Sparse Suffix Tree Construction in Small Space

Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj
(Technical University of Denmark)
Johannes Fischer, *(Karlsruhe Institute of Technology)*
Tsvi Kopelowitz, *(Weizmann Institute of Science)*
Benjamin Sach *(University of Warwick)*

THE UNIVERSITY OF WARWICK

The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

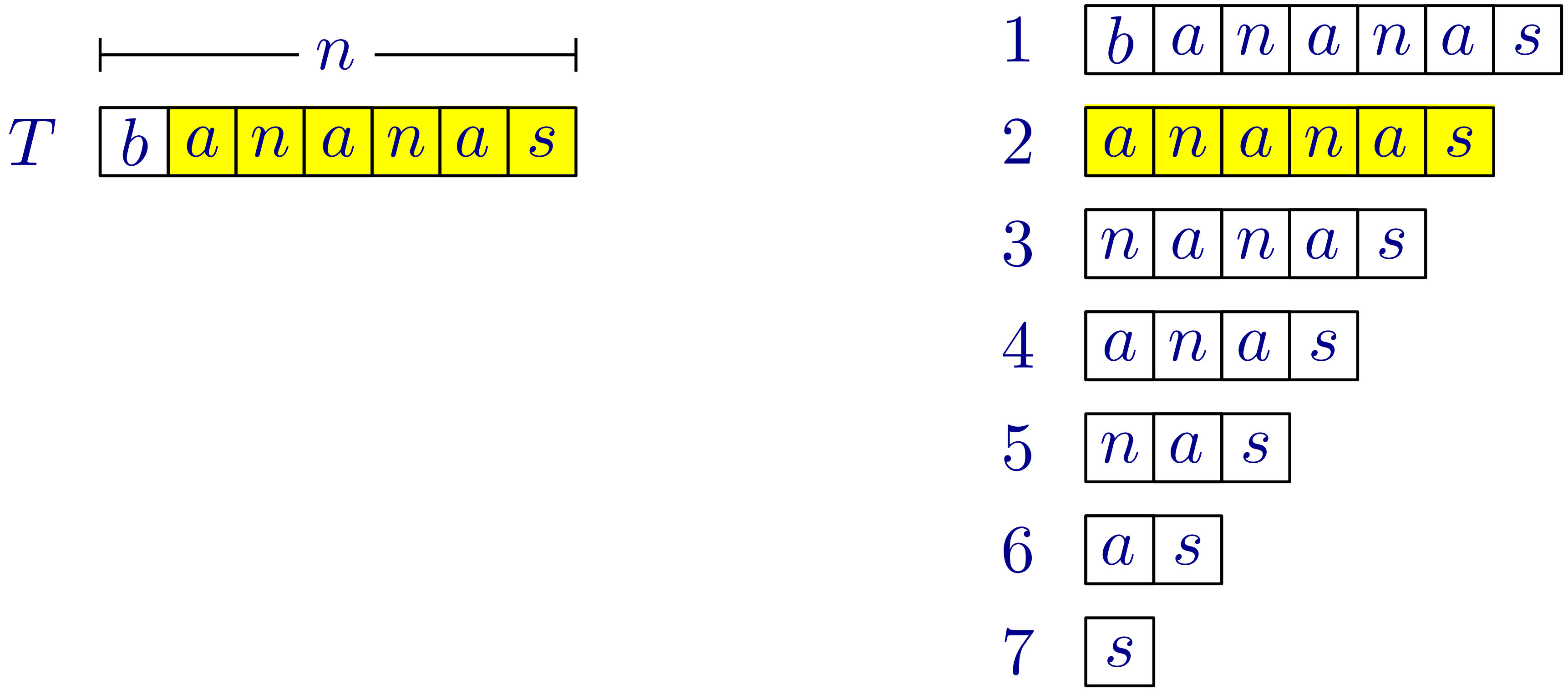
The sparse suffix array (SSA)

$\longleftarrow n \longrightarrow$

T	b	a	n	a	n	a	s

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

The sparse suffix array (SSA)



The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

Sort the suffixes lexicographically

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

Sort the suffixes lexicographically

2	a	n	a	n	a	s	
4	a	n	a	s			
6	a	s					
1	b	a	n	a	n	a	s
3	n	a	n	a	s		
5	n	a	s				
7	s						

The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

Sort the suffixes lexicographically

n

Suffix Array	2	4	6	1	3	5	7

2	a	n	a	n	a	s	
4	a	n	a	s			
6	a	s					
1	b	a	n	a	n	a	s
3	n	a	n	a	s		
5	n	a	s				
7	s						

The sparse suffix array (SSA)

$\vdash\!\!-\!\!-\; n \;-\!\!-\!\!\dashv$

T	b	a	n	a	n	a	s

Sort the suffixes lexicographically

$\vdash\!\!-\!\!-\; n \;-\!\!-\!\!\dashv$

Suffix Array	2	4	6	1	3	5	7

2	a	n	a	n	a	s	
4	a	n	a	s			
6	a	s					
1	b	a	n	a	n	a	s
3	n	a	n	a	s		
5	n	a	s				
7	s						

- Can be built in $O(n)$ time and $O(n)$ *extra space*

The sparse suffix array (SSA)

$\overbrace{\hspace{6em}}^{n}$

T	b	a	n	a	n	a	s

Sort the suffixes lexicographically

$\overbrace{\hspace{6em}}^{n}$

Suffix Array	2	4	6	1	3	5	7

2	a	n	a	n	a	s	
4	a	n	a	s			
6	a	s					
1	b	a	n	a	n	a	s
3	n	a	n	a	s		
5	n	a	s				
7	s						

- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)

n

T	b	a	n	a	n	a	s

2	a	n	a	n	a	s	
4	a	n	a	s			
6	a	s					
1	b	a	n	a	n	a	s
3	n	a	n	a	s		
5	n	a	s				
7	s						

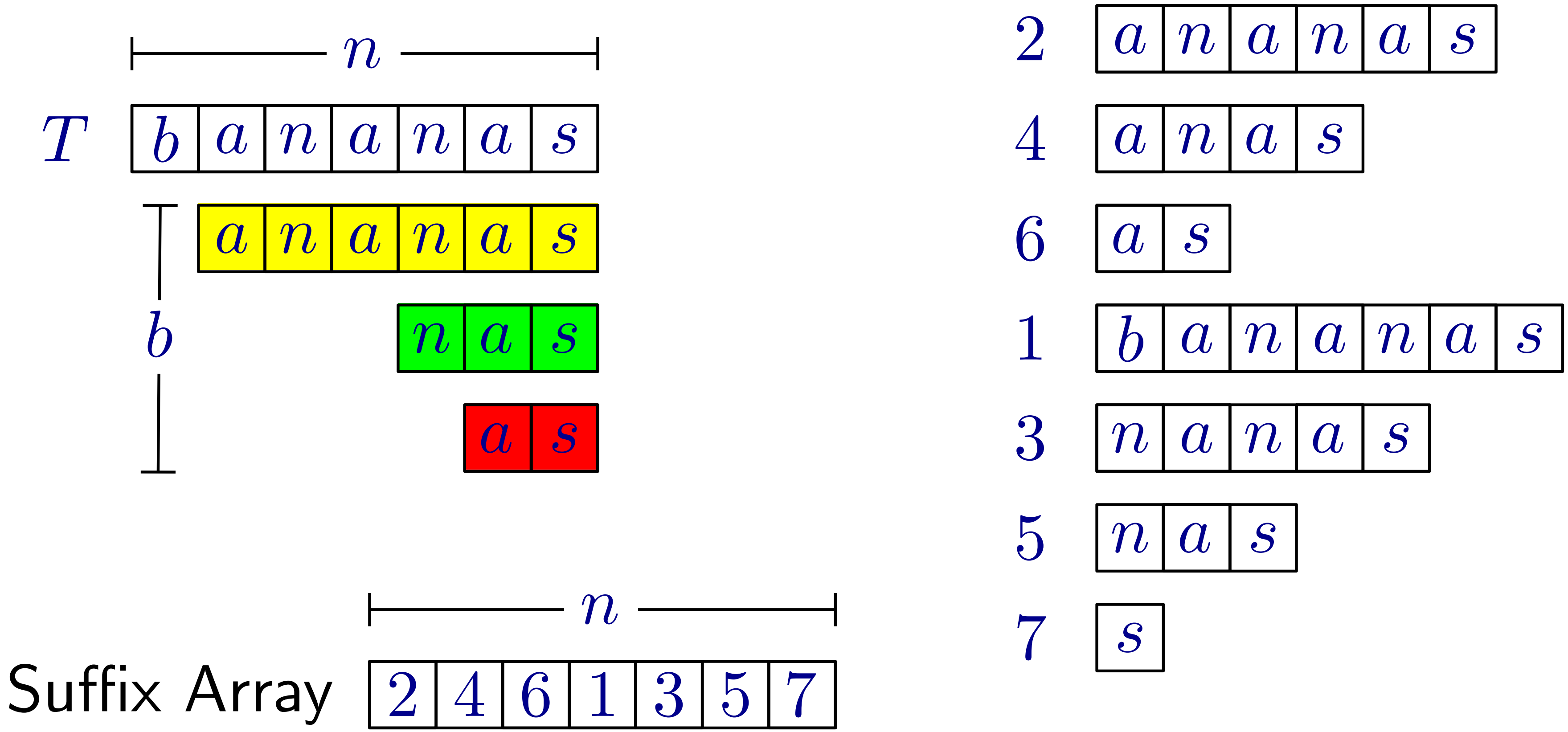
n

Suffix Array	2	4	6	1	3	5	7

- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

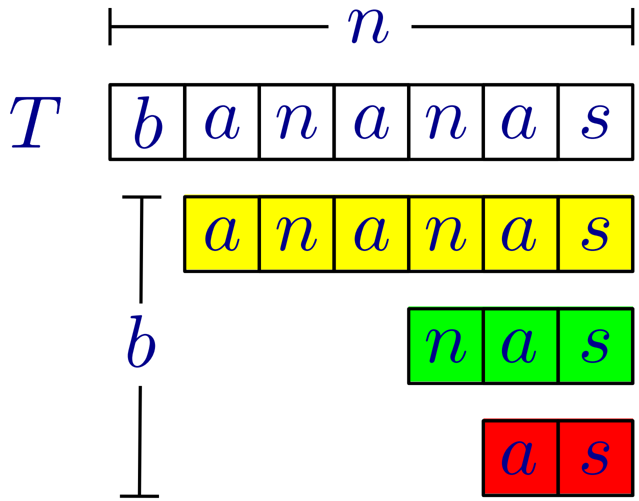
The sparse suffix array (SSA)

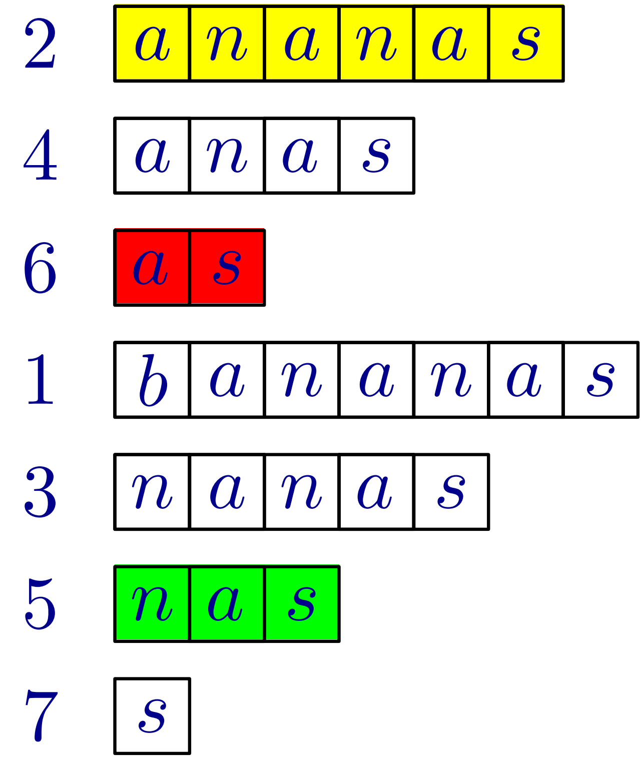


- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)





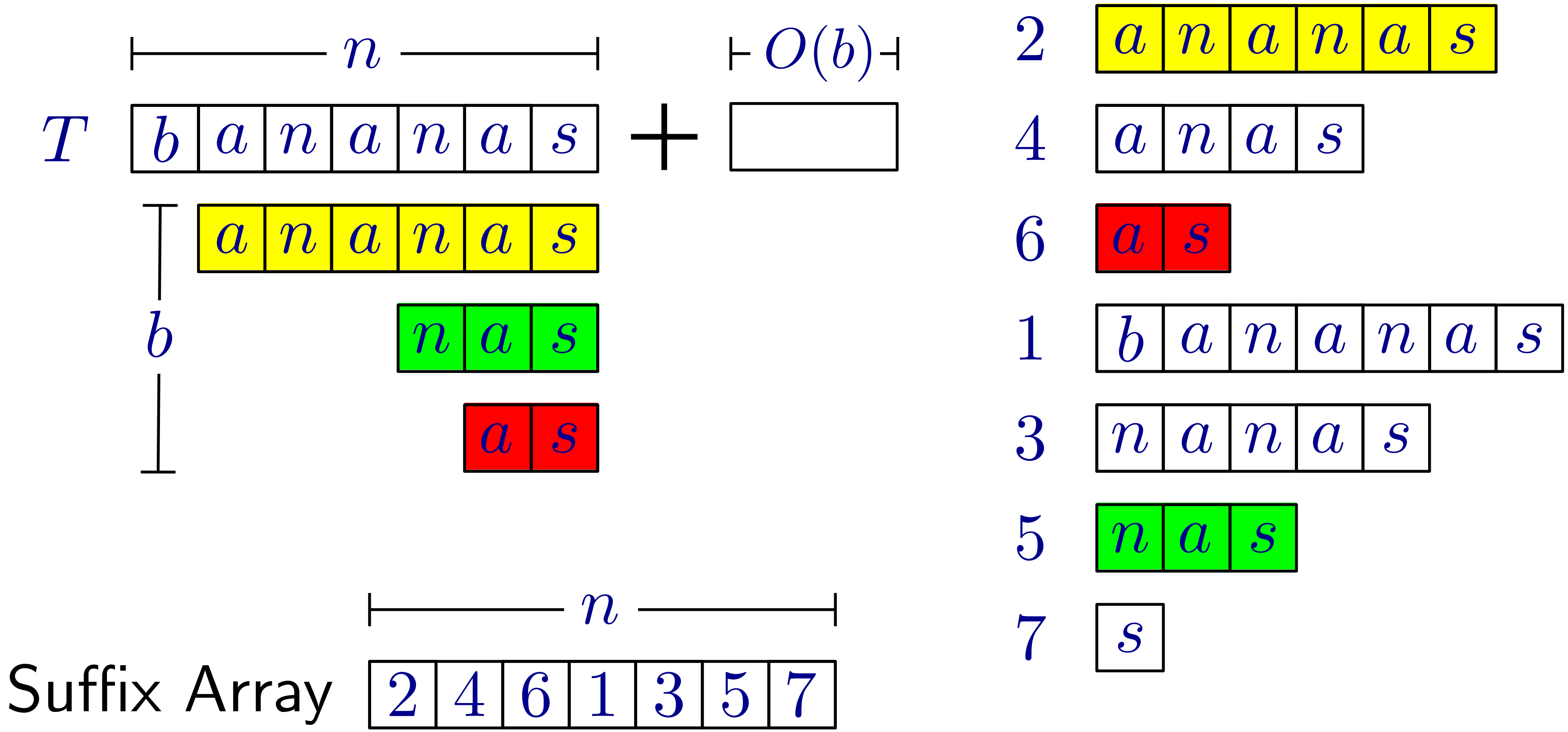
Suffix Array (n)

2	4	6	1	3	5	7

- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

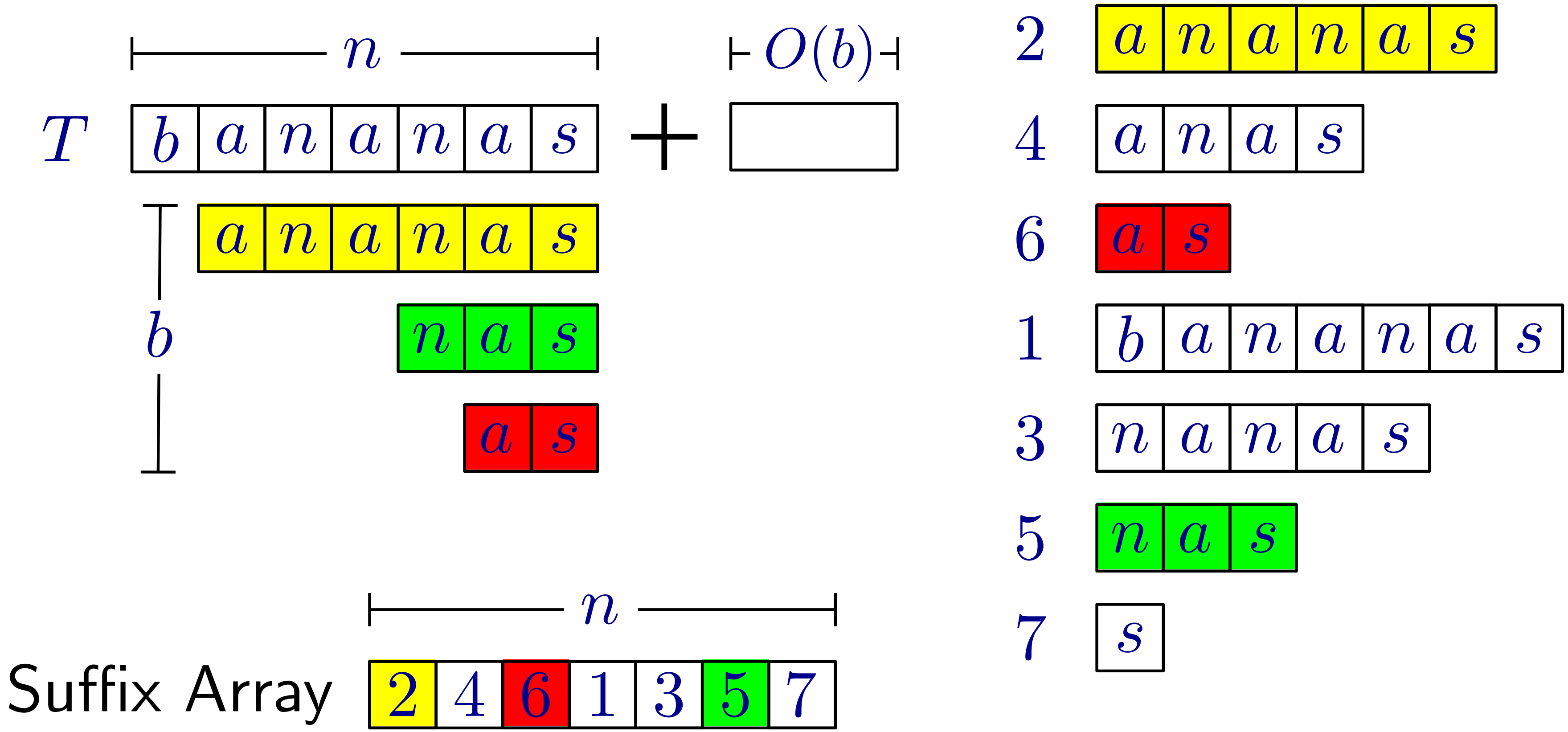
The sparse suffix array (SSA)



- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

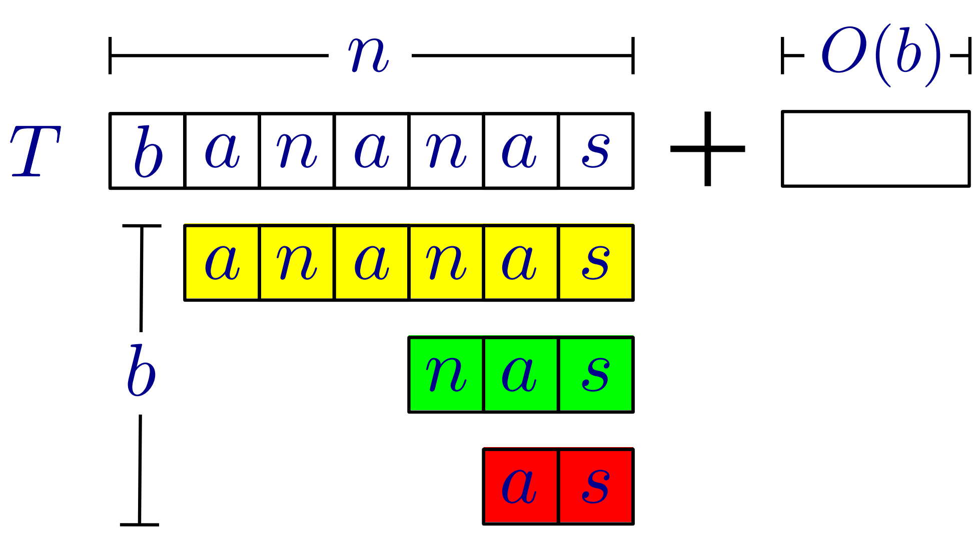
The sparse suffix array (SSA)

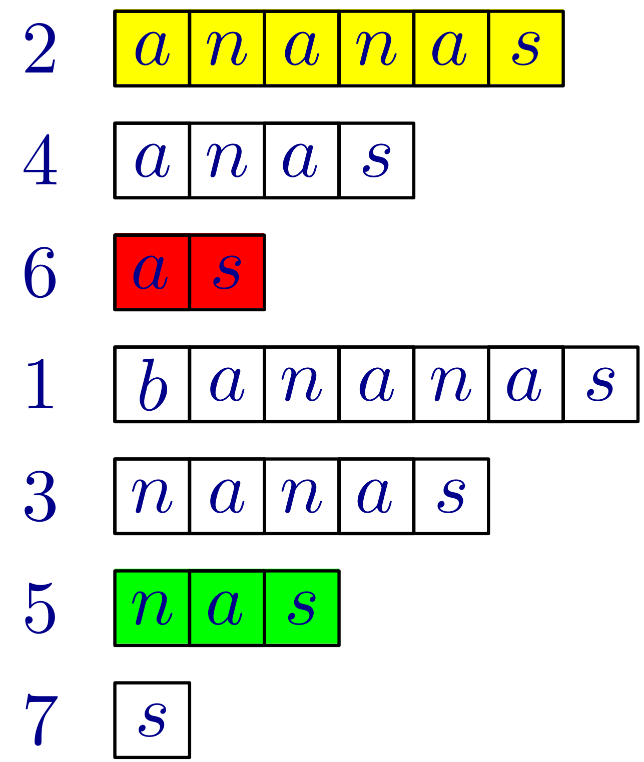


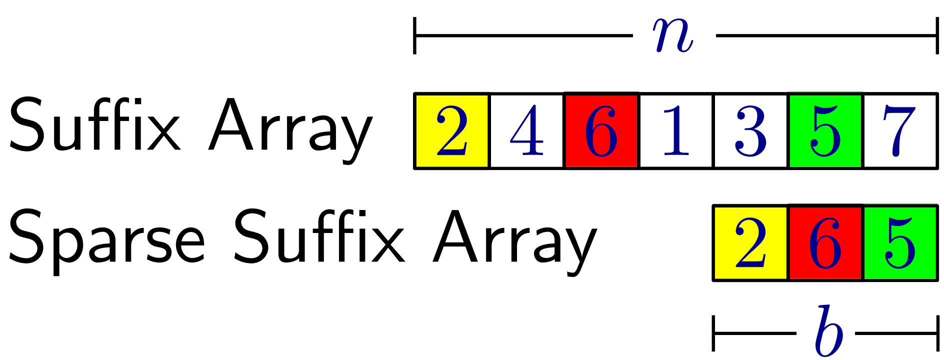
- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)



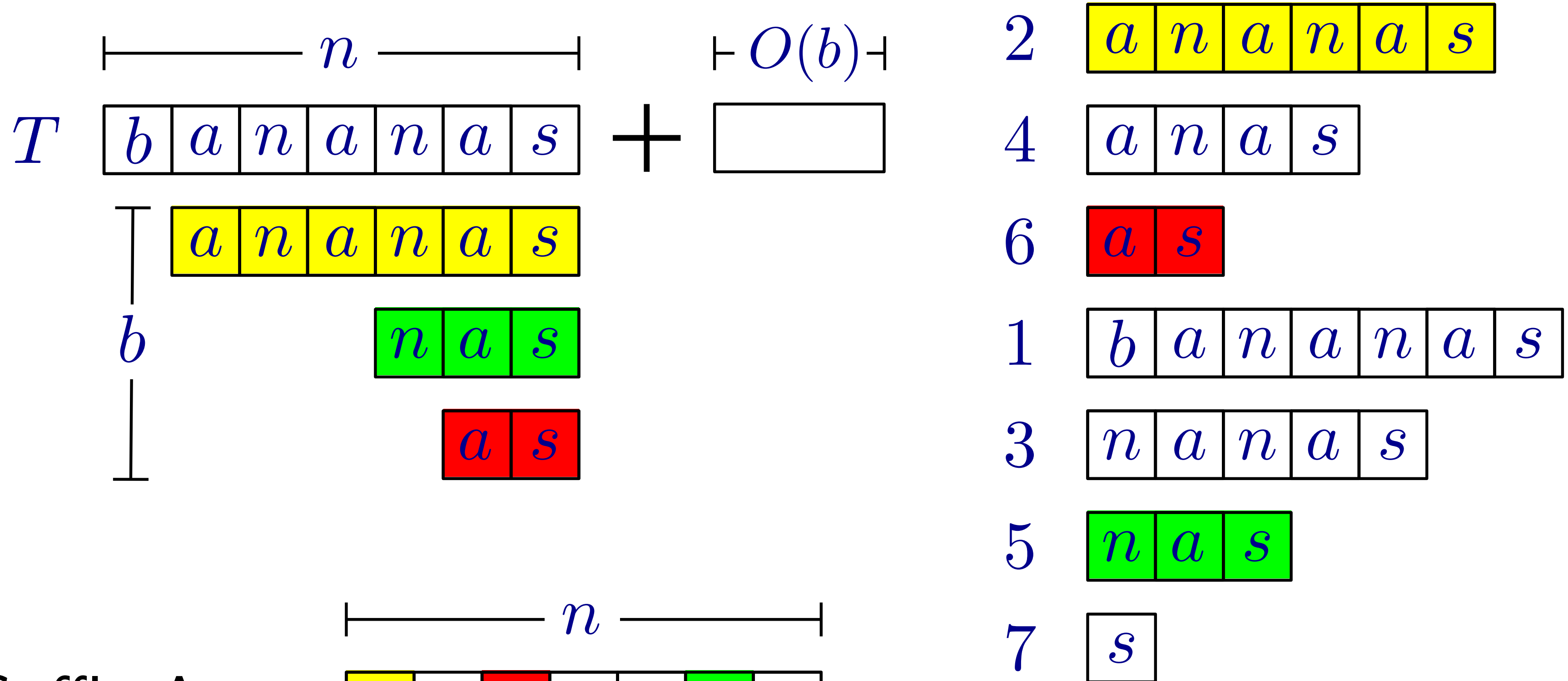


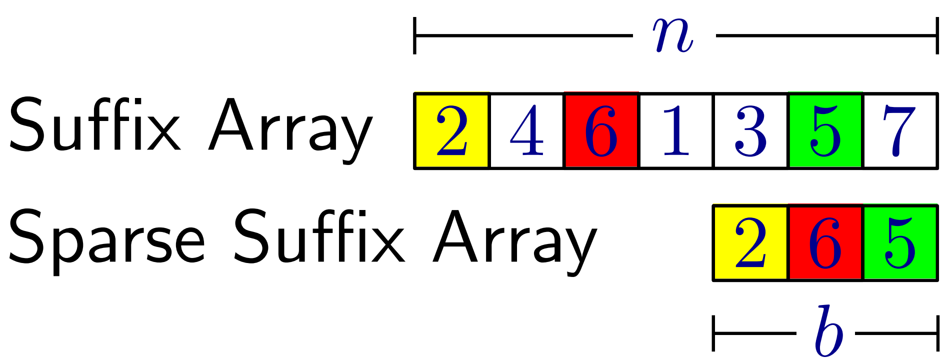


- Can be built in $O(n)$ time and $O(n)$ *extra space*

- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)



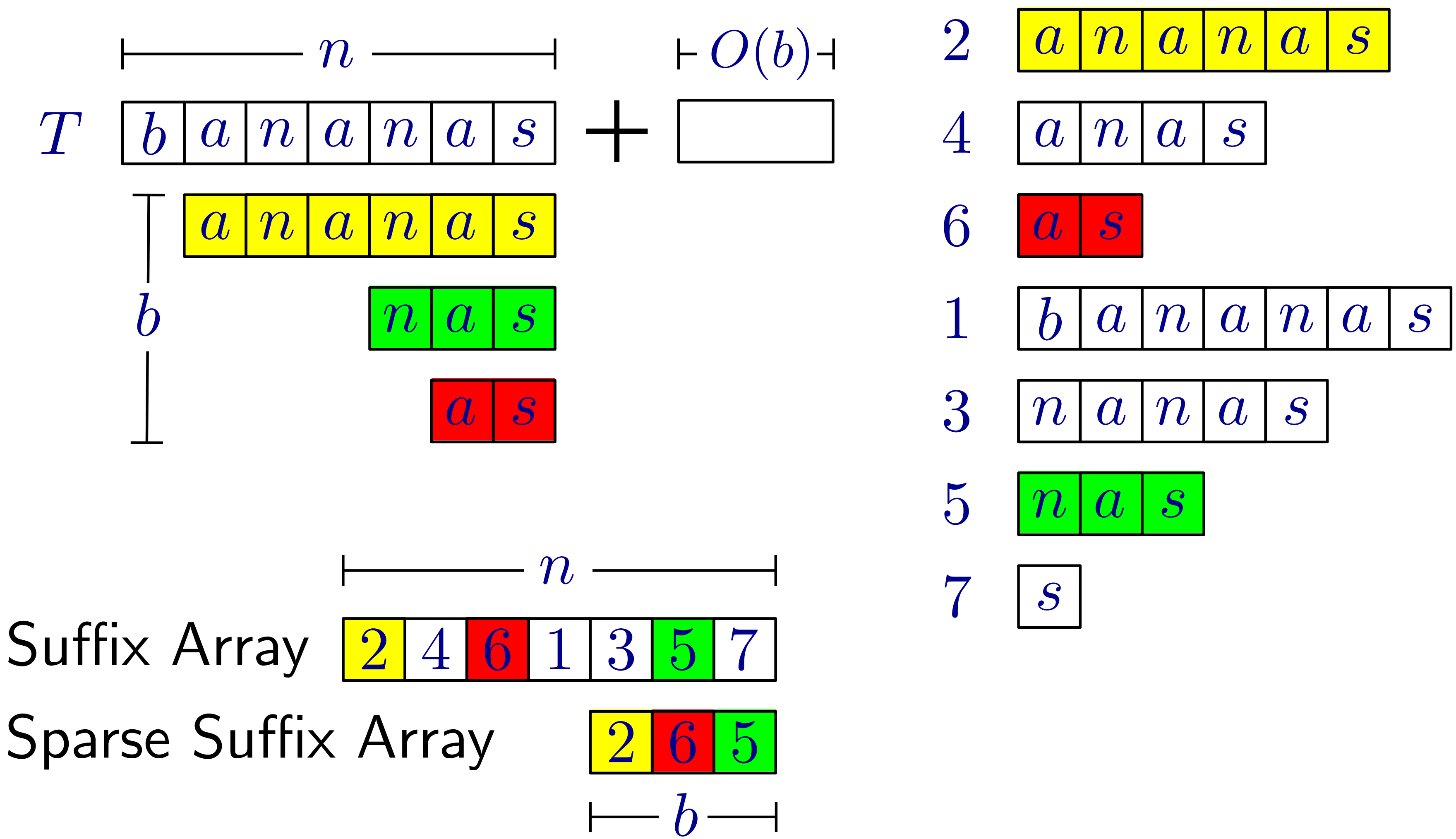


- Can be built in $O(n)$ time and $O(n)$ *extra space*

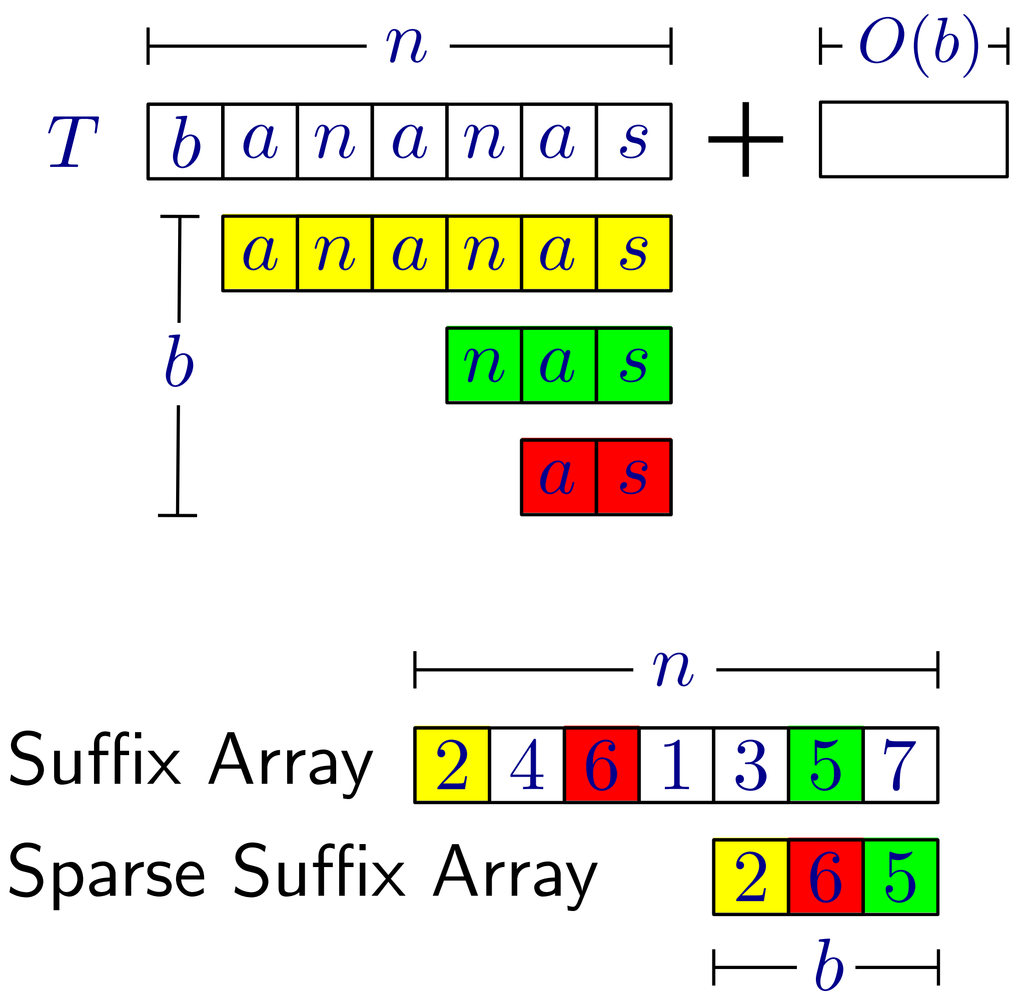
- What if we only care about a few of the suffixes?

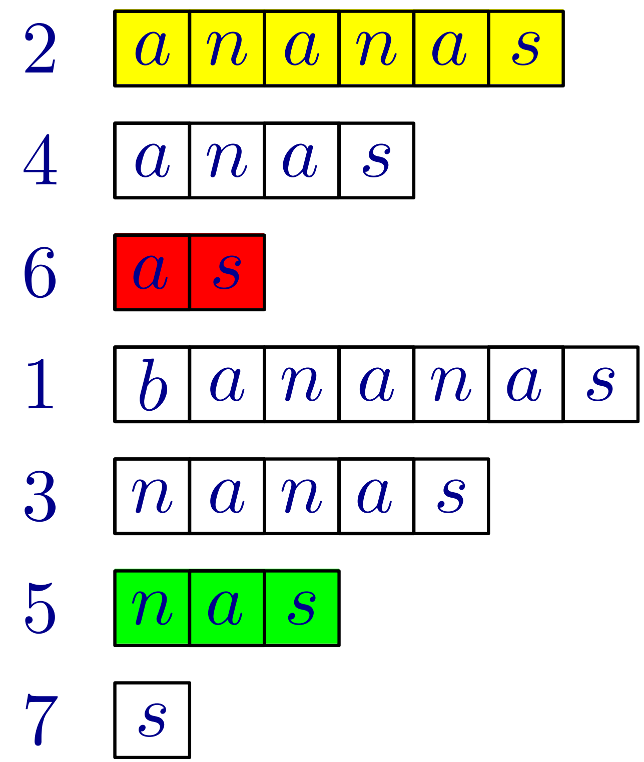
The sparse text indexing problem has been open since the 1960s
...with first, partial results from 1996 onwards

The sparse suffix array (SSA)



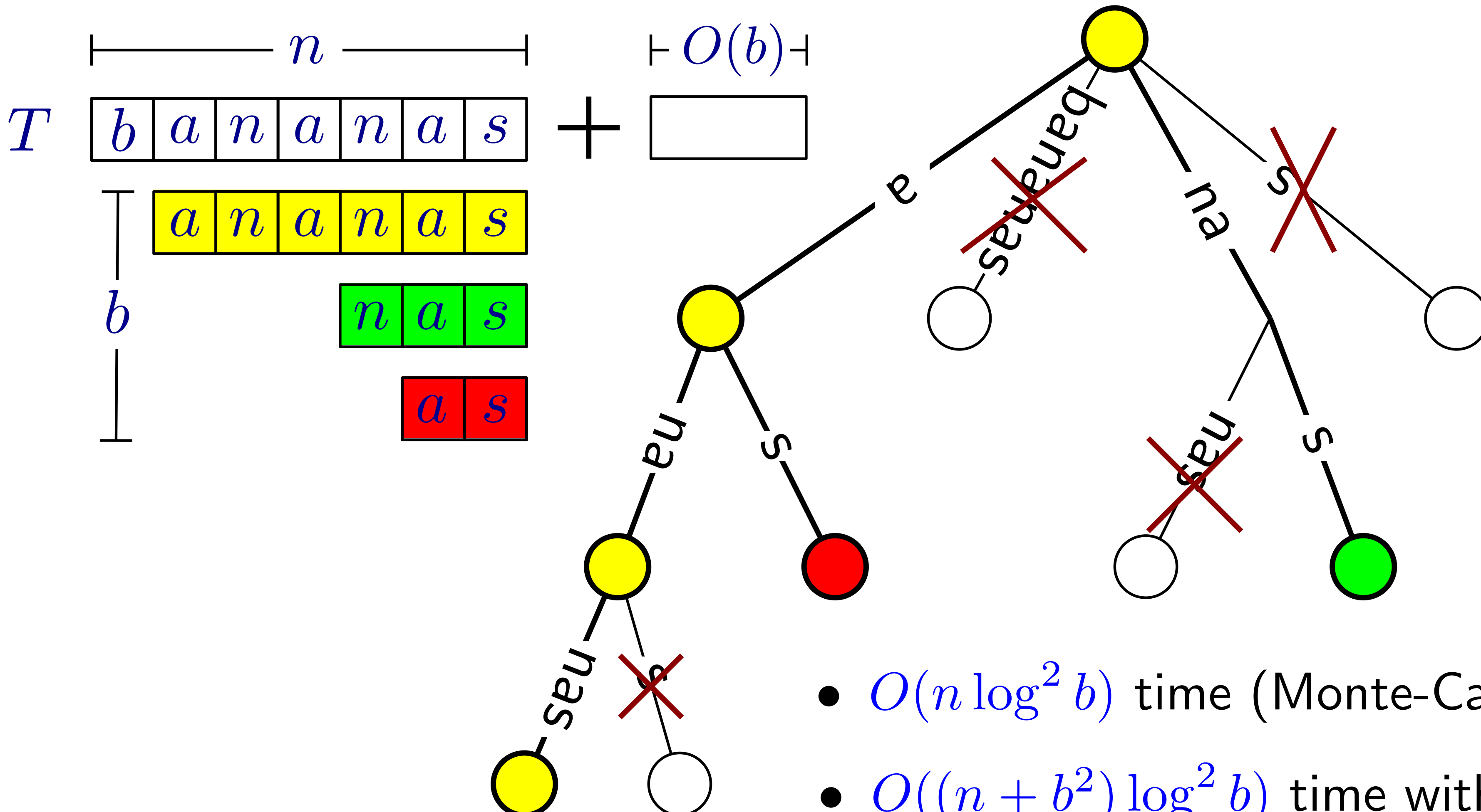
The sparse suffix array (SSA)





- $O(n \log^2 b)$ time (Monte-Carlo)
- $O((n + b^2) \log^2 b)$ time with high probability (Las-Vegas)
- both in $O(b)$ extra space

The sparse suffix tree (SST)



- $O(n \log^2 b)$ time (Monte-Carlo)
- $O((n + b^2) \log^2 b)$ time with high probability (Las-Vegas)
- both in $O(b)$ space

Conversion between SSA and SST is simple and takes $O(n \log b)$ *time*

LCPs - a fundamental tool for string algorithms

$$\overbrace{\hspace{20em}}^{n}$$

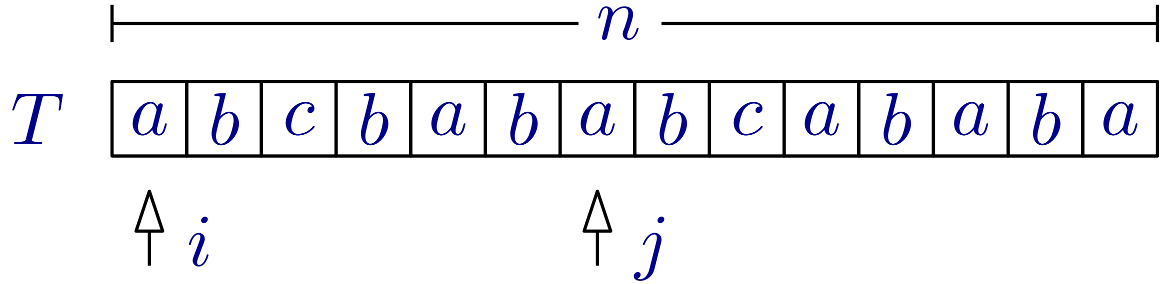
T	a	b	c	b	a	b	a	b	c	a	b	a	b	a

For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

it's the furthest you can go before hitting a mismatch

LCPs - a fundamental tool for string algorithms

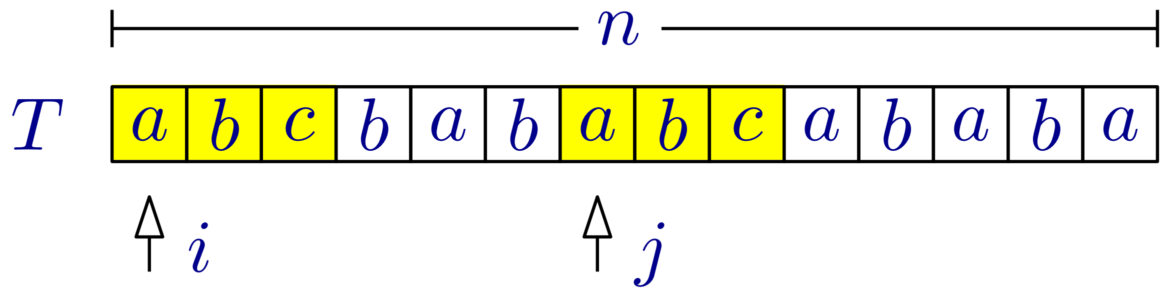


For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \dots i+\ell-1] = T[j \dots j+\ell-1]$$

it's the furthest you can go before hitting a mismatch

LCPs - a fundamental tool for string algorithms



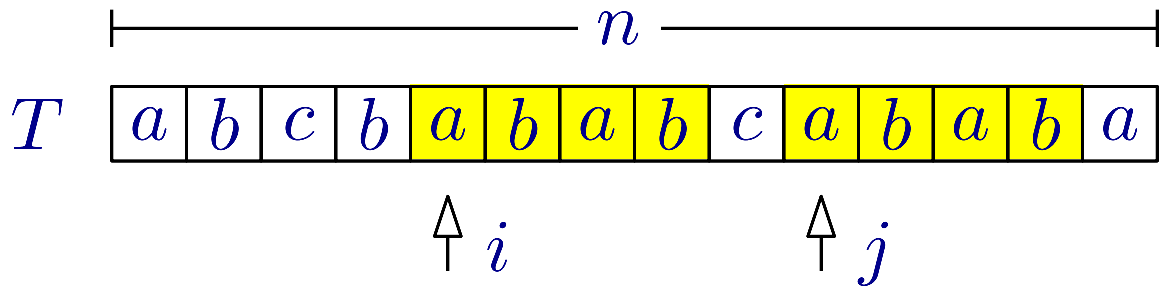
3

For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

it's the furthest you can go before hitting a mismatch

LCPs - a fundamental tool for string algorithms



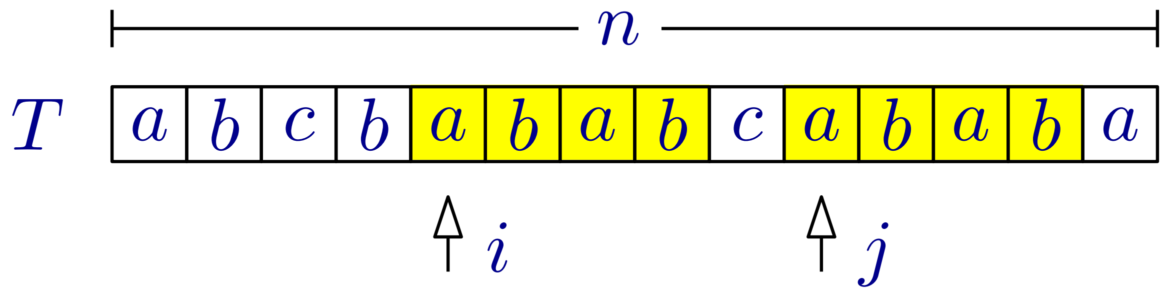
4

For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$

it's the furthest you can go before hitting a mismatch

LCPs - a fundamental tool for string algorithms



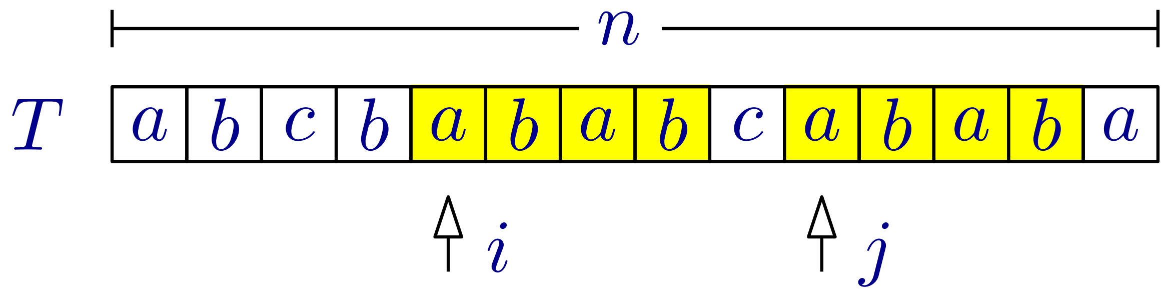
For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

it's the furthest you can go before hitting a mismatch

- LCP data structures are typically based
  on the suffix array or suffix tree.

LCPs - a fundamental tool for string algorithms



For any (i, j), the longest common prefix is the largest ℓ such that

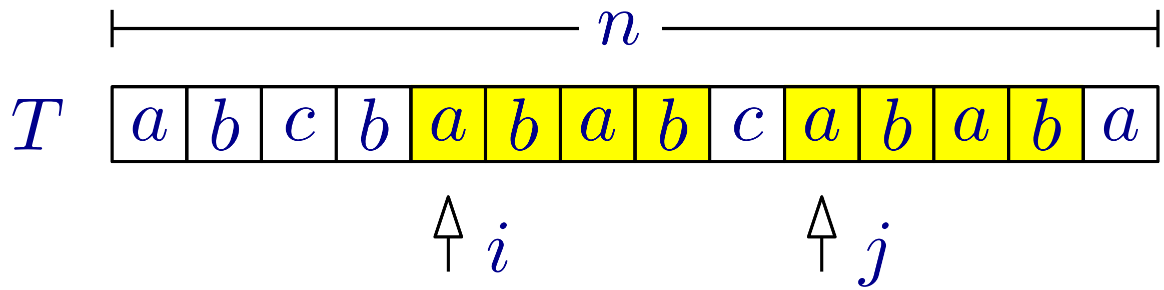
$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

it's the furthest you can go before hitting a mismatch

- LCP data structures are typically based
  on the suffix array or suffix tree.

- We do the opposite - we use *batched* LCP queries
  to construct the sparse suffix array

LCPs - a fundamental tool for string algorithms



For any (i, j), the longest common prefix is the largest ℓ such that

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

it's the furthest you can go before hitting a mismatch

- LCP data structures are typically based
  on the suffix array or suffix tree.
- We do the opposite - we use *batched* LCP queries
  to construct the sparse suffix array
- These LCP queries will be answered using Karp-Rabin fingerprints
  to ensure that the space remains small

Karp-Rabin fingerprints of strings

S

a	b	a	c	c	b	a	b	c	b

$$\phi(S) = \sum_{k=0}^{|S|-1} S[k]r^k \bmod p$$

Here $p = \Theta(n^4)$ is a prime and $1 \leq r < p$ is a random integer

with high probability, $S_1 = S_2$ iff $\phi(S_1) = \phi(S_2)$

Karp-Rabin fingerprints of strings

S	a	b	a	c	c	b	a	b	c	b

$$\phi(S) = \sum_{k=0}^{|S|-1} S[k]r^k \bmod p$$

Here $p = \Theta(n^4)$ is a prime and $1 \leq r < p$ is a random integer

with high probability, $S_1 = S_2$ iff $\phi(S_1) = \phi(S_2)$

Observe that $\phi(S)$ fits in an $O(\log n)$ bit word

Karp-Rabin fingerprints of strings

S	a	b	a	c	c	b	a	b	c	b

$$\phi(S) = \sum_{k=0}^{|S|-1} S[k]r^k \bmod p$$

Here $p = \Theta(n^4)$ is a prime and $1 \leq r < p$ is a random integer

with high probability, $S_1 = S_2$ iff $\phi(S_1) = \phi(S_2)$

Observe that $\phi(S)$ fits in an $O(\log n)$ bit word

Given $\phi(S[0, \ell])$ and $\phi(S[0, r])$ we can compute $\phi(S[\ell + 1, r])$ in $O(1)$ time

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$

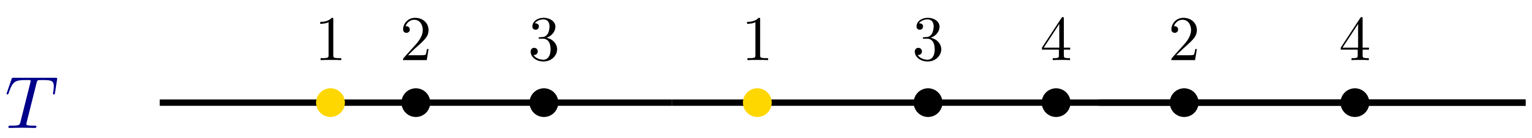
T

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$

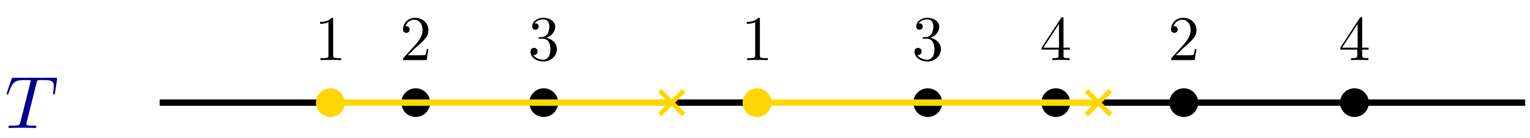


Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \dots i + \ell - 1] = T[j \dots j + \ell - 1]$$

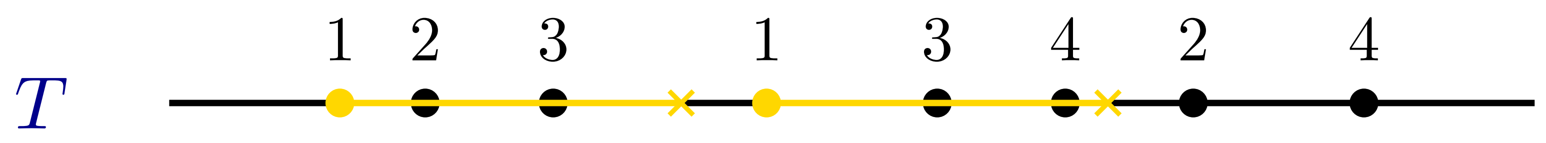


Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



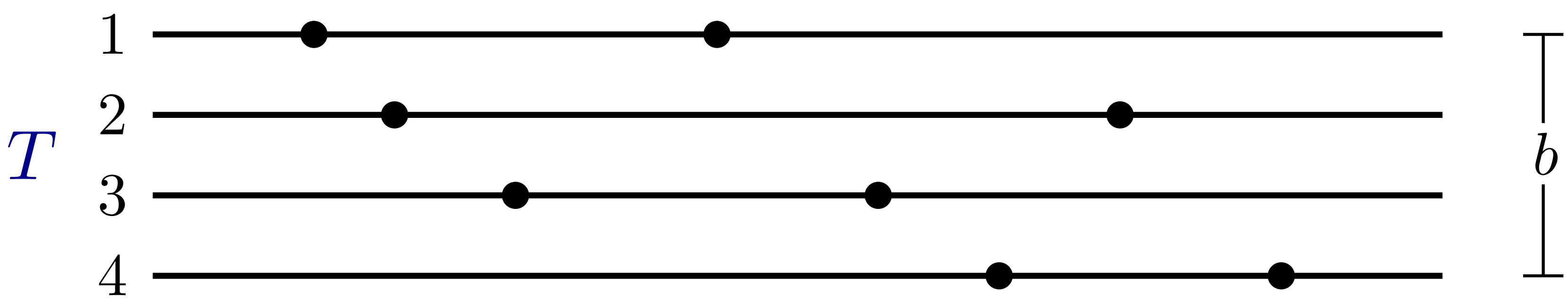
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



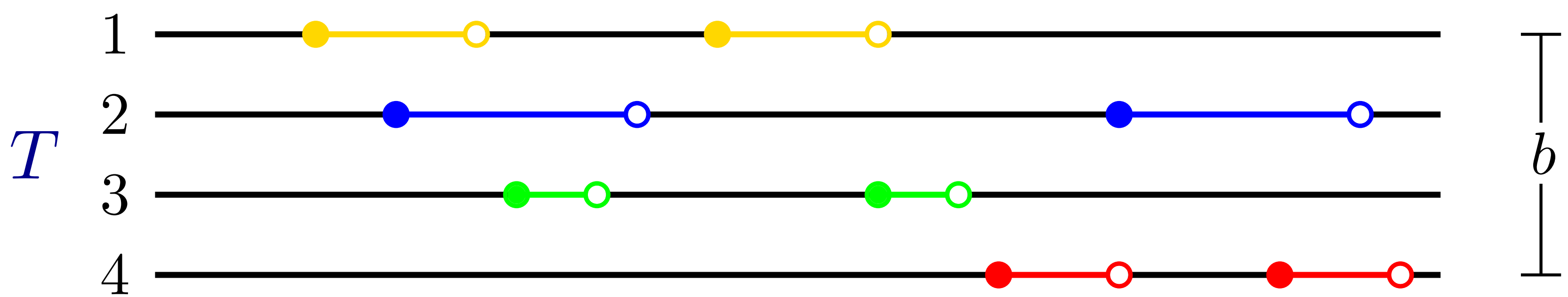
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i,j)

Output : for each pair (i,j) output the largest ℓ s.t.

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$



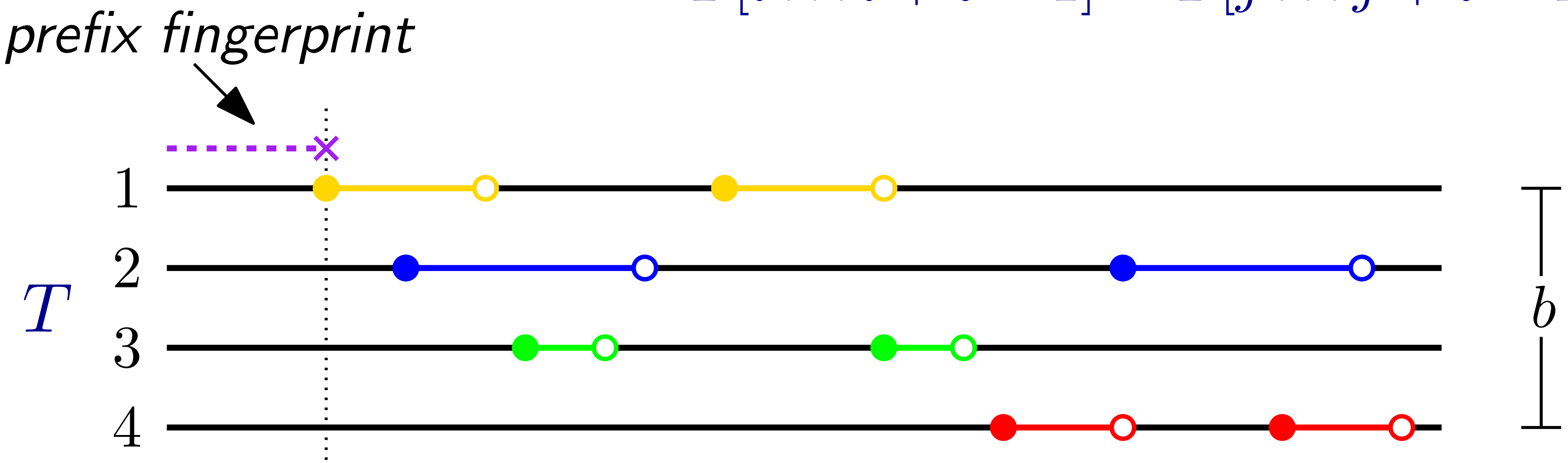
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i,j)

Output : for each pair (i,j) output the largest ℓ s.t.

$$T[i \dots i+\ell-1] = T[j \dots j+\ell-1]$$



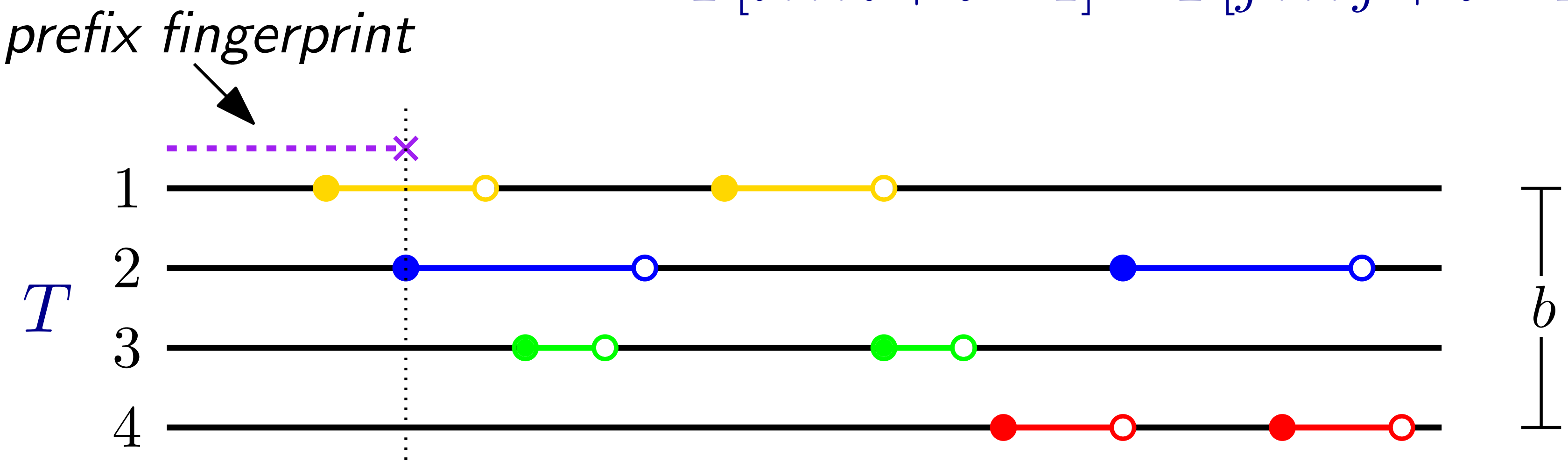
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



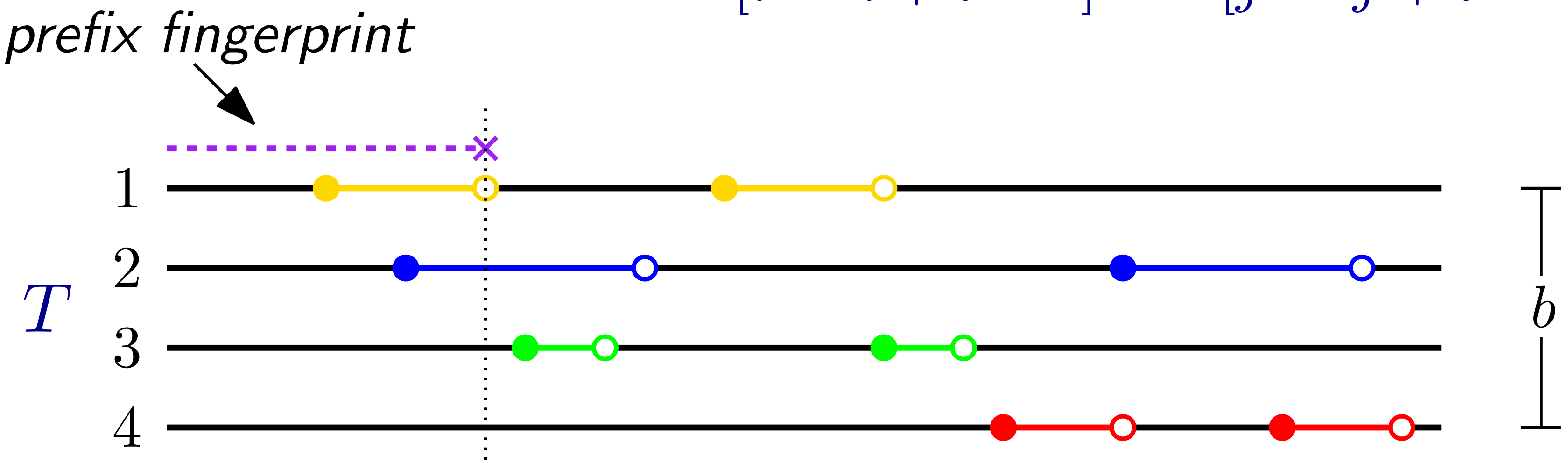
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



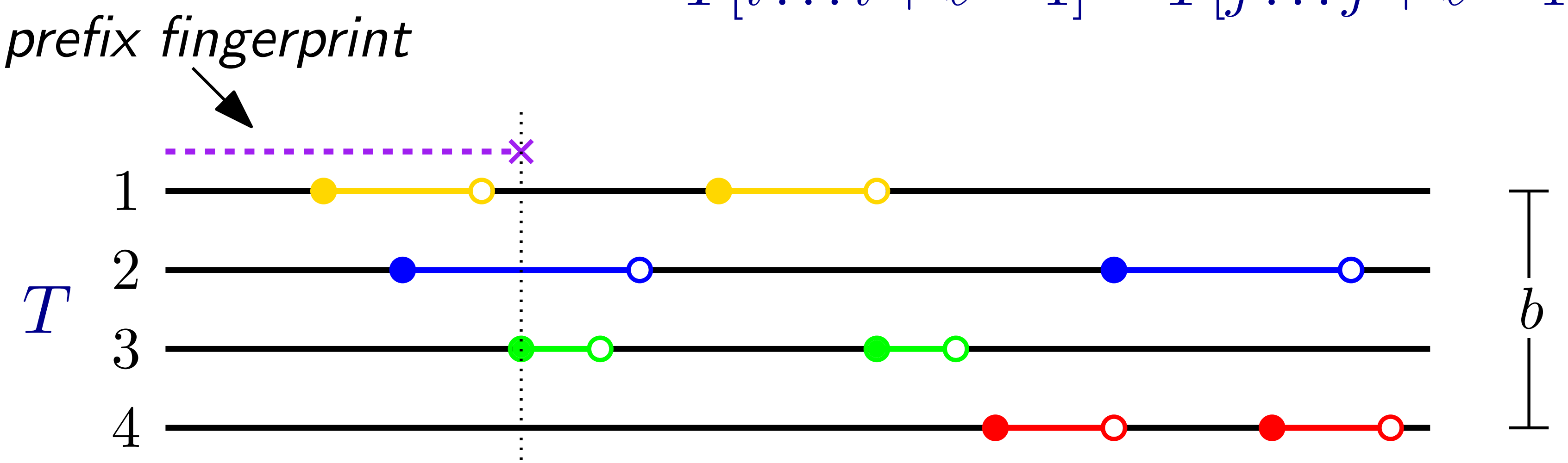
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \dots i + \ell - 1] = T[j \dots j + \ell - 1]$$



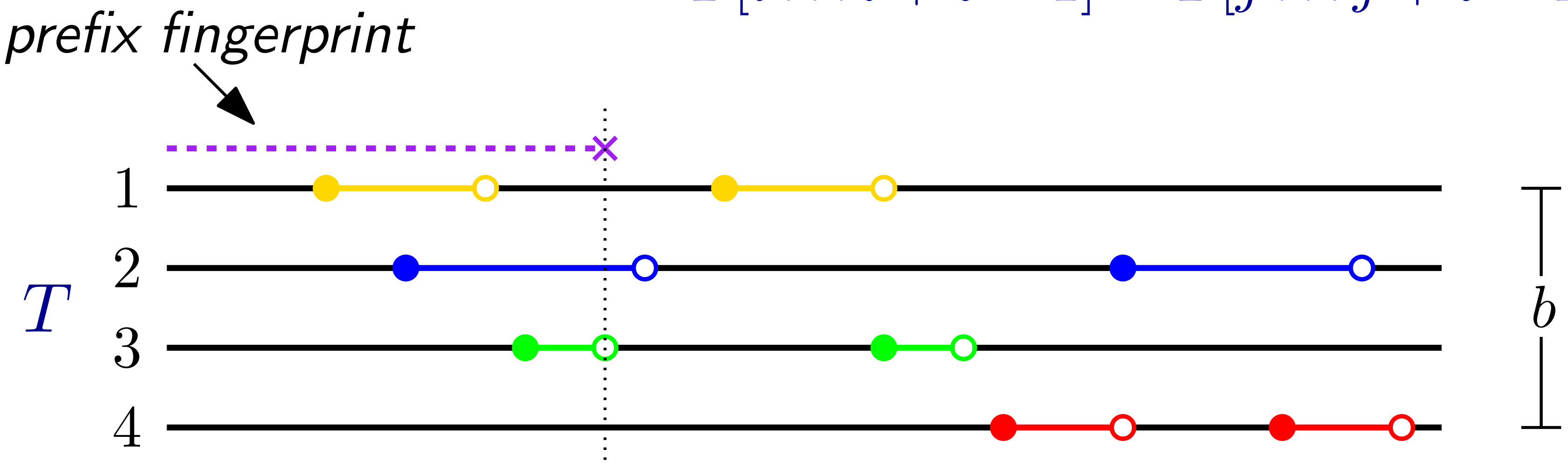
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \dots i + \ell - 1] = T[j \dots j + \ell - 1]$$



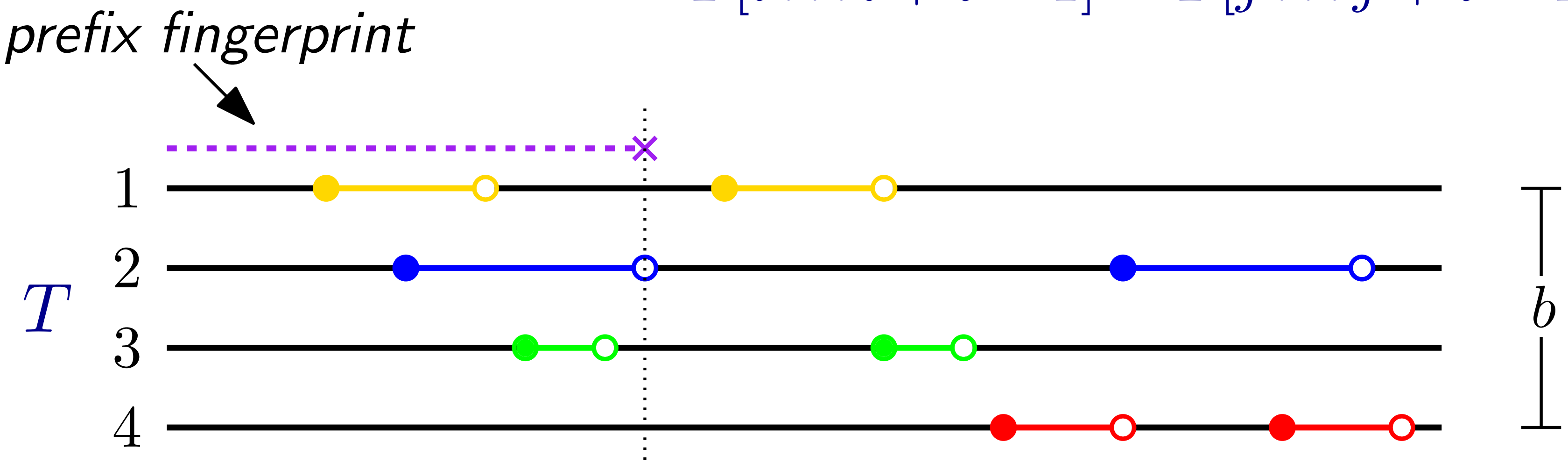
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



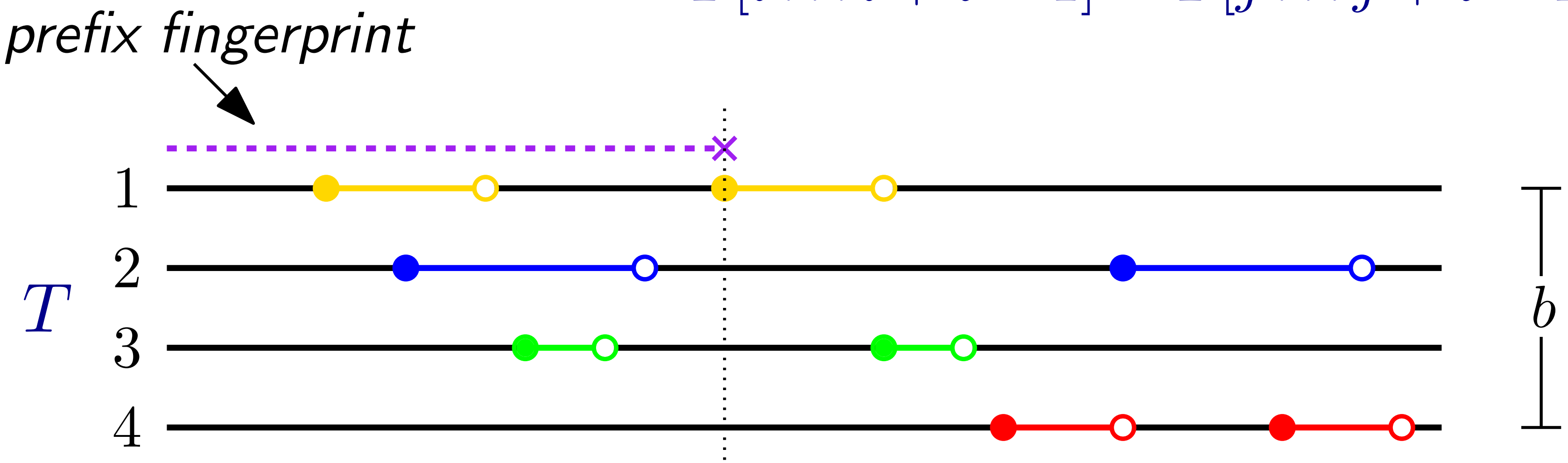
- We find the largest ℓ for each pair by binary search (in parallel)

  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i,j)
Output : for each pair (i,j) output the largest ℓ s.t.

$$T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$



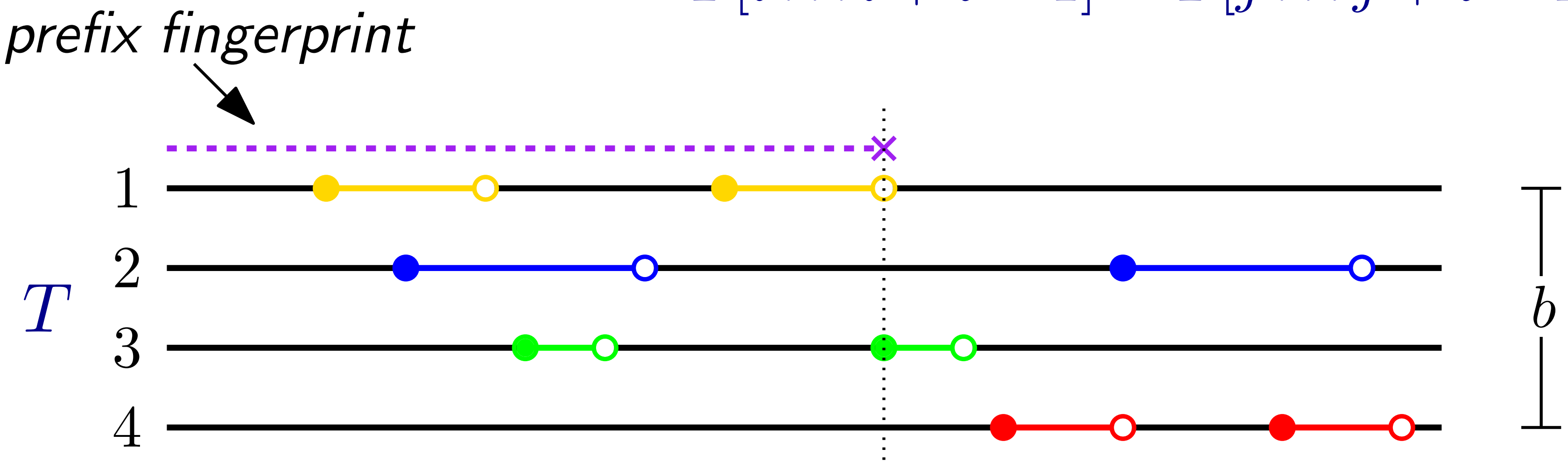
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



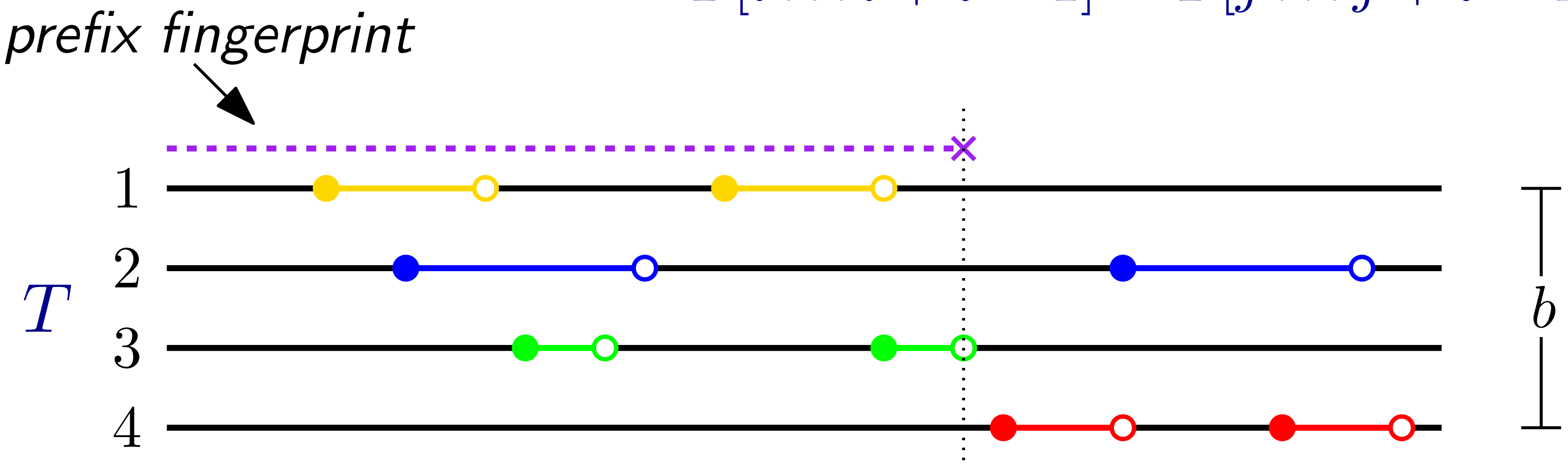
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \dots i + \ell - 1] = T[j \dots j + \ell - 1]$$



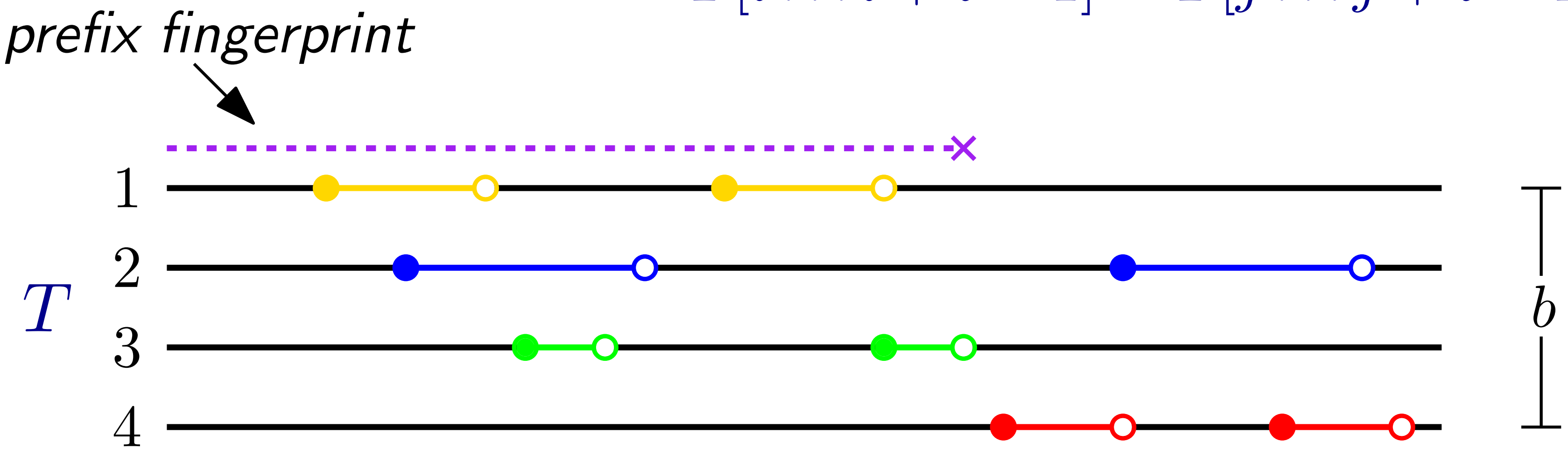
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



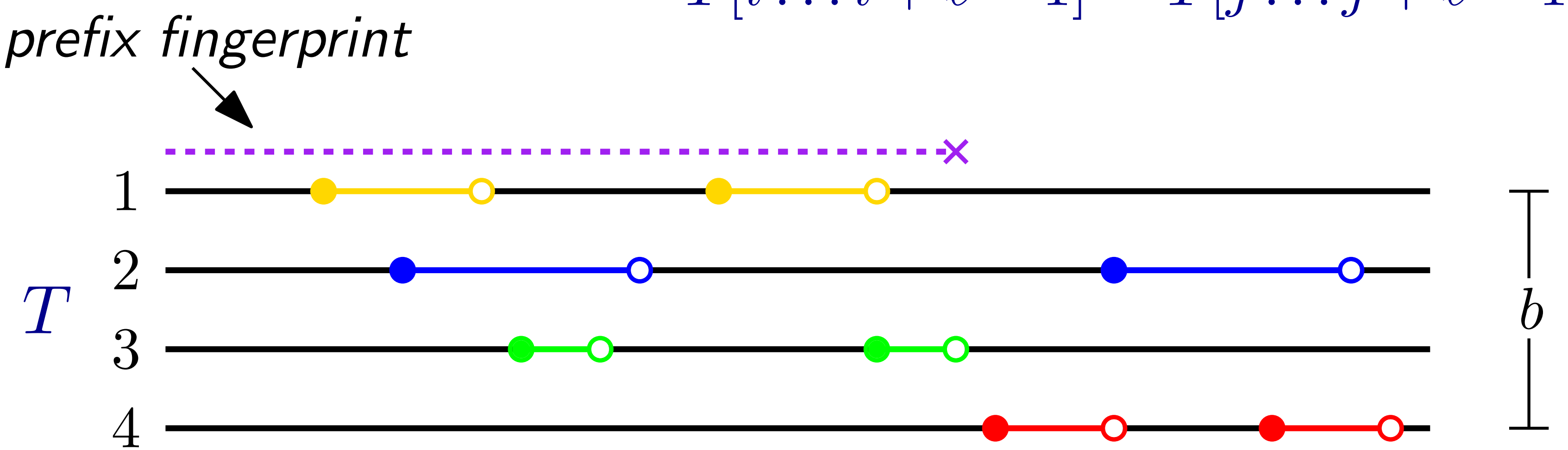
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints
- In each pass we store (at most) $4b$ prefix fingerprints

Simple, Monte-Carlo batched LCP queries

Input : a string, T of length n and b pairs, (i, j)

Output : for each pair (i, j) output the largest ℓ s.t.

$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints
- In each pass we store (at most) $4b$ prefix fingerprints

this takes $O(n \log b)$ time, $O(b)$ space and is correct whp.

Building the sparse suffix array using batched LCPs

T	b	a	n	a	n	a	s

Building the sparse suffix array using batched LCPs

T

b	a	n	a	n	a	s

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

Building the sparse suffix array using batched LCPs

T	b	a	n	a	n	a	s

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

Building the sparse suffix array using batched LCPs

T	b	a	n	a	n	a	s

The LCP of two suffixes gives us their order

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

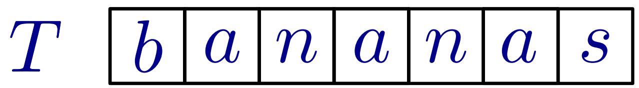
Building the sparse suffix array using batched LCPs

T							
	b	a	n	a	n	a	s

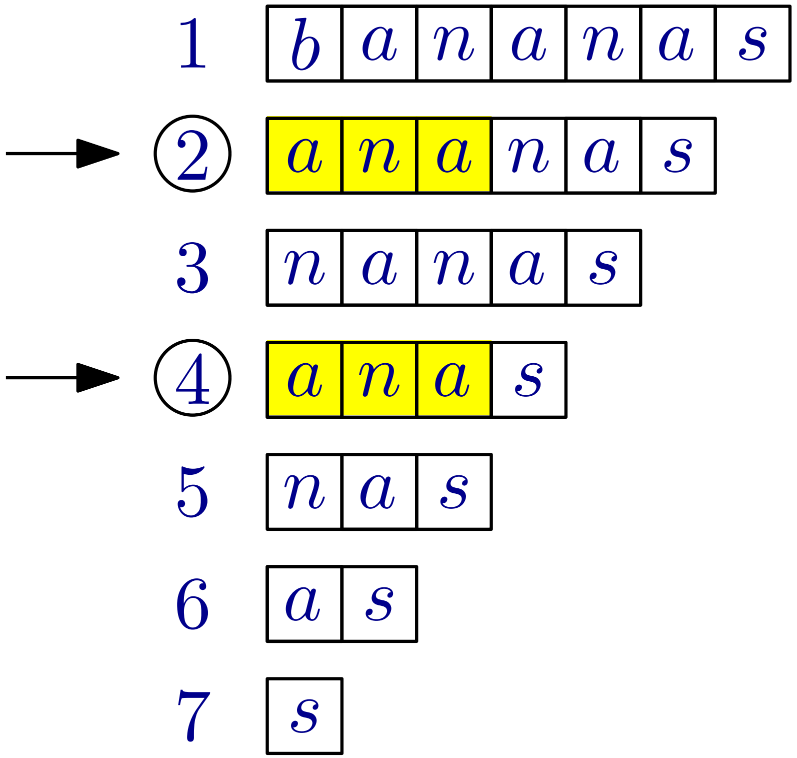
The LCP of two suffixes gives us their order

1	b	a	n	a	n	a	s
2	a	n	a	n	a	s	
3	n	a	n	a	s		
4	a	n	a	s			
5	n	a	s				
6	a	s					
7	s						

Building the sparse suffix array using batched LCPs



The LCP of two suffixes gives us their order

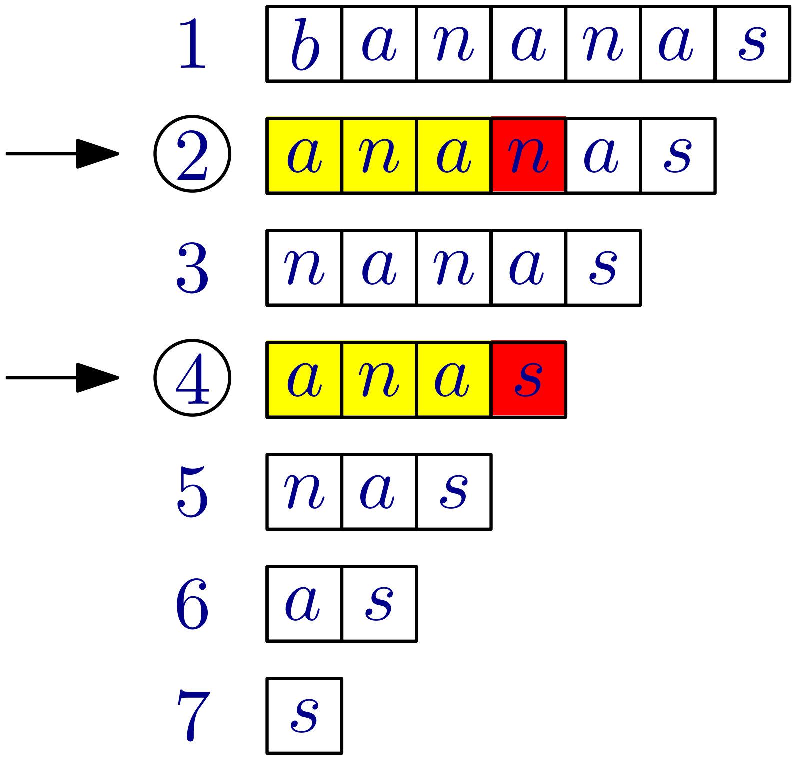


Building the sparse suffix array using batched LCPs

T | b | a | n | a | n | a | s |

The LCP of two suffixes gives us their order

②<④ because $\boxed{n} < \boxed{s}$



Building the sparse suffix array using batched LCPs

T $\boxed{b}\boxed{a}\boxed{n}\boxed{a}\boxed{n}\boxed{a}\boxed{s}$

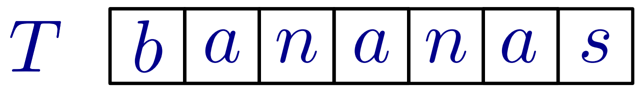
1	$\boxed{b}\boxed{a}\boxed{n}\boxed{a}\boxed{n}\boxed{a}\boxed{s}$
→ 2	$\boxed{a}\boxed{n}\boxed{a}\boxed{n}\boxed{a}\boxed{s}$
3	$\boxed{n}\boxed{a}\boxed{n}\boxed{a}\boxed{s}$
→ 4	$\boxed{a}\boxed{n}\boxed{a}\boxed{s}$
5	$\boxed{n}\boxed{a}\boxed{s}$
6	$\boxed{a}\boxed{s}$
7	$\boxed{s}$

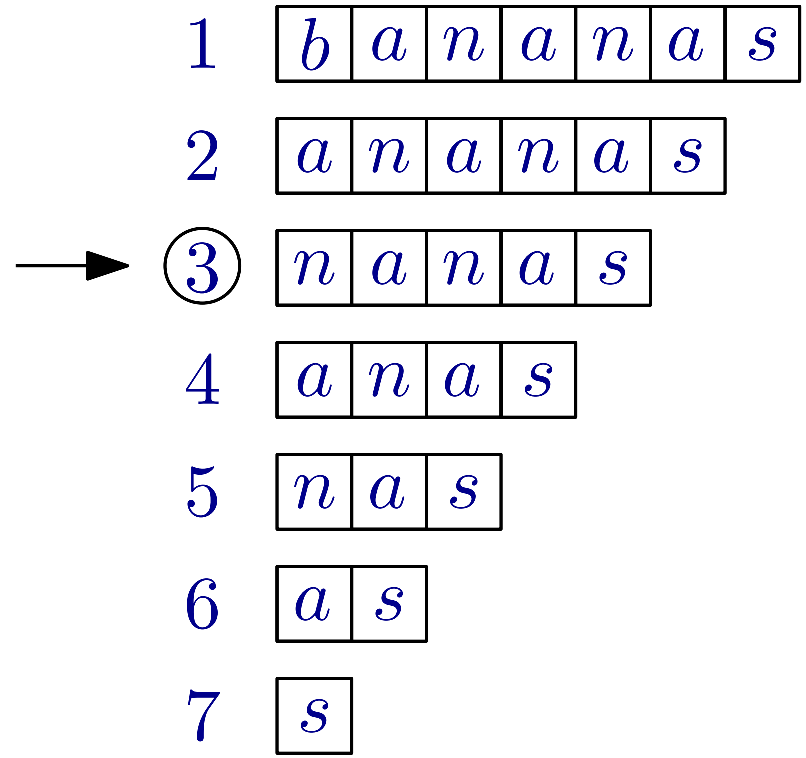
The LCP of two suffixes gives us their order

$2 < 4$ because $\boxed{n} < \boxed{s}$

- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons

Building the sparse suffix array using batched LCPs

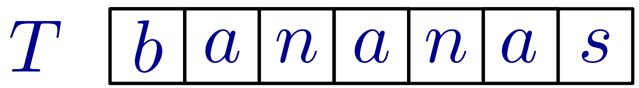




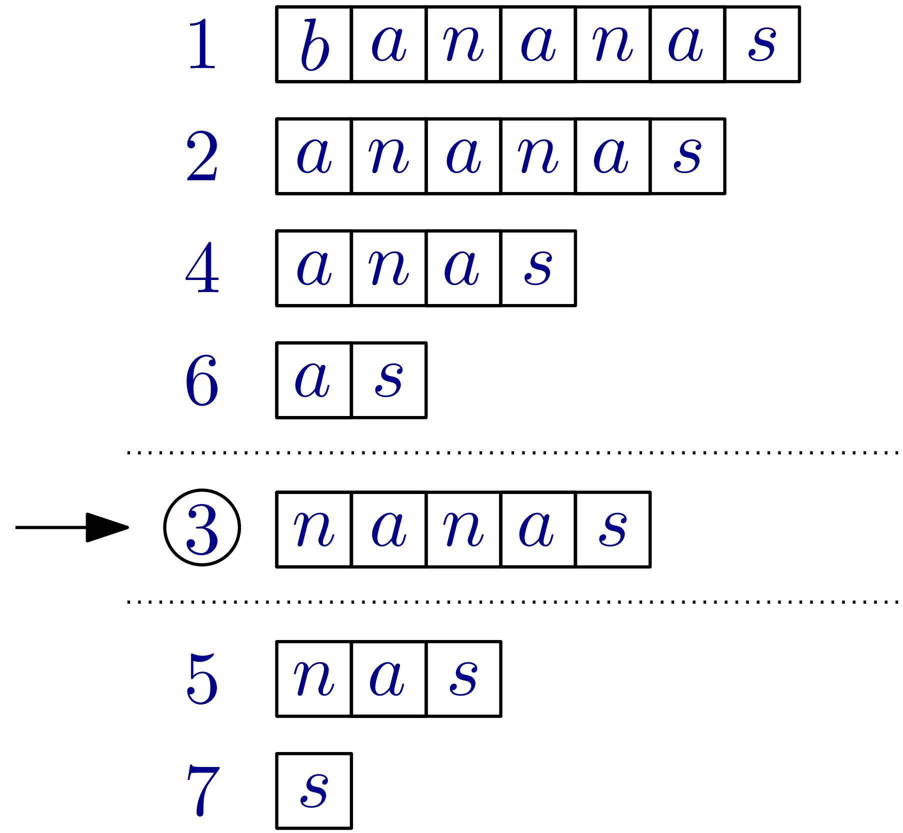
The LCP of two suffixes gives us their order

- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- Pick a random pivot and compare each other suffix to it
  - This partitions the suffixes in $O(n \log b)$ time and $O(b)$ space

Building the sparse suffix array using batched LCPs

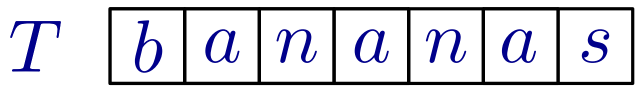


The LCP of two suffixes gives us their order

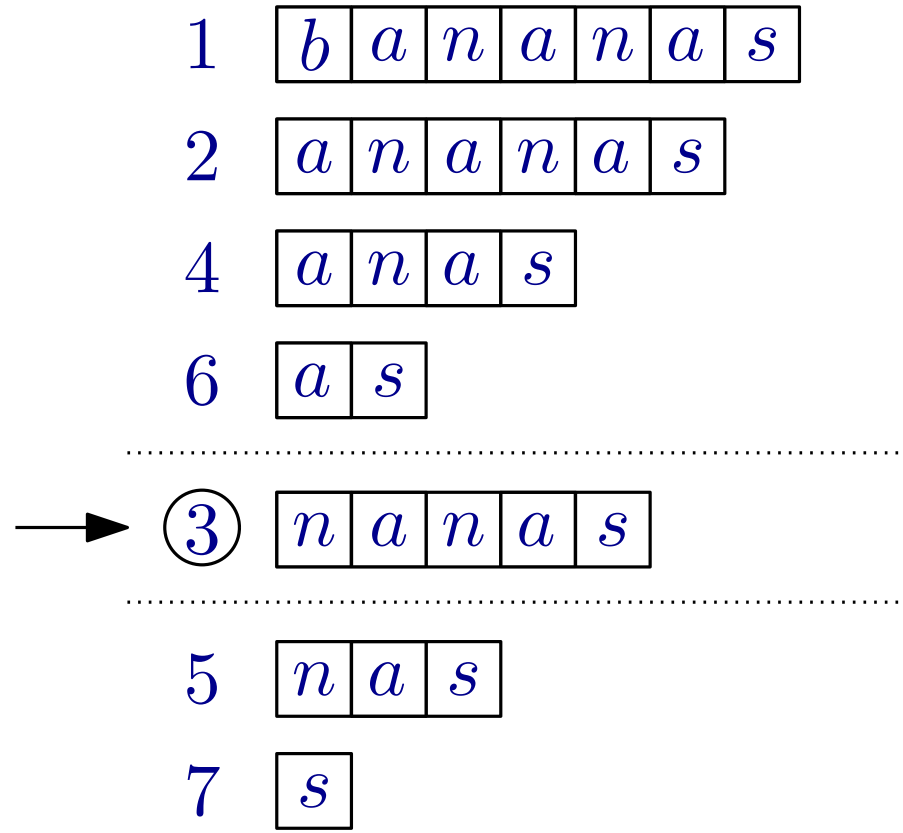


- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- Pick a random pivot and compare each other suffix to it
  - This partitions the suffixes in $O(n \log b)$ time and $O(b)$ space

Building the sparse suffix array using batched LCPs

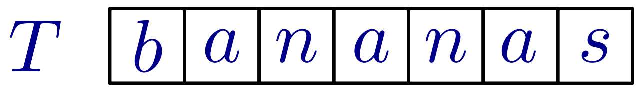


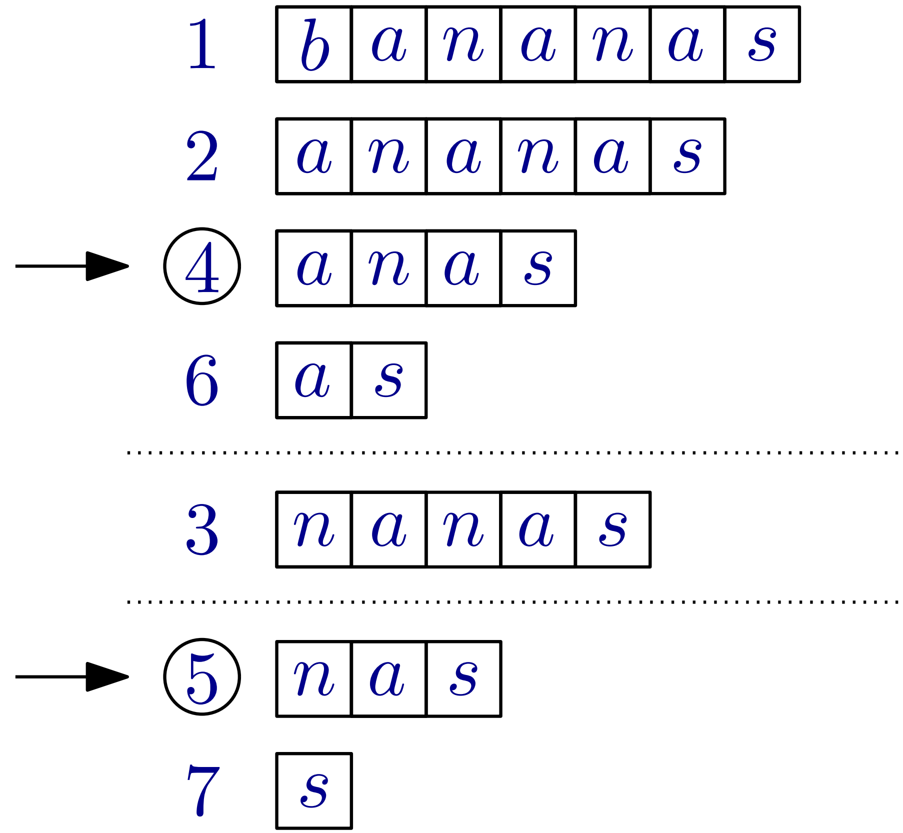
The LCP of two suffixes gives us their order



- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- Pick a random pivot and compare each other suffix to it
  - This partitions the suffixes in $O(n \log b)$ time and $O(b)$ space
  - *Recurse on each partition (the batch still contains b LCPs)*

Building the sparse suffix array using batched LCPs

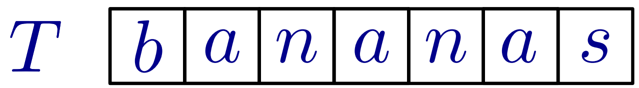




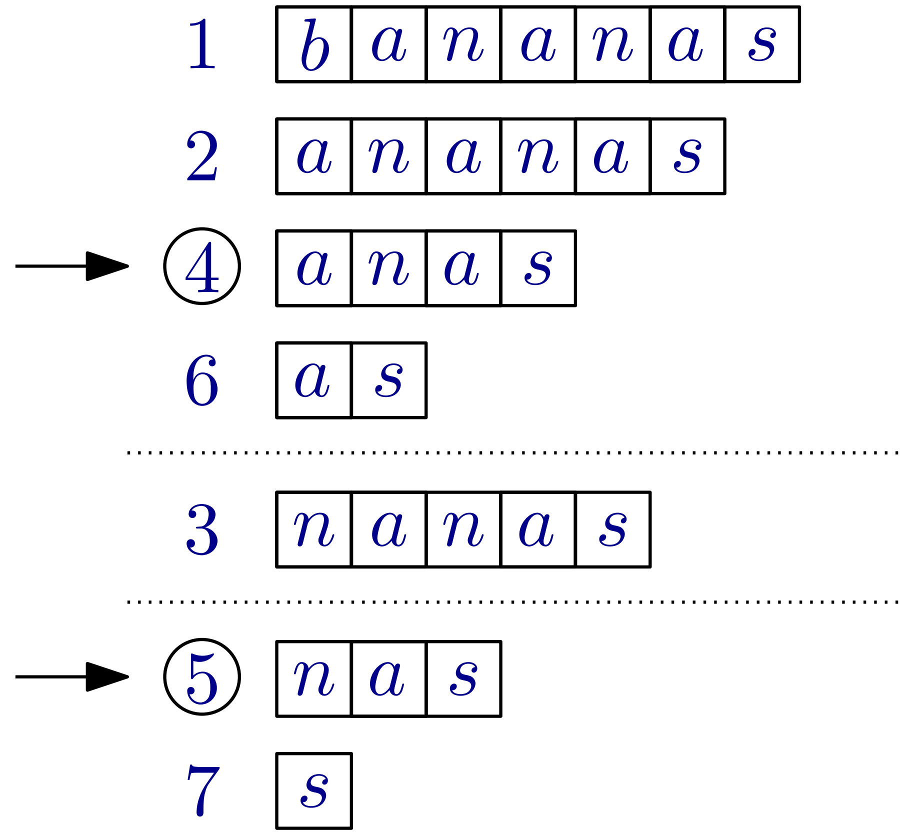
The LCP of two suffixes gives us their order

- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- Pick a random pivot and compare each other suffix to it
  - This partitions the suffixes in $O(n \log b)$ time and $O(b)$ space
  - *Recurse on each partition (the batch still contains b LCPs)*

Building the sparse suffix array using batched LCPs



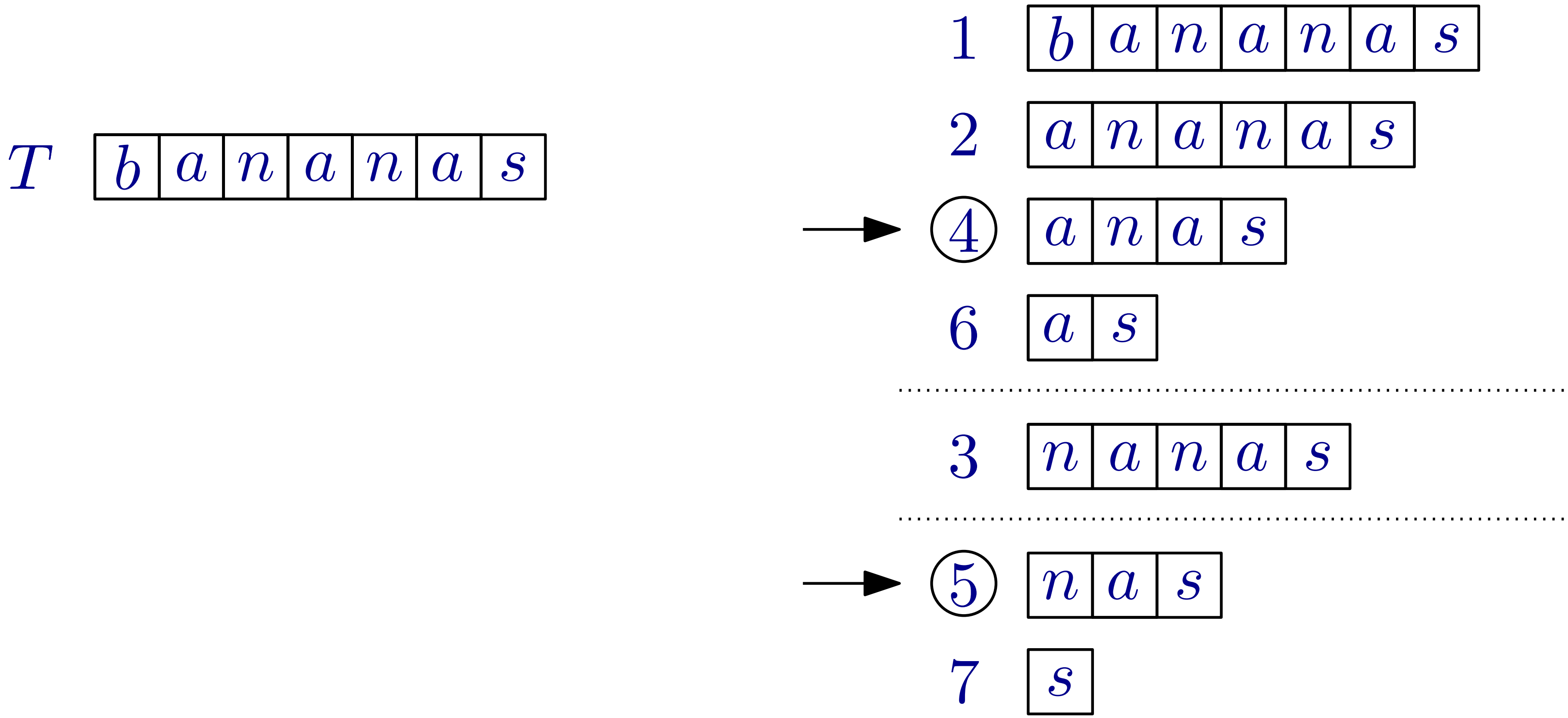
The LCP of two suffixes gives us their order



- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- The depth of the recursion is $O(\log b)$ whp. so...

  The total time is $O(n \log^2 b)$ and the space is $O(b)$

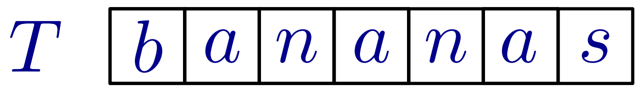
Building the sparse suffix array using batched LCPs

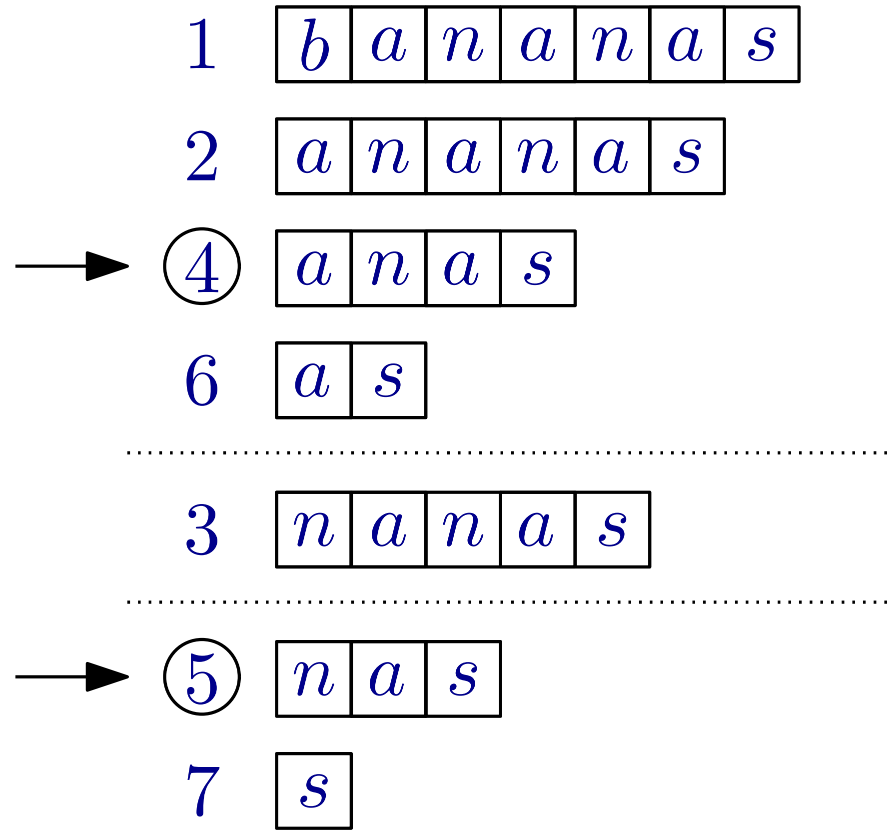


- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- The depth of the recursion is $O(\log b)$ whp. so...

  The total time is $O(n \log^2 b)$ and the space is $O(b)$

Building the sparse suffix array using batched LCPs



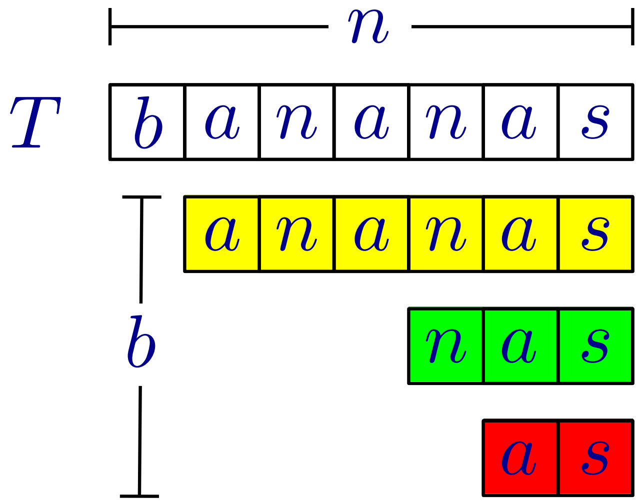


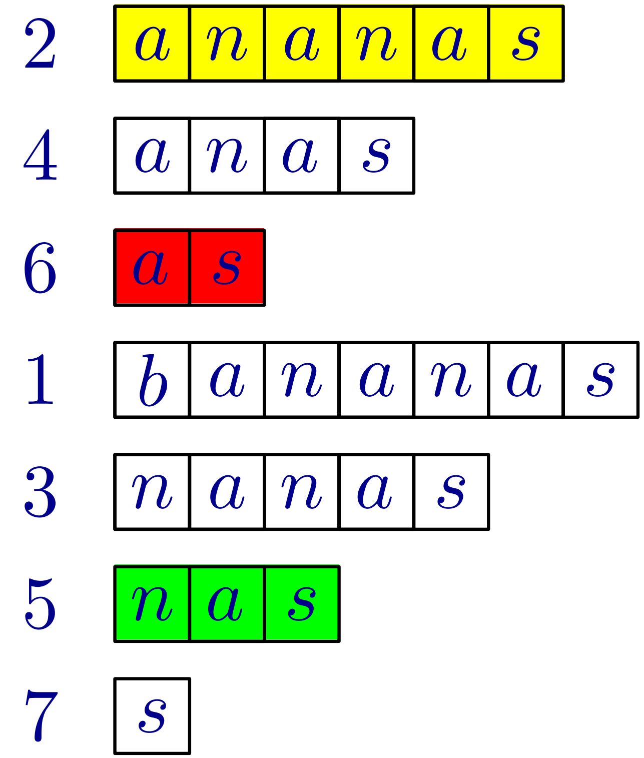
This algorithm is Monte-Carlo *and* Las-Vegas. It can be made Monte-Carlo only by aborting the quicksort early

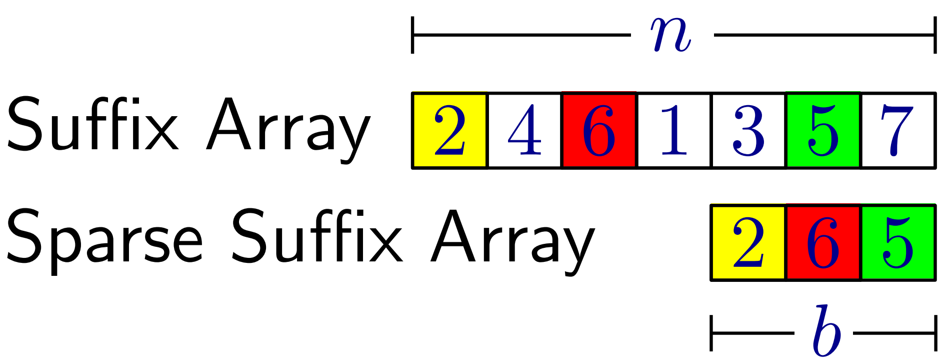
- We perform randomised quicksort on the b suffixes using batched LCPs for suffix comparisons
- The depth of the recursion is $O(\log b)$ whp. so...

The total time is $O(n \log^2 b)$ and the space is $O(b)$

The sparse suffix array (SSA)







- $O(n \log^2 b)$ time (Monte-Carlo)
- $O((n + b^2) \log^2 b)$ time with high probability (Las-Vegas)
- both in $O(b)$ space

Verifying the sparse suffix array

T	b	a	n	a	n	a	s
Suffix Array	2	4	6	1	3	5	7

n

How can we tell if this suffix array is correct?

Verifying the sparse suffix array

T	b	a	n	a	n	a	s
	$\vdash$			n			$\dashv$
Suffix Array	2	4	6	1	3	5	7

How can we tell if this suffix array is correct?

Check that $\boxed{2} < \boxed{4}\,,\ \boxed{4} < \boxed{6}\,,\ \boxed{6} < \boxed{1}\,,\ \boxed{1} < \boxed{3}\ \ldots$

Verifying the sparse suffix array

T	b	a	n	a	n	a	s
	$\longleftarrow$			n			$\longrightarrow$
Suffix Array	2	4	6	1	3	5	7

How can we tell if this suffix array is correct?

Check that $\boxed{2} < \boxed{4}$, $\boxed{4} < \boxed{6}$, $\boxed{6} < \boxed{1}$, $\boxed{1} < \boxed{3}$ $\ldots$

Verifying the sparse suffix array

T	b	a	n	a	n	a	s
	n						
Suffix Array	2	4	6	1	3	5	7

How can we tell if this suffix array is correct?

Check that $\boxed{2} < \boxed{4}$, $\boxed{4} < \boxed{6}$, $\boxed{6} < \boxed{1}$, $\boxed{1} < \boxed{3}$ $\ldots$

We could check $\boxed{2} < \boxed{4}$ using an LCP query if we verified it

Verifying the sparse suffix array

$$T \quad \boxed{b}\boxed{a}\boxed{n}\boxed{a}\boxed{n}\boxed{a}\boxed{s}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{n}$$

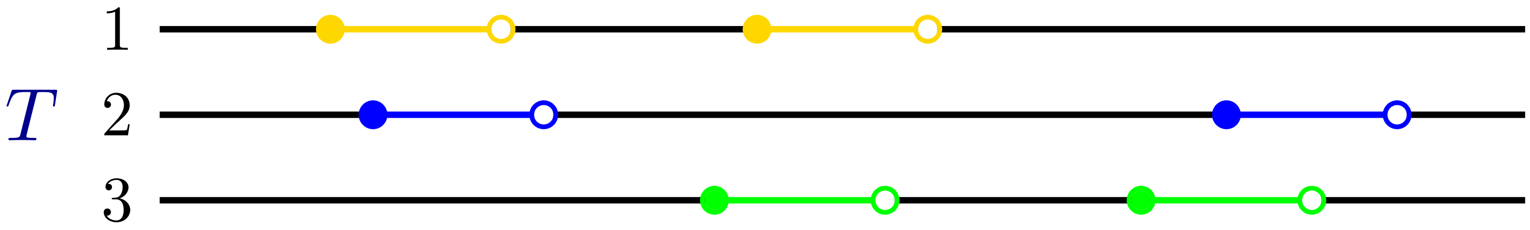
Suffix Array $\boxed{2}\boxed{4}\boxed{6}\boxed{1}\boxed{3}\boxed{5}\boxed{7}$

How can we tell if this suffix array is correct?

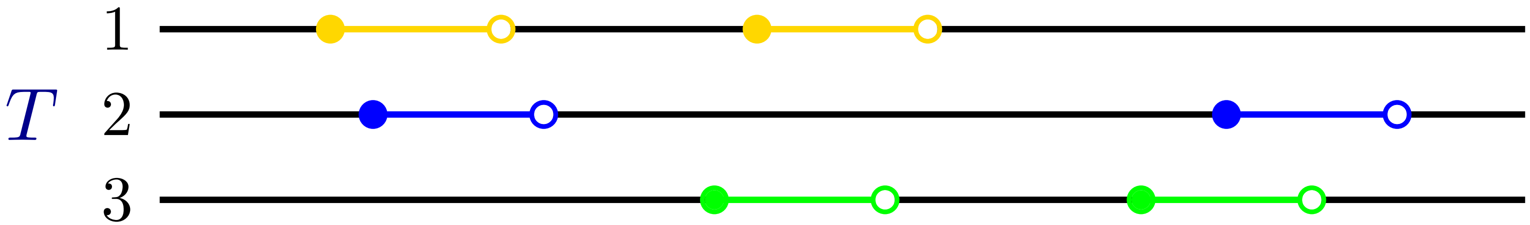
Check that $\boxed{2} < \boxed{4}\ ,\ \boxed{4} < \boxed{6}\ ,\ \boxed{6} < \boxed{1}\ ,\ \boxed{1} < \boxed{3}\ \ldots$

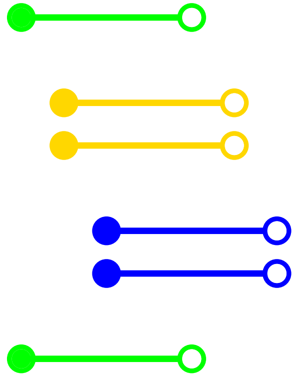
We could check $\boxed{2} < \boxed{4}$ using an LCP query if we verified it

A first example

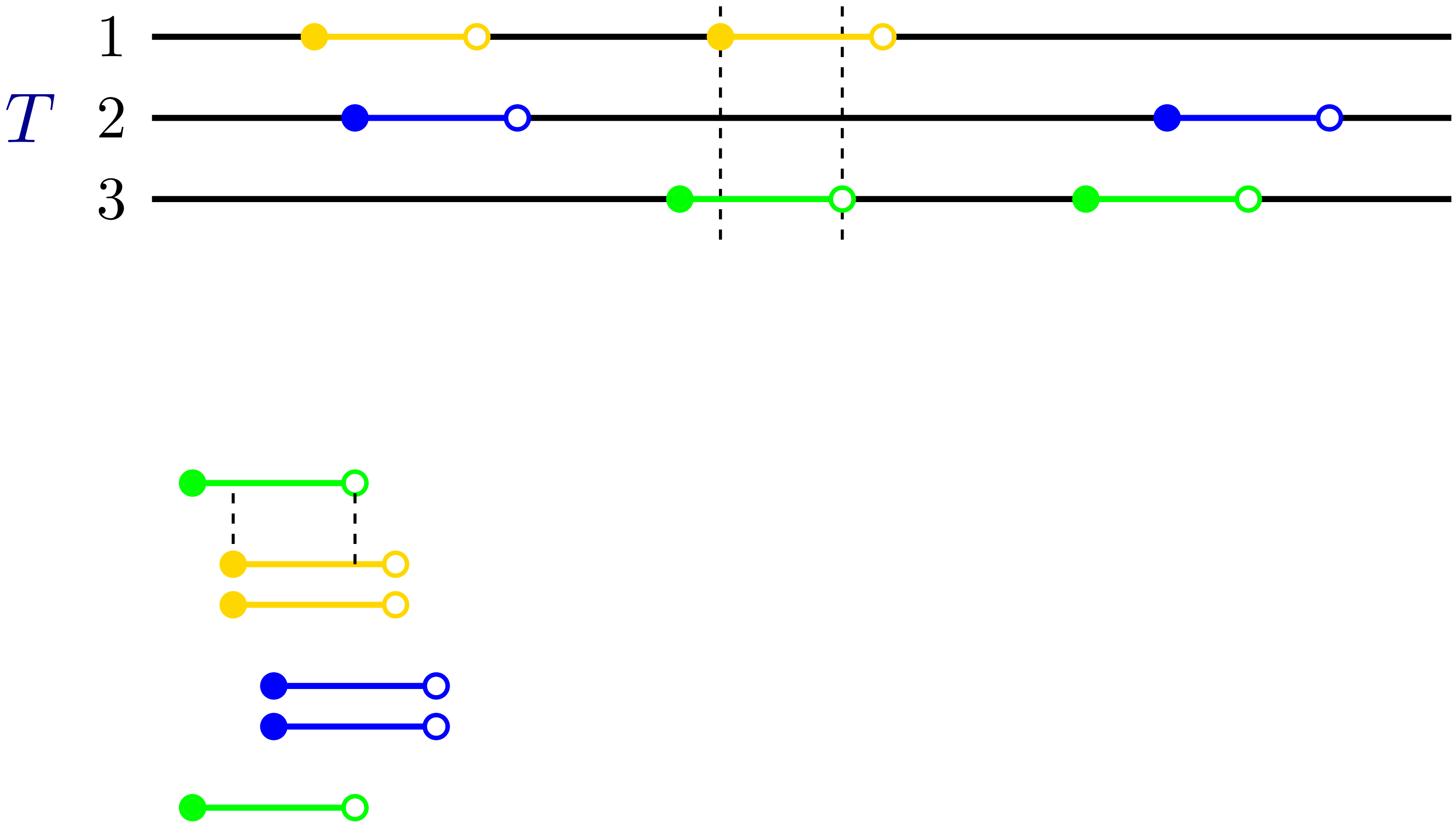


A first example

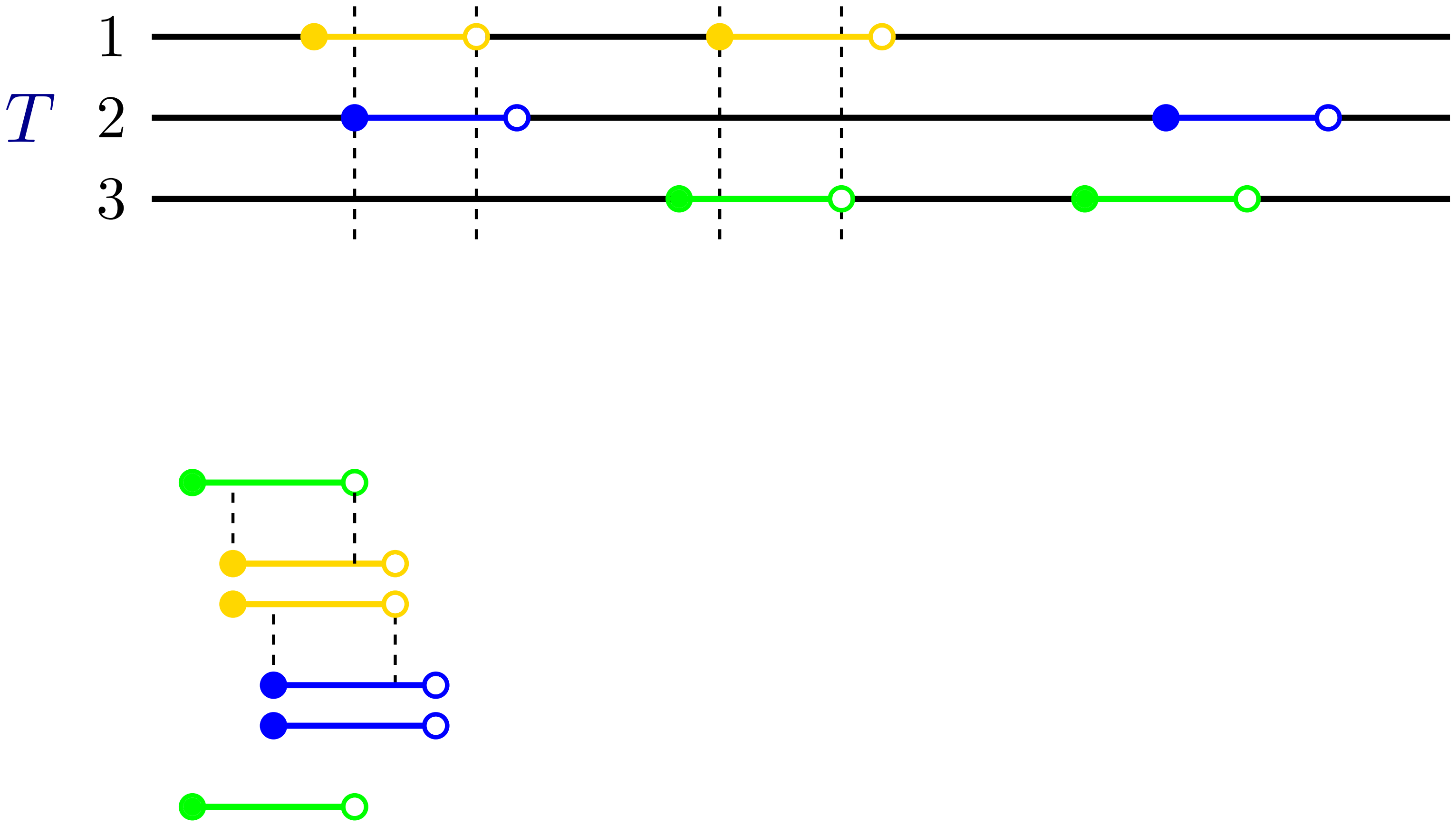




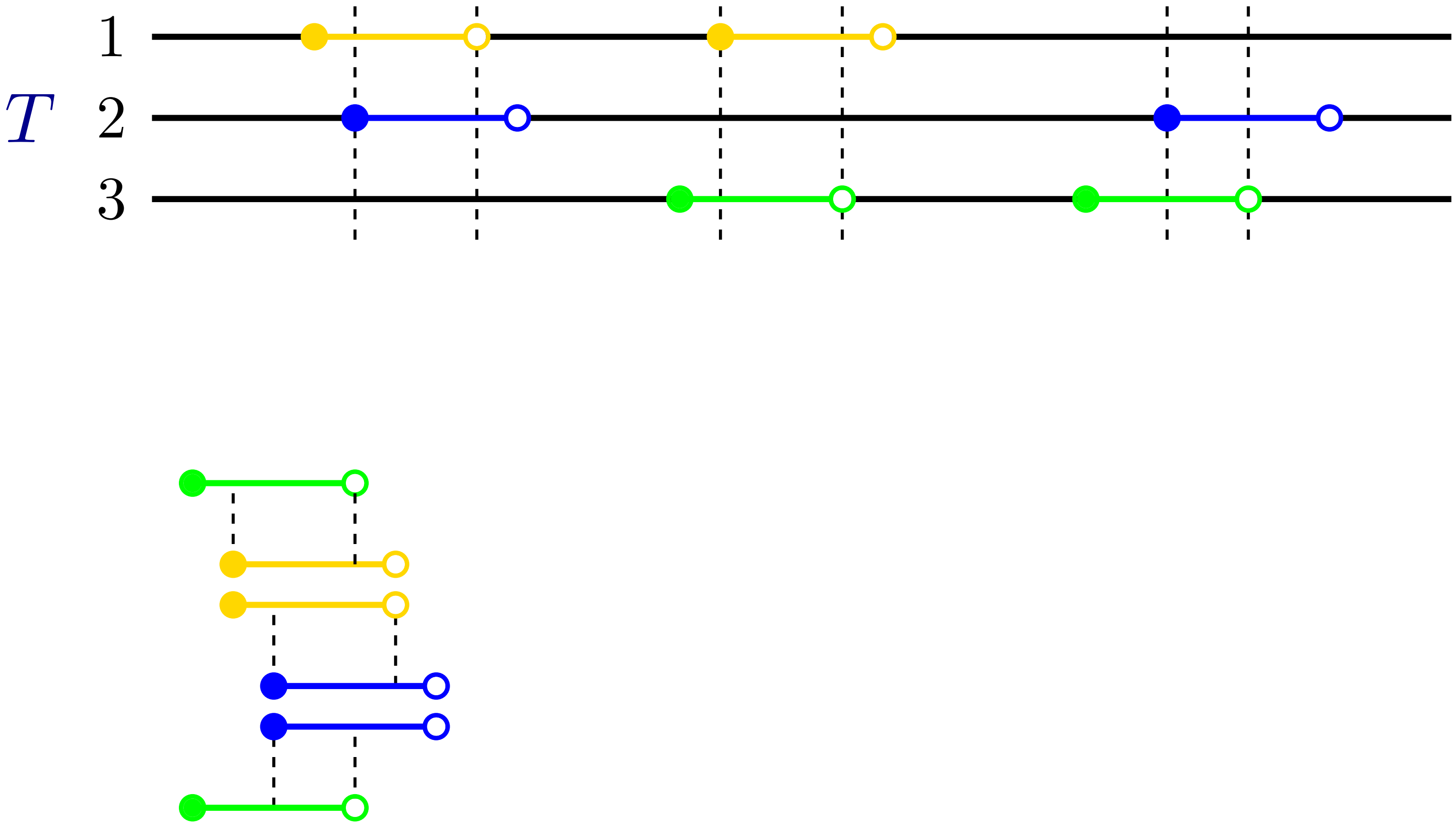
A first example



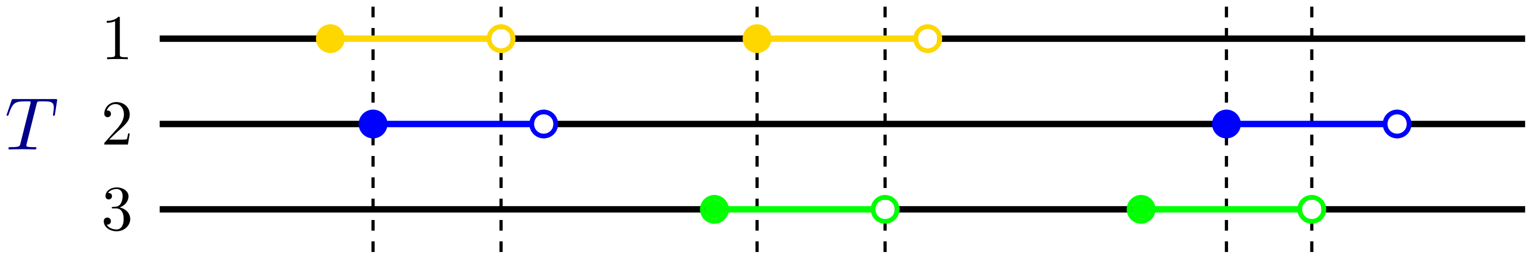
A first example

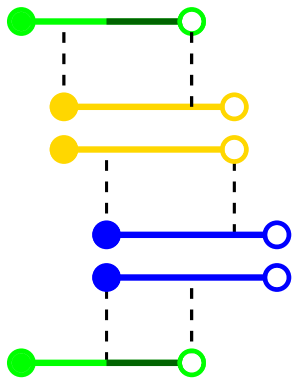


A first example



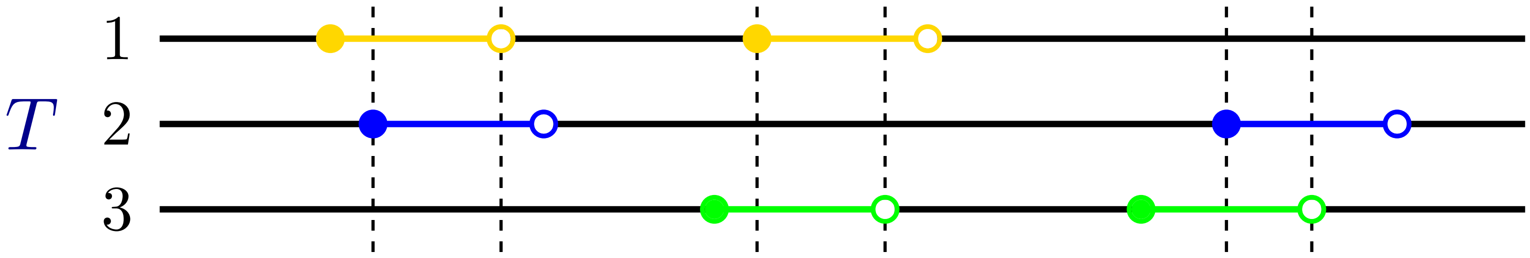
A first example

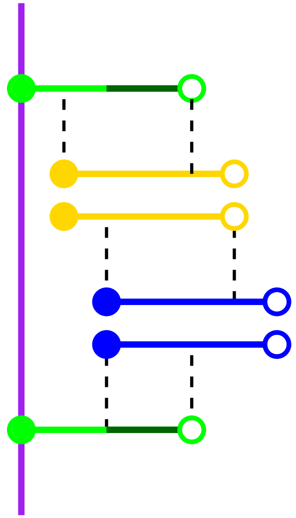




If yellow (1) and blue (2) match then the right half of green (3) matches

A first example

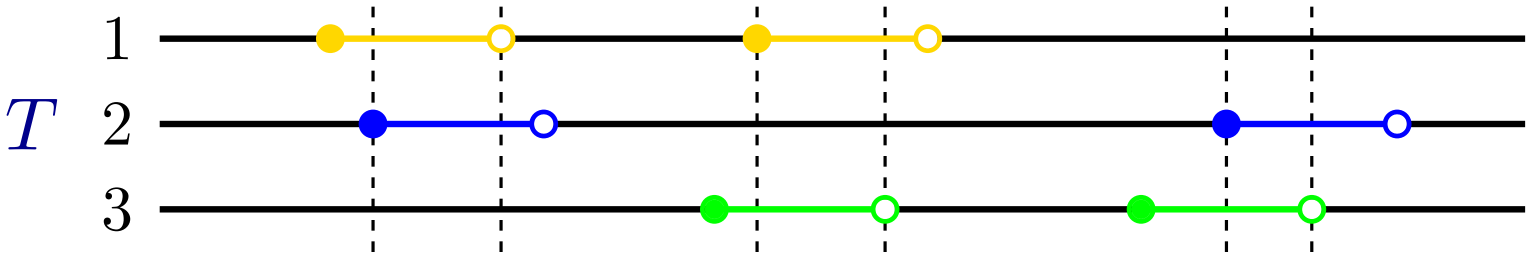


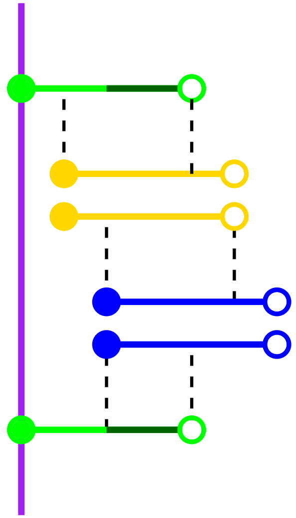


If yellow (1) and blue (2) match then the right half of green (3) matches

This is a *lock-stepped* cycle

A first example

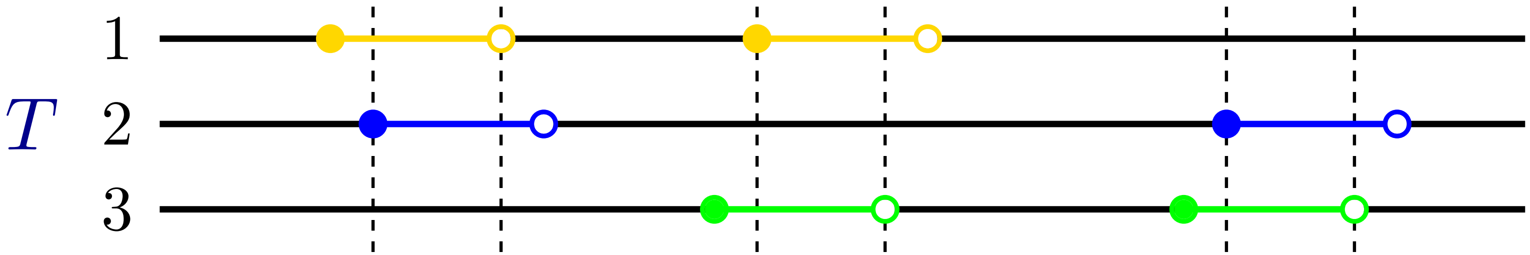


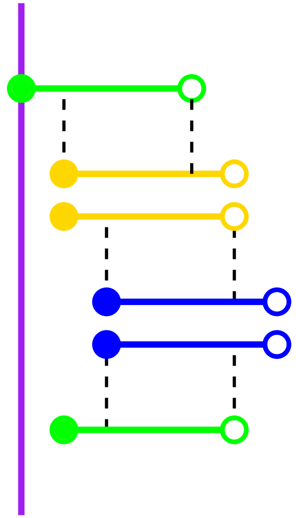


If yellow (1) and blue (2) match then the right half of green (3) matches

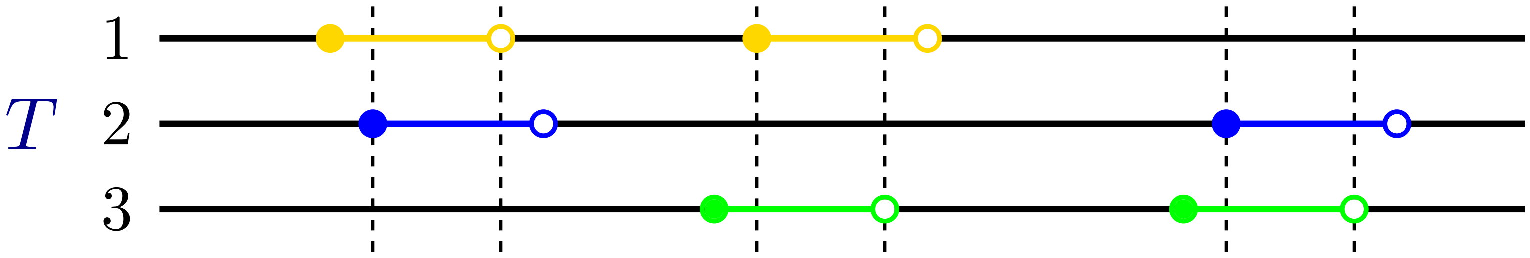
This is a *lock-stepped* cycle

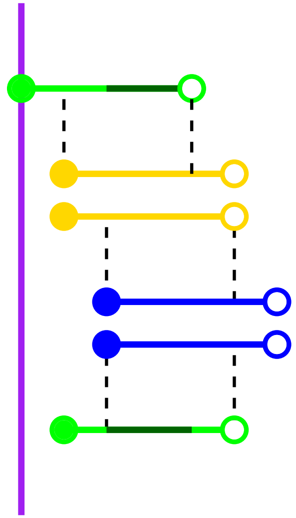
A second example





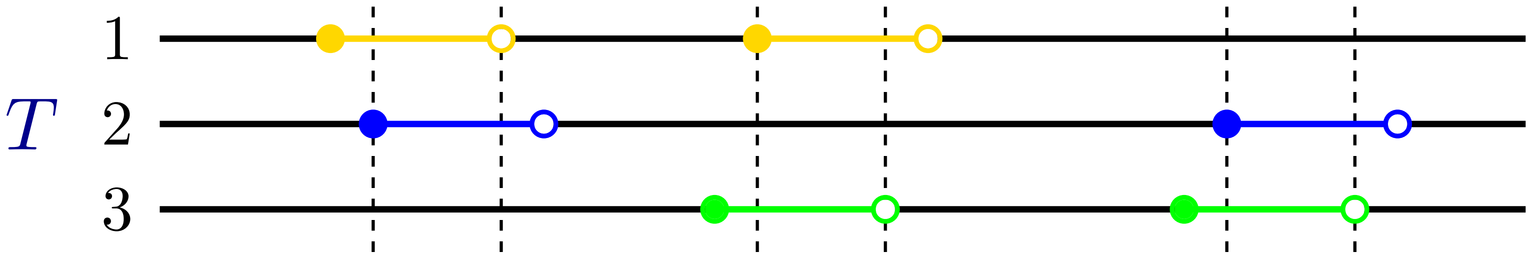
A second example

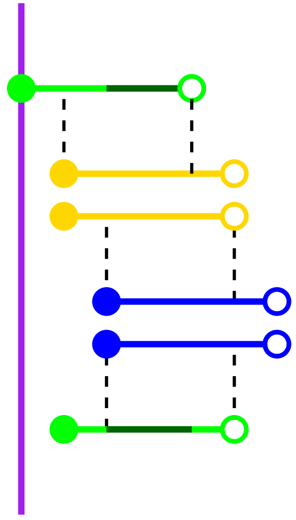




If yellow (1),blue (2) and green (3) match then $\frac{3}{4}$ of green (3) is periodic

A second example

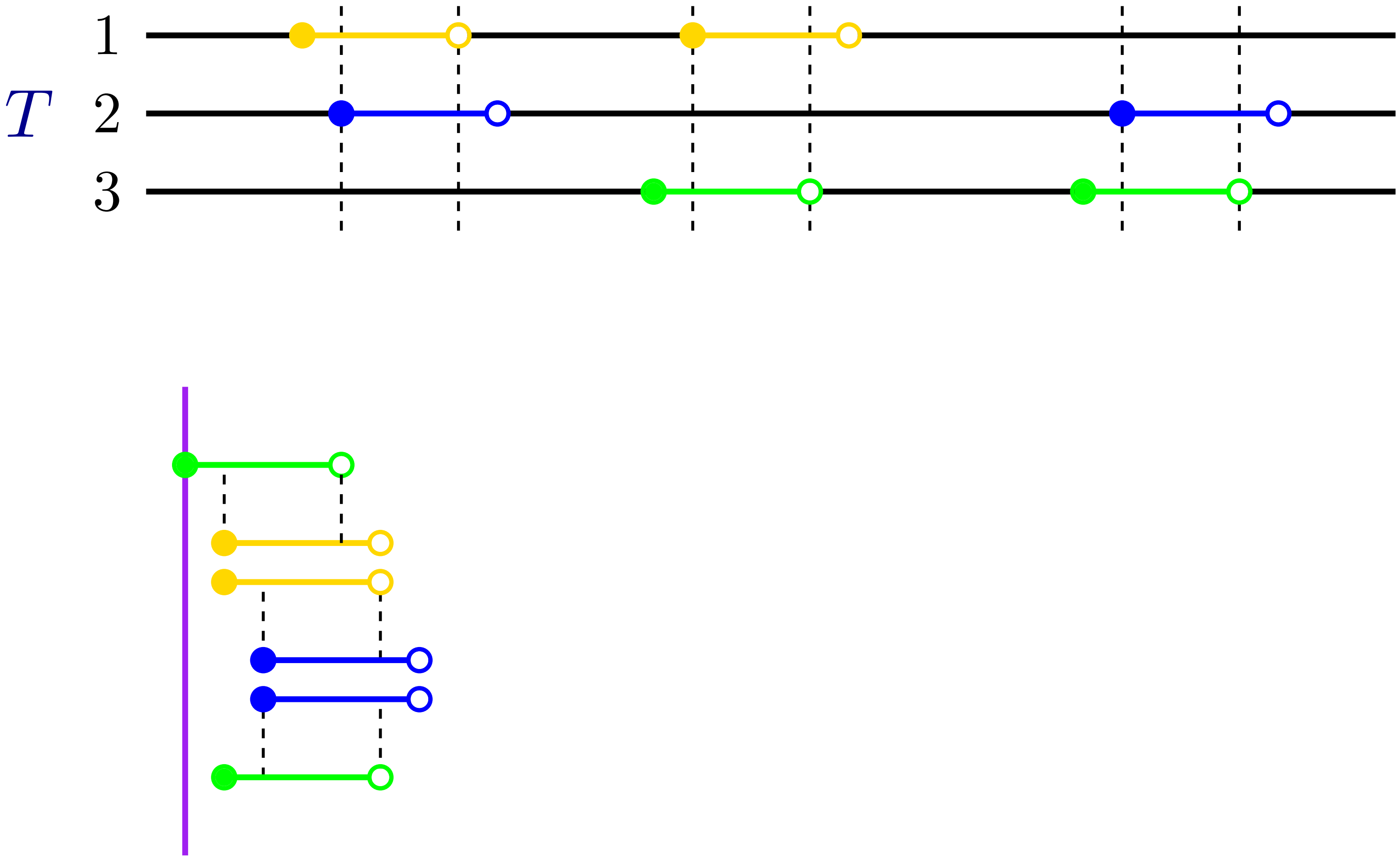




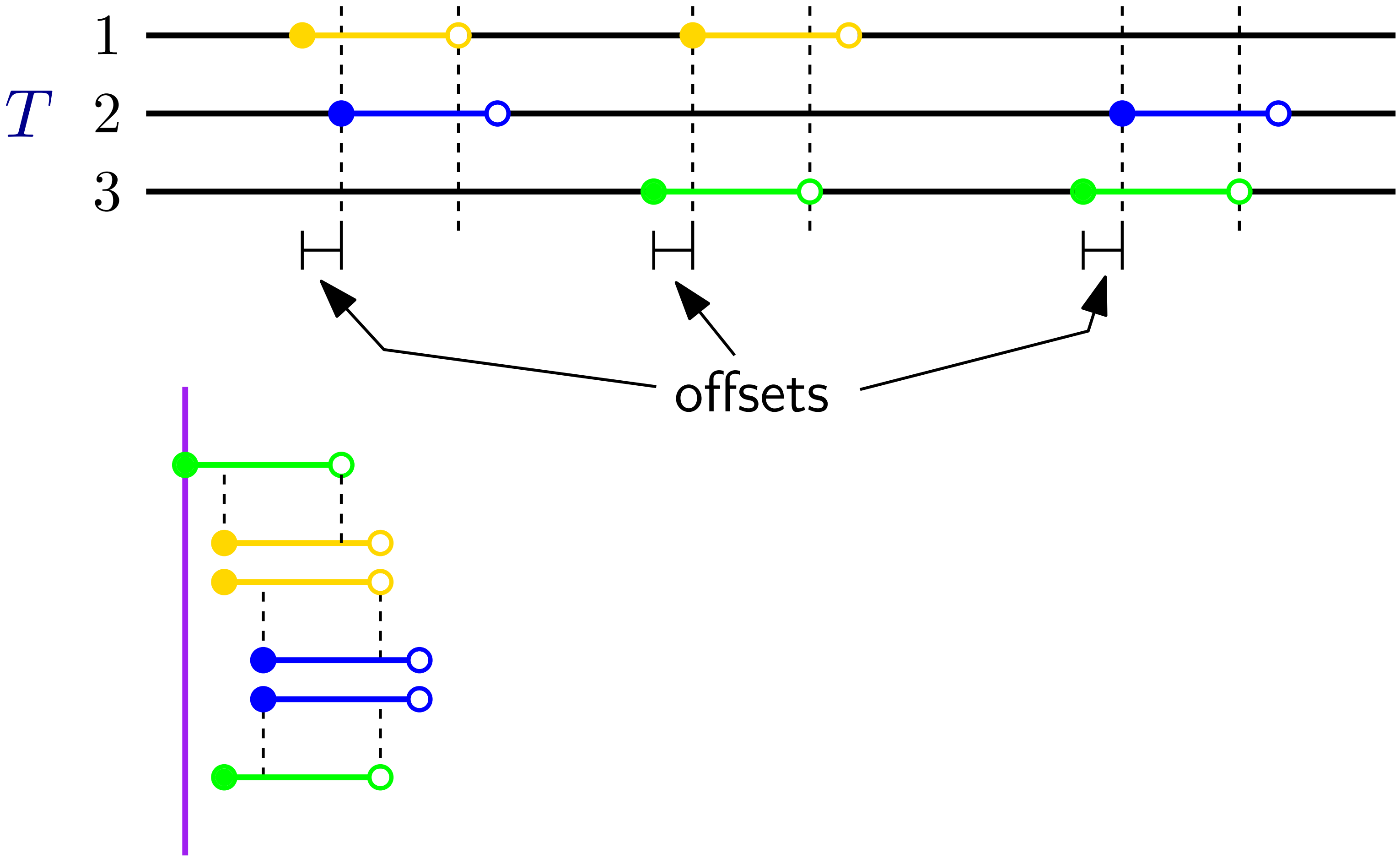
If yellow (1), blue (2) and green (3) match then $\frac{3}{4}$ of green (3) is periodic

This is an *unlocked* cycle

A second example

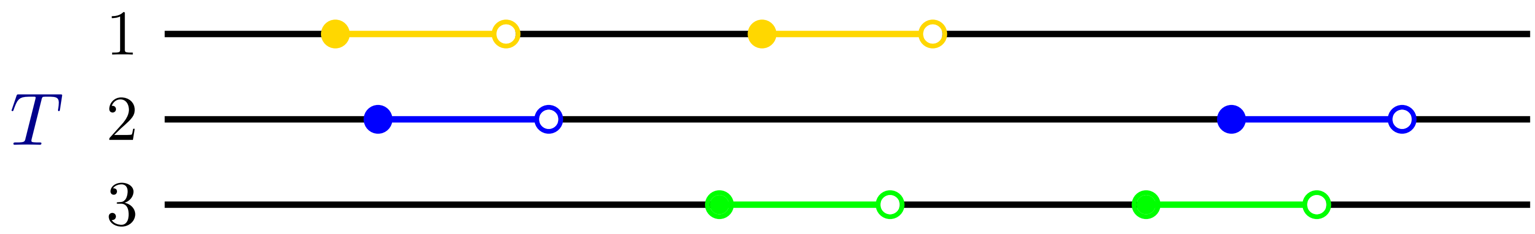


A second example



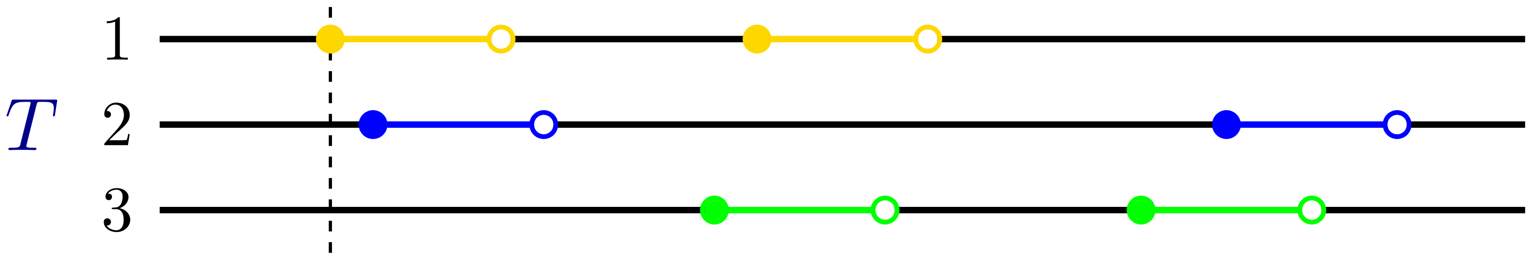
These tricks only work when the offsets are *small*

The overall idea

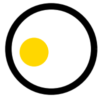


- We build a graph which encodes the structure of the queries

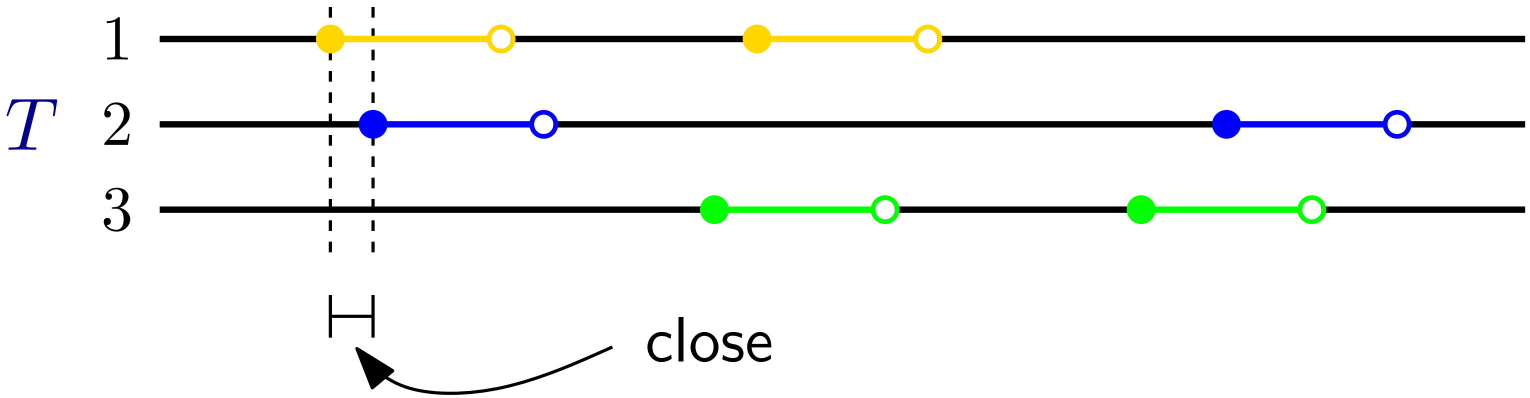
The overall idea



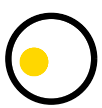
- We build a graph which encodes the structure of the queries



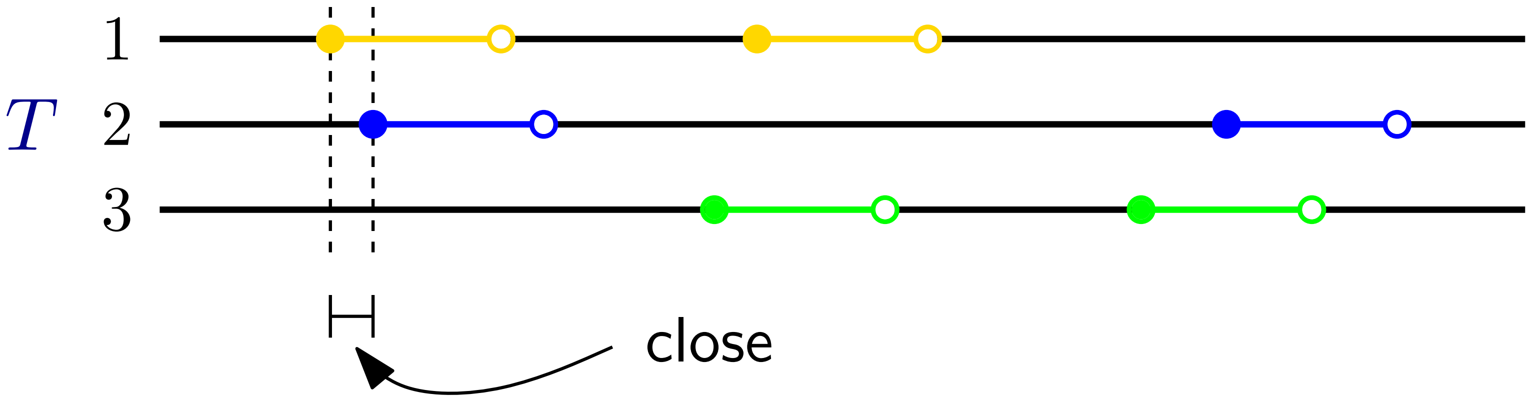
The overall idea



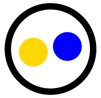
- We build a graph which encodes the structure of the queries



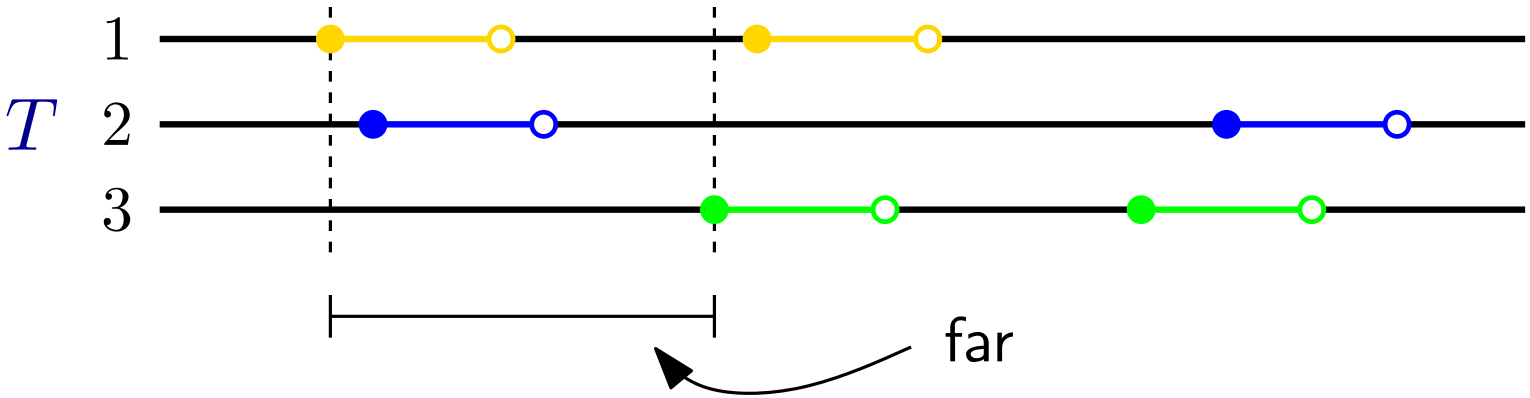
The overall idea



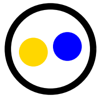
- We build a graph which encodes the structure of the queries



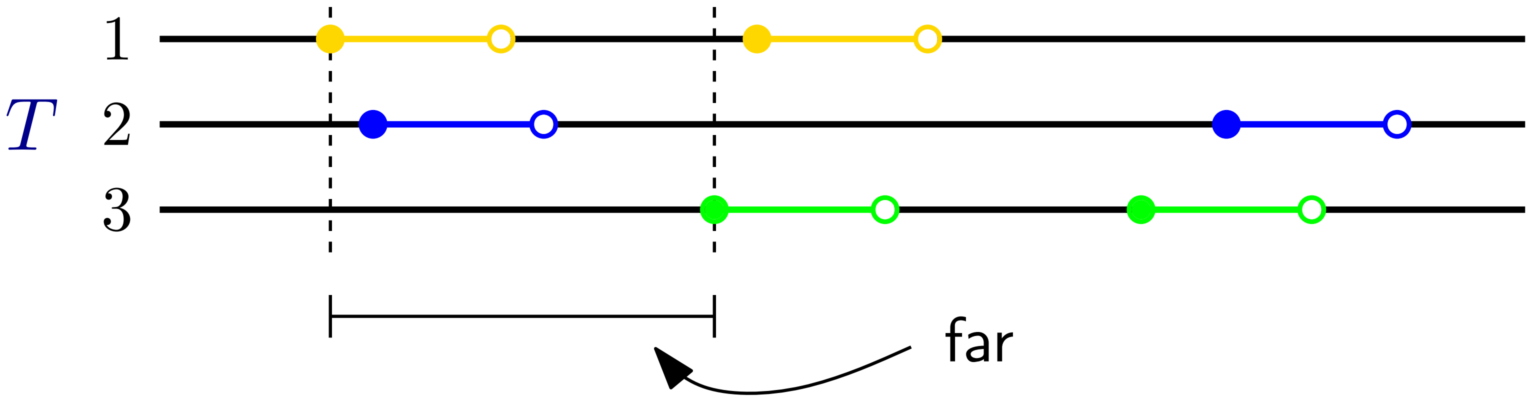
The overall idea



- We build a graph which encodes the structure of the queries

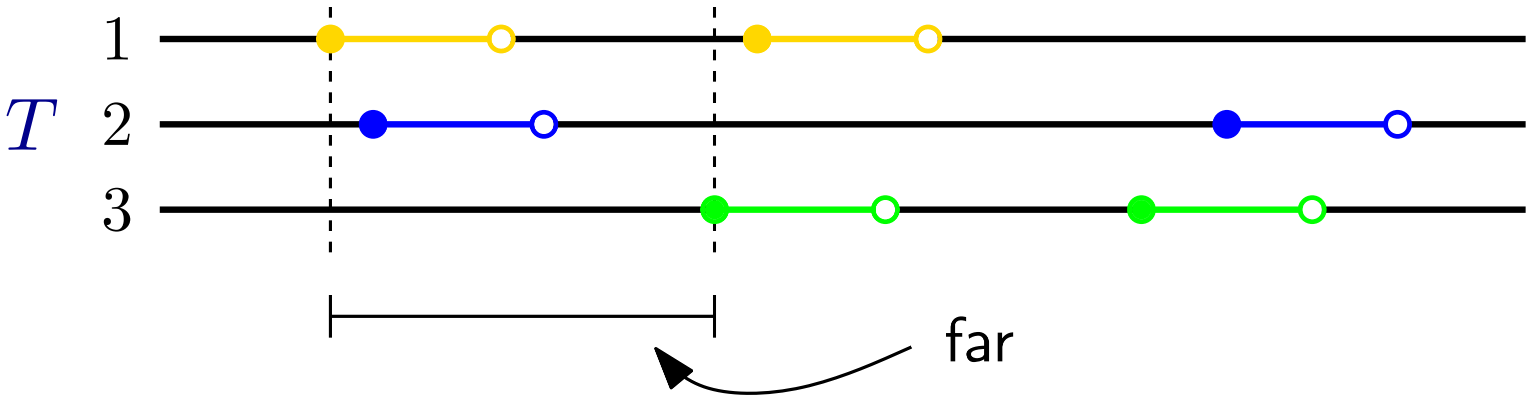


The overall idea



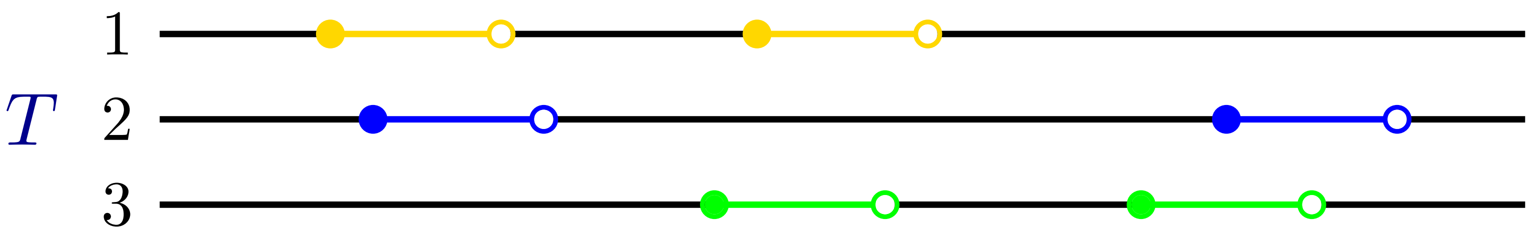
- We build a graph which encodes the structure of the queries

The overall idea

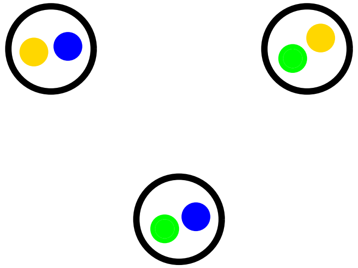


- We build a graph which encodes the structure of the queries

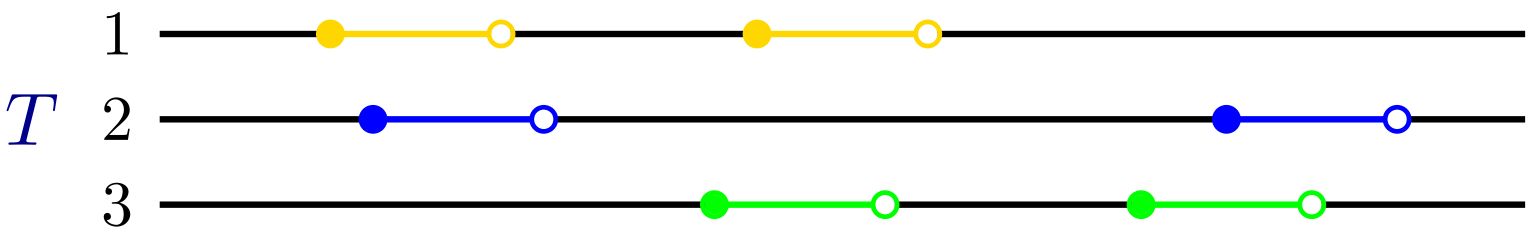
The overall idea



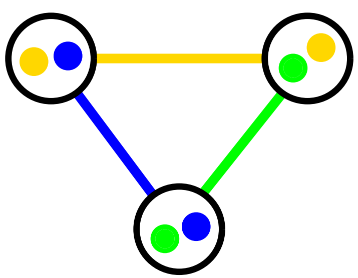
- We build a graph which encodes the structure of the queries



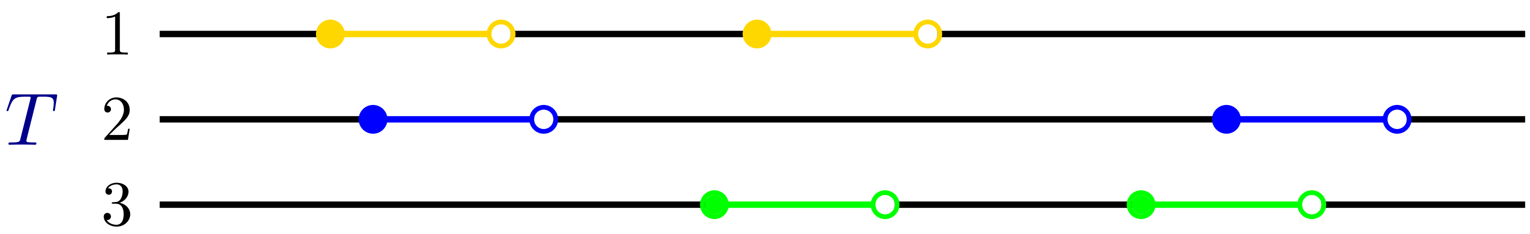
The overall idea



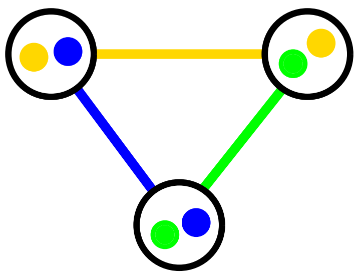
- We build a graph which encodes the structure of the queries



The overall idea

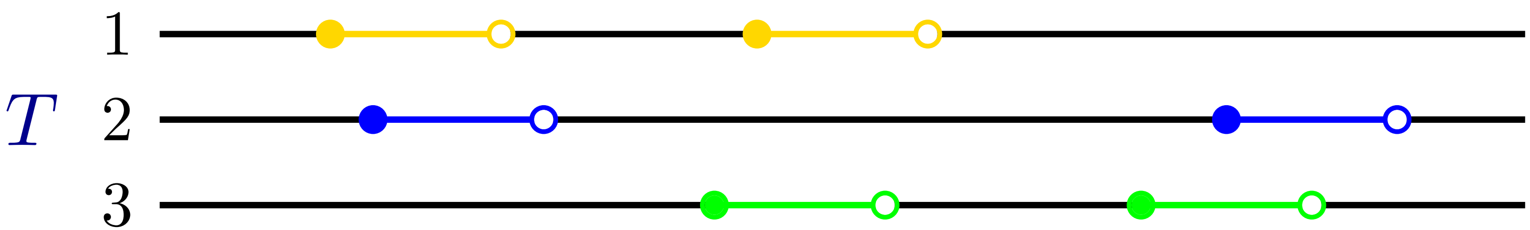


- We build a graph which encodes the structure of the queries

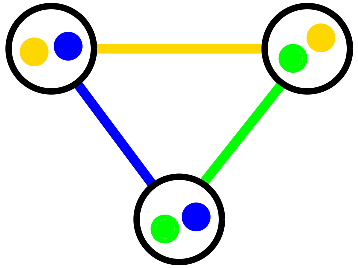


- We can apply one of the two tricks to any *short* cycle

The overall idea

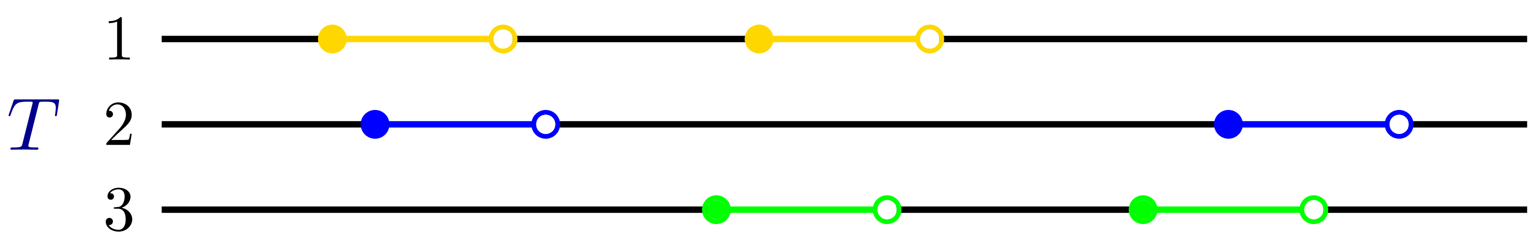


- We build a graph which encodes the structure of the queries

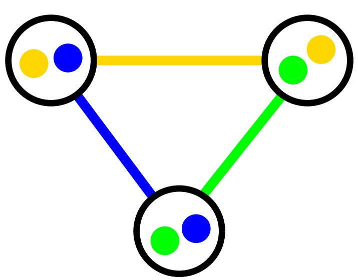


- We can apply one of the two tricks to any *short* cycle (length at most $2\log b + 1$)

The overall idea



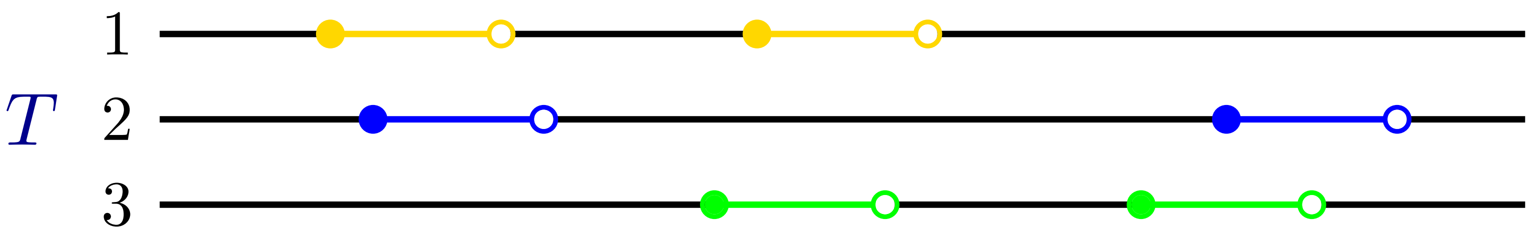
- We build a graph which encodes the structure of the queries



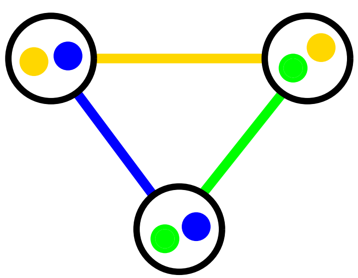
- We can apply one of the two tricks
  to any *short* cycle (length at most $2\log b + 1$)

- This breaks the cycle (because we delete an edge)

The overall idea



- We build a graph which encodes the structure of the queries

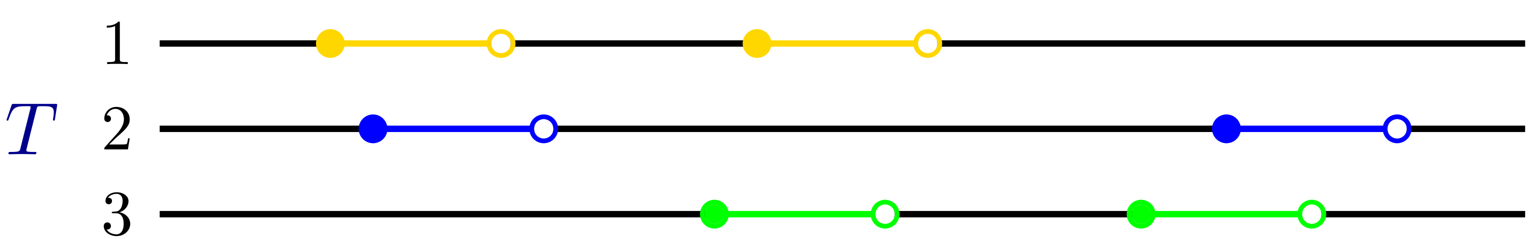


- We can apply one of the two tricks
  to any *short* cycle (length at most $2\log b + 1$)

- This breaks the cycle (because we delete an edge)

Fact If every node has degree at least three there is a short cycle

The overall idea

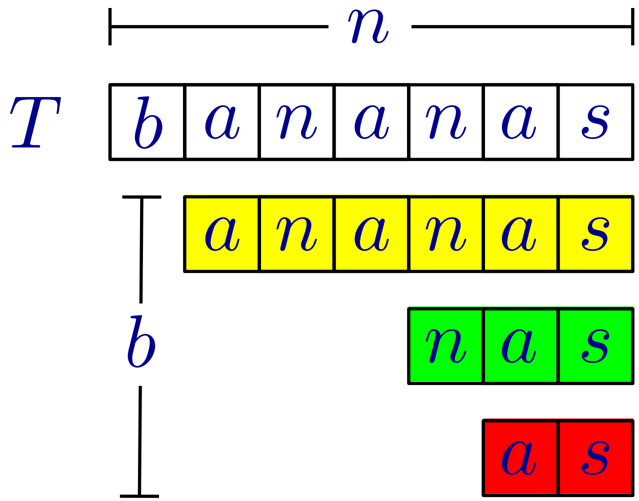


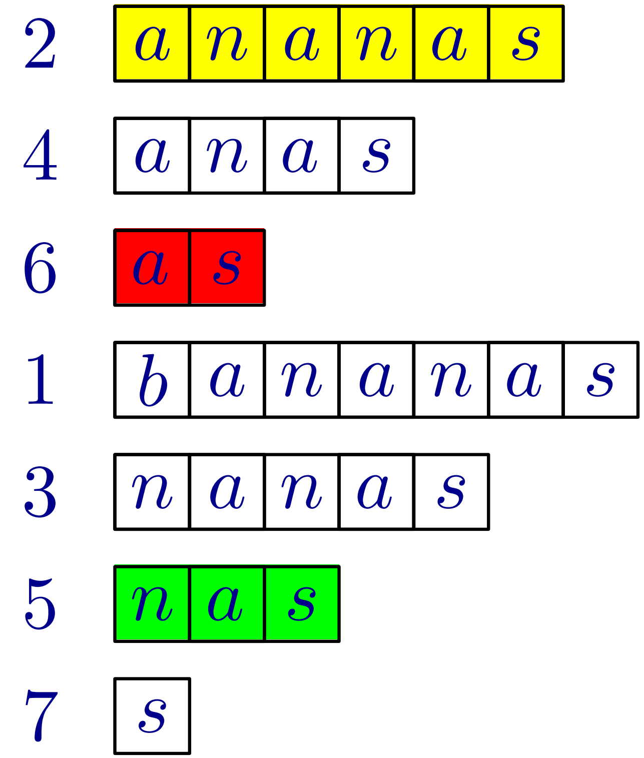
- We build a graph which encodes the structure of the queries

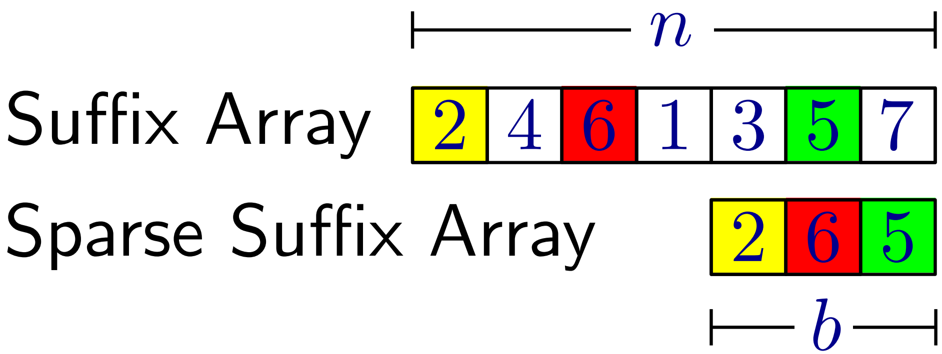
Fact If every node has degree at least three there is a short cycle

- Finding a short cycle in the graph takes $O(b)$ time
- This gives the additive $O(b^2 \log b)$ term
- All other steps take $O(n \log b)$ time over all rounds
  (and use $O(b)$ space)

Summary







- $O(n \log^2 b)$ time (Monte-Carlo)
- $O((n + b^2) \log^2 b)$ time with high probability (Las-Vegas)
- both in $O(b)$ space