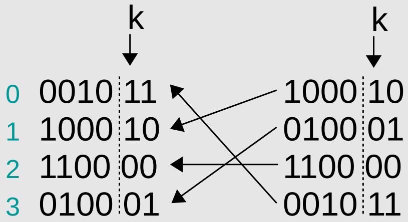
0 0010 11
1 1000 10
2 1100 00
3 0100 01

Positional Prefix Array by Example



	unsorted:	sorted:
	k	k
0	0010 11	1000 10
1	1000 10	0100 01
2	1100 00	1100 00
3	0100 01	0010 11

Positional Prefix Array by Example



unsorted:

k

0 0010 11
1 1000 10
2 1100 00
3 0100 01

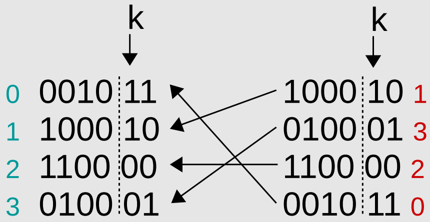
sorted:

k

1000 10
0100 01
1100 00
0010 11

Positional Prefix Array by Example

unsorted: sorted:



k k

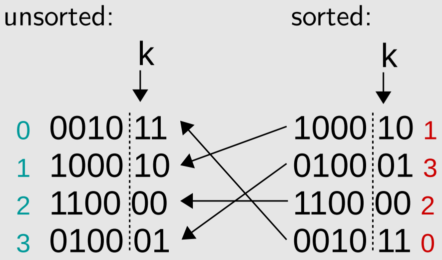
0 0010 11 1000 10 1

1 1000 10 0100 01 3

2 1100 00 1100 00 2

3 0100 01 0010 11 0

Positional Prefix Array by Example



unsorted:

k

0 0010 11

1 1000 10

2 1100 00

3 0100 01

sorted:

k

1000 10 1

0100 01 3

1100 00 2

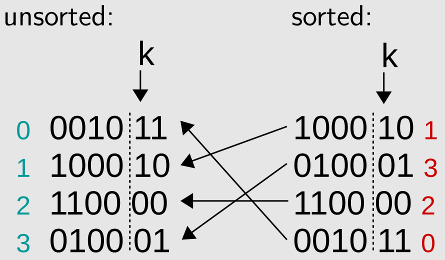
0010 11 0

$a_k[0] = 1,\ a_k[1] = 3$

$a_k[2] = 2,\ a_k[3] = 0$

$$a_k = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 0 \end{pmatrix}$$

Positional Prefix Array by Example



unsorted:

k

0 0010 11
1 1000 10
2 1100 00
3 0100 01

sorted:

k

1000 10 1
0100 01 3
1100 00 2
0010 11 0

$a_k[0] = 1,\ a_k[1] = 3$
$a_k[2] = 2,\ a_k[3] = 0$

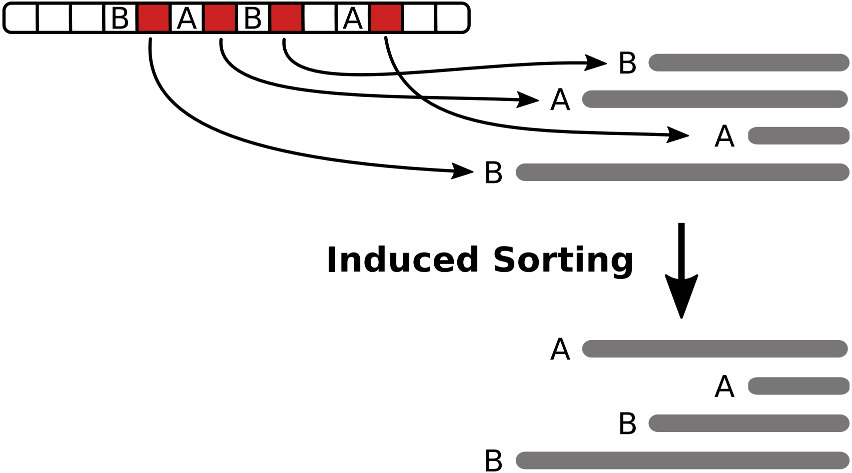
$$a_k = \begin{pmatrix} 1 \\ 3 \\ 2 \\ 0 \end{pmatrix}$$

Definition: Positional prefix array

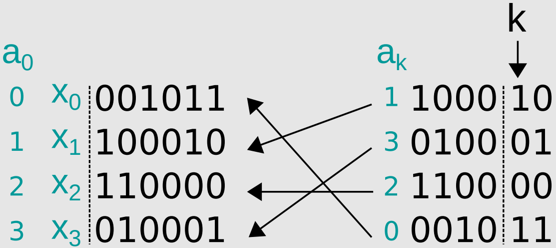
For an input set of strings $x_0, \ldots, x_{M-1} \in \Sigma^N$, the **positional prefix array** a_k for position k lists the sequence indices in lexicographic order with respect to the **reversed prefix** of the length-k prefixes:

$$rev(x_{a_k[0]}[0..k)) \leq rev(x_{a_k[1]}[0..k)) \leq \cdots \leq rev(x_{a_k[M-1]}[0..k))$$

Reminder: Induced Sorting Idea for Suffix Arrays



Building a_{k+1} from a_k

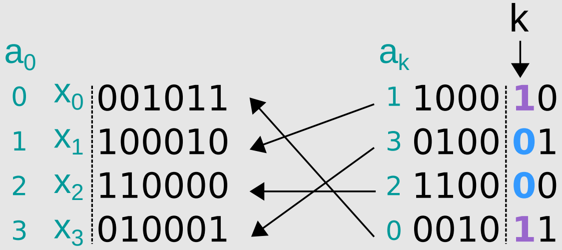


a_0		
0	x_0	001011
1	x_1	100010
2	x_2	110000
3	x_3	010001

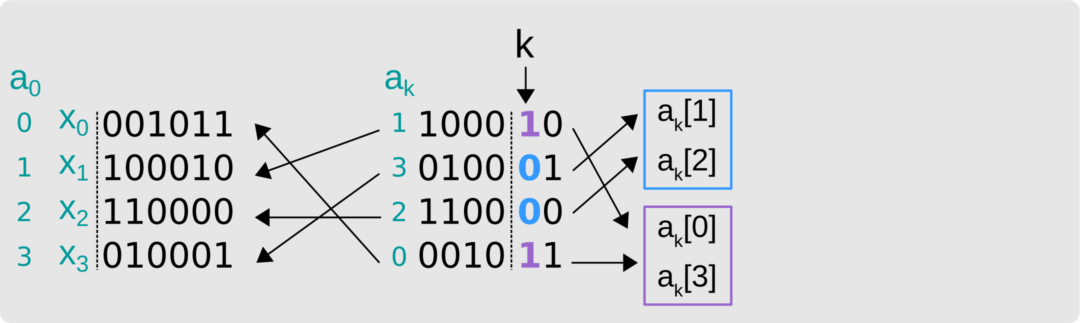
k

a_k		
1	1000	10
3	0100	01
2	1100	00
0	0010	11

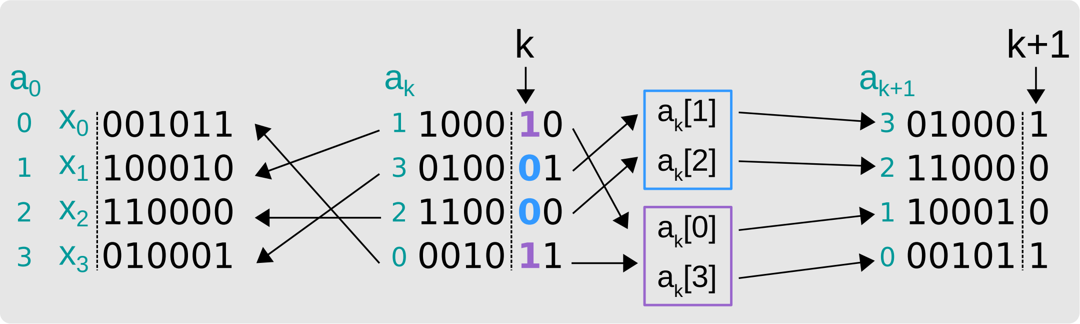
Building a_{k+1} from a_k



Building a_{k+1} from a_k



Building a_{k+1} from a_k



Code: Building a_{k+1} from a_k

```python
def build_reverse_prefix_array_column(X, k, a_k):
    """Take a_k and return a_{k+1}."""
    M = len(X)  # number of strings
    a = [[],[]]  # a[0] and a[1] are both empty lists
    for r in range(M):
        i = a_k[r] # index of string at rank r
        j = int(X[i][k])  # 0 or 1?
        a[j].append(i)  # put i into correct list a[j]
    return a[0] + a[1]  # concatenate lists
```

Code: Complete procedure

```python
def build_reverse_prefix_array(X):
    M = len(X)     # number of strings
    N = len(X[0]) # number of columns
    # column 0 of prefix array: same order as input
    a = [list(range(M))]  # [[0,1,...,M-1]]
    for k in range(N):
        # Build a[k+1] from a[k]
        a_next = build_reverse_prefix_array_column(X, k, a[k])
        a.append(a_next)
    return a
```

The Divergence Array

Reminder: LCP Array

`r`	`pos[r]`	`lcp[r]`	`s[pos[r]..]`
0	6	-1	`$`
1	5	0	`a$`
2	3	1	`ana$`
3	1	3	`anana$`
4	0	0	`banana$`
5	4	0	`na$`
6	2	2	`nana$`
7		-1	

```
    0123456
s = banana$
```

The **longest common prefix (LCP)** array at rank `r` contains
the length of the longest common prefix between the suffixes at ranks `r` and `r-1`.

The Divergence Array

Definition

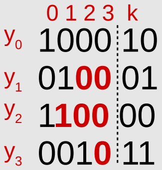
$d_k[i]$: leftmost column j such that $y_i[j, k) = y_{i-1}[j, k)$, where $y_i := x_{a_k[i]}$.

The Divergence Array

Definition

$d_k[i]$: leftmost column j such that $y_i[j,k) = y_{i-1}[j,k)$, where $y_i := x_{a_k[i]}$.

Example for $k = 4$



$$\begin{array}{l|cccc|cc}
 & 0 & 1 & 2 & 3 & k & \\
y_0 & 1 & 0 & 0 & 0 & 1 & 0 \\
y_1 & 0 & 1 & \mathbf{0} & \mathbf{0} & 0 & 1 \\
y_2 & 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 & 0 \\
y_3 & 0 & 0 & 1 & \mathbf{0} & 1 & 1
\end{array}$$

$$d_k[i] = \begin{pmatrix} 4 \\ 2 \\ 1 \\ 3 \end{pmatrix}$$

Note

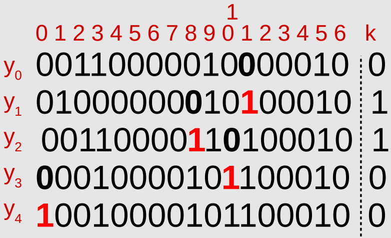
$d_k[0] = k$ by definition, corresponding to an “empty common string”.

Common Reverse Prefix between y_i and y_j

Observation

Start of maximal match between y_i and y_j: $\max\limits_{i<m\leq j} d_k[m]$

Example



	0 1 2 3 4 5 6 7 8 9 1 0 1 2 3 4 5 6	k
y_0	0011000001**0**000010	0
y_1	01000000**0**10**1**00010	1
y_2	00110000**1**1**0**100010	1
y_3	**0**001000010**1**100010	0
y_4	**1**0010000101100010	0

Common Reverse Prefix between y_i and y_j

Observation

Start of maximal match between y_i and y_j: $\max\limits_{i<m\leq j} d_k[m]$

Example

											1							
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	k
y_0	0	0	1	1	0	0	0	0	0	1	0	**0**	0	0	0	1	0	0
y_1	0	1	0	0	0	0	0	0	**0**	1	0	**1**	0	0	0	1	0	1
y_2	0	0	1	1	0	0	0	0	**1**	1	**0**	1	0	0	0	1	0	1
y_3	**0**	0	0	1	0	0	0	0	1	0	**1**	1	0	0	0	1	0	0
y_4	**1**	0	0	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0

$d_k[1] = 12$
$d_k[2] = 9$
$d_k[3] = 11$
$d_k[4] = 1$

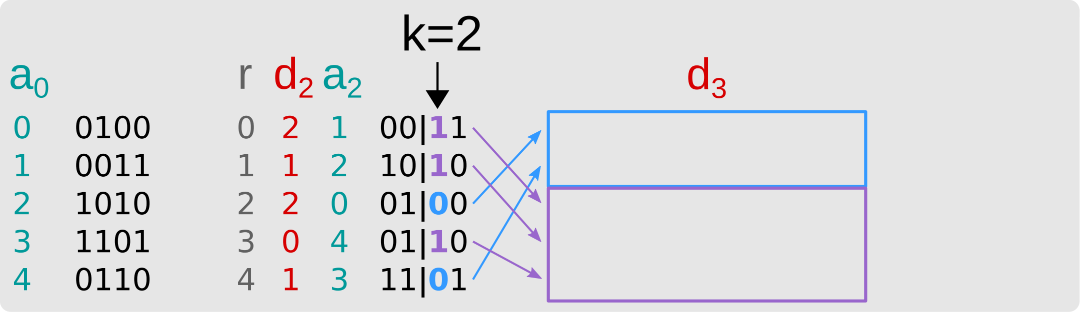
start of maximal match of y_0 and y_4:

$$\max_{0<m\leq 4} d_k[m] = 12$$

Building d_{k+1} from d_k

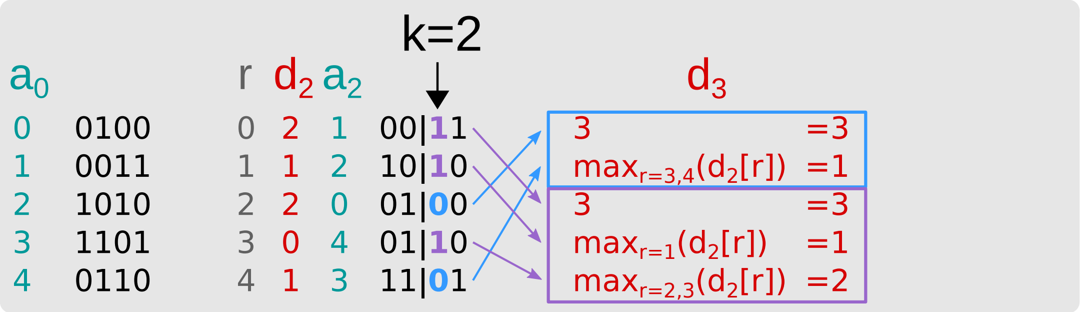
a_0		r	d_2	a_2	k=2
0	0100	0	2	1	00|11
1	0011	1	1	2	10|10
2	1010	2	2	0	01|00
3	1101	3	0	4	01|10
4	0110	4	1	3	11|01

Building d_{k+1} from d_k



a_0		r	d_2	a_2	
0	0100	0	2	1	00|**1**1
1	0011	1	1	2	10|**1**0
2	1010	2	2	0	01|**0**0
3	1101	3	0	4	01|**1**0
4	0110	4	1	3	11|**0**1

Building d_{k+1} from d_k



k=2

a_0		r	d_2	a_2	
0	0100	0	2	1	00|11
1	0011	1	1	2	10|10
2	1010	2	2	0	01|00
3	1101	3	0	4	01|10
4	0110	4	1	3	11|01

d_3

$3 = 3$

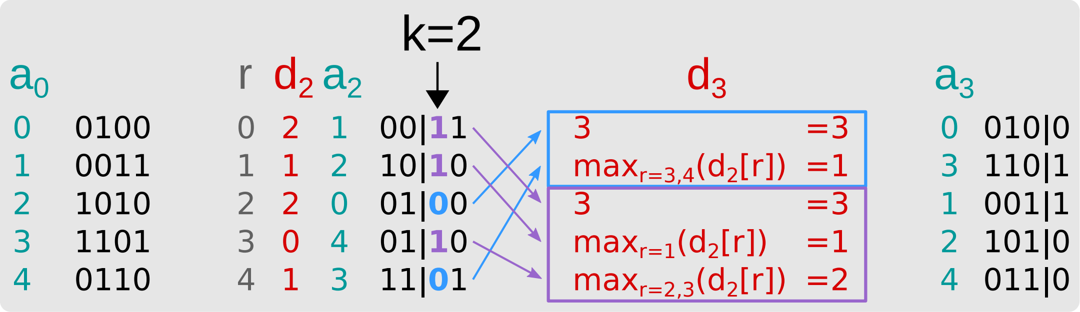
$\max_{r=3,4}(d_2[r]) = 1$

$3 = 3$

$\max_{r=1}(d_2[r]) = 1$

$\max_{r=2,3}(d_2[r]) = 2$

Building d_{k+1} from d_k



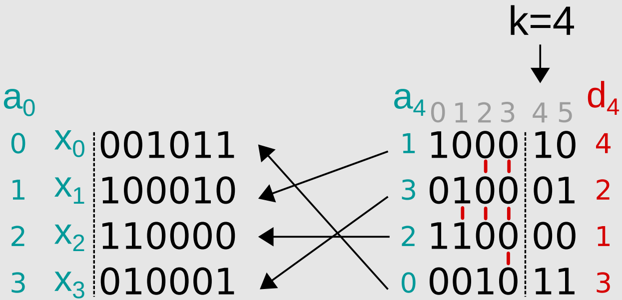
a_0		r	d_2	a_2	k=2	d_3		a_3	
0	0100	0	2	1	00|**1**1	3	=3	0	010|0
1	0011	1	1	2	10|**1**0	$\max_{r=3,4}(d_2[r])$	=1	3	110|1
2	1010	2	2	0	01|**0**0	3	=3	1	001|1
3	1101	3	0	4	01|**1**0	$\max_{r=1}(d_2[r])$	=1	2	101|0
4	0110	4	1	3	11|**0**1	$\max_{r=2,3}(d_2[r])$	=2	4	011|0

Code: Efficiently Building Divergence Arrays

```python
def build_divergence_array(X, a):
    M, N = len(X), len(X[0])  # number of strings, columns
    D = [[0] * M]  # [[0,...,0]]
    for k in range(N):
        # compute d = D[k+1] from D[k]
        d = [ [] for _ in range(2) ]  # two empty lists
        maxima = [k+1] * 2  # [k+1, k+1]
        for r in range(M):
            i = a[k][r]  # index of string at rank r
            maxima = [max(D[k][r], maxima[q]) for q in range(2)]
            j = int(X[i][k])
            d[j].append(maxima[j])
            maxima[j] = 0
        D.append(d[0] + d[1])
    return D
```

Application:
Finding all length-L matches within X

Idea: Finding all length-L matches



- Matches are adjacent in the reverse prefix sorting
- For each column k:
  - Iterate through d_k to find runs of matches of length L

Code: Finding all length-L matches

```python
def length_L_matches(a, d, L):
    M = len(a[0])  # number of strings
    for k, (a_k, d_k) in enumerate(zip(a,d)):
        start = None
        for r in range(M):
            # match between row r-1 and r ending in column k-1?
            if d_k[r] <= k - L:
                if start is None: start = r-1
            else:
                if start is not None:
                    # yield (interval, [row indices]):
                    yield ( (k-L, k),
                      [a_k[r] for r in range(start, r)] )
                    start = None
        # any matches that include the last row?
        if start is not None:
            yield ((k-L,k), [a_k[r] for r in range(start, M)])
```

Running time: Finding all length-L matches

```
for k, (a_k, d_k) in enumerate(zip(a,d)):
  ...
  for r in range(M):
    ...
      if start is not None:
        yield (k-L, k), [a_k[r] for r in range(start, r)]
        ...
  if start is not None:
    yield (k-L,k), [a_k[r] for r in range(start, M)]
```

$O(NM + Z)$, where

- N: number of sites
- M: number of sequences in X
- Z: number of length-L matches (output size)

Summary

- MSA gives rise to **2D matrix** of (today: binary) characters
- Index structures:
  - **Positional prefix array** a_k is related to **suffix arrays**, but with reversed prefixes instead
  - **Divergence array** d_k is related to **longest common prefix arrays**, also with reversed prefixes
- Application: Length-L matches

Possible Exam questions

- How are haplotype panels related to multiple sequence alignments?
- What is a positional prefix array?
- What are commonalities/differences to a suffix array?
- Explain how to build the positional prefix arrays for all columns in $O(MN)$ time.
- How does this algorithm relate to induced sorting?
- Define the divergence array.
- What are commonalities and differences between the divergence array and the LCP array?
- How can the divergence array be constructed efficiently?
- Explain how to find all positional length-L matches in a binary matrix X.