Sparse Suffix Tree Construction in Small Space

Benjamin Sach

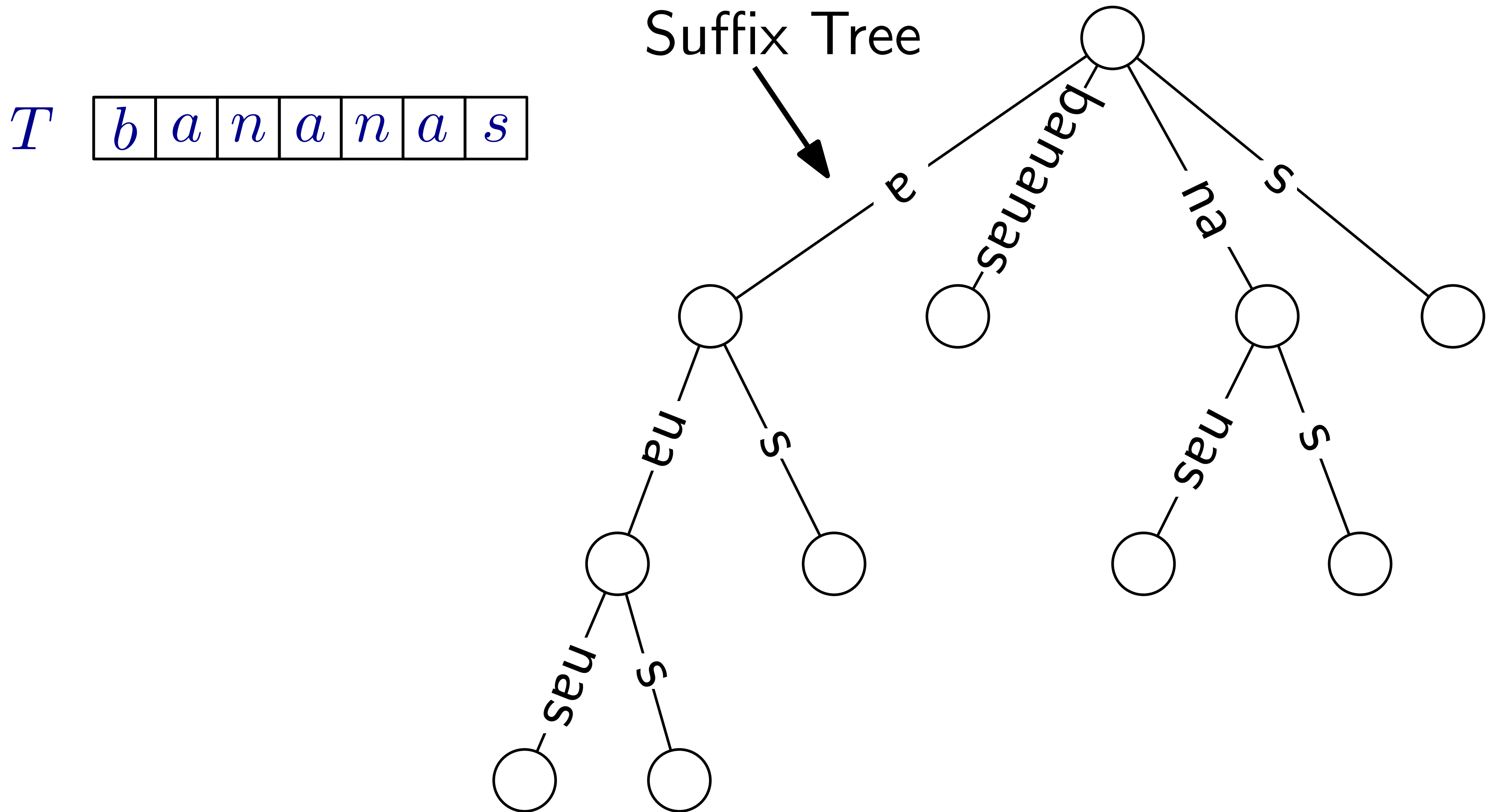
Joint work with

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

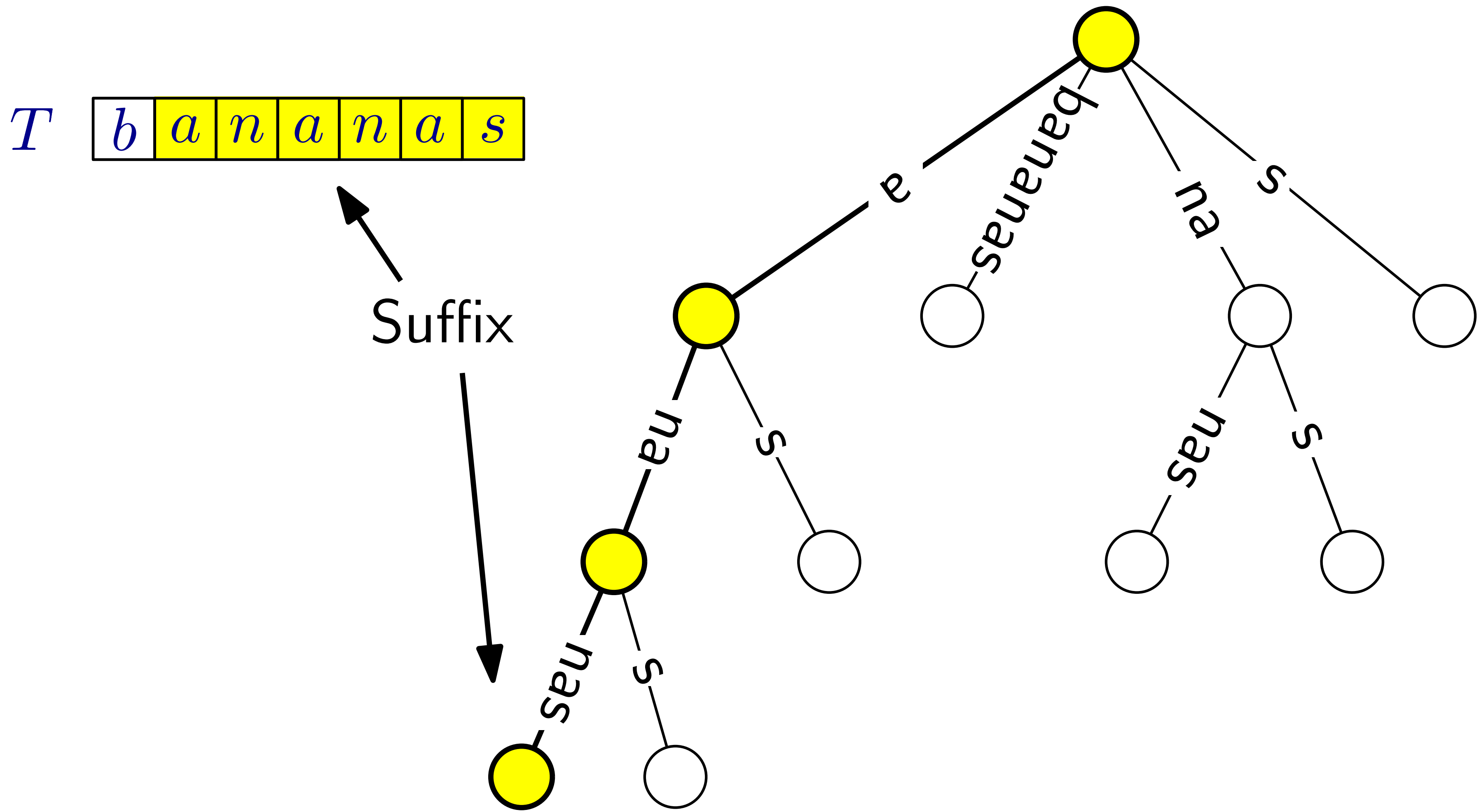
THE UNIVERSITY OF
WARWICK

to appear in ICALP 2013

The sparse suffix tree (SST)

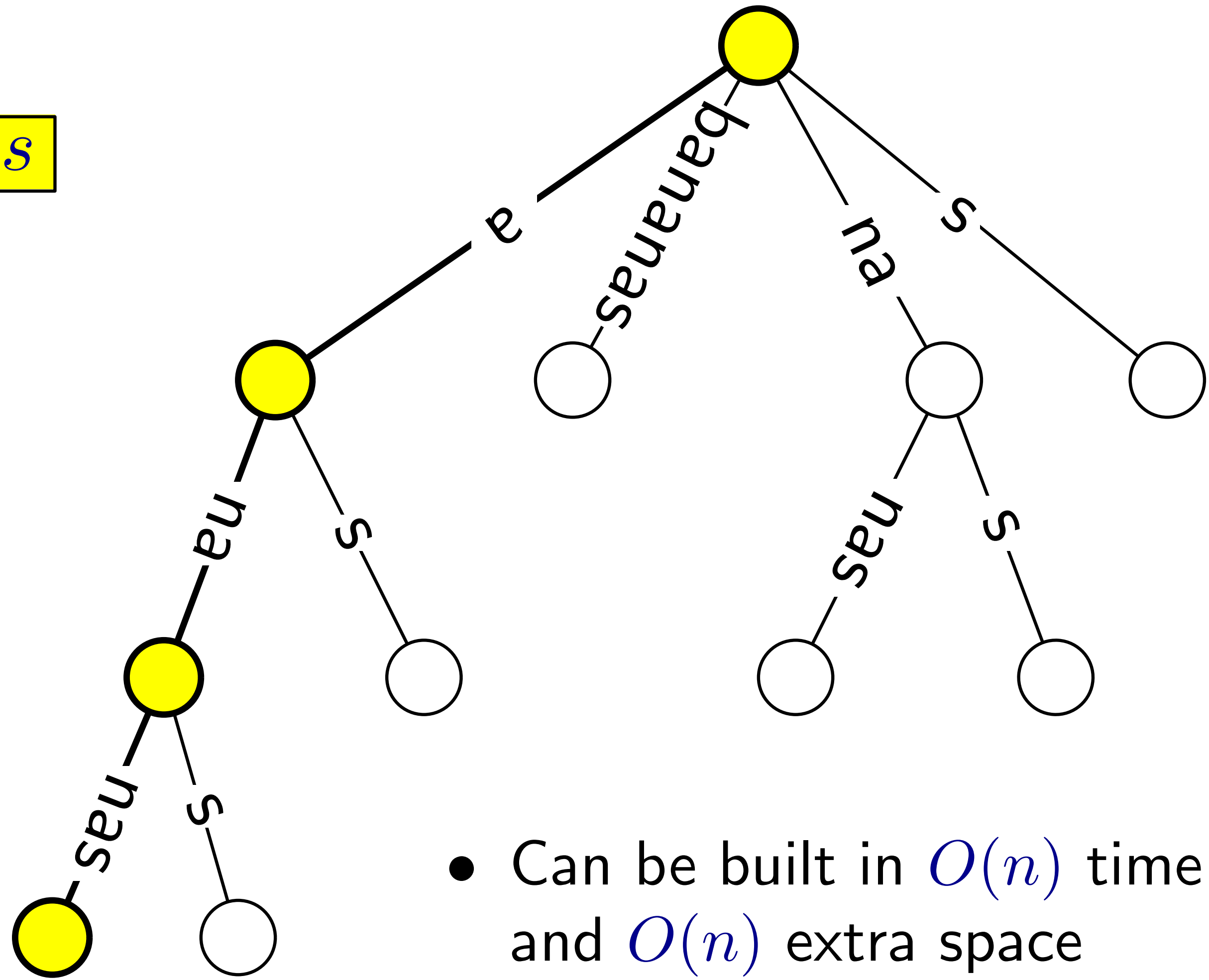


The sparse suffix tree (SST)



The sparse suffix tree (SST)

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

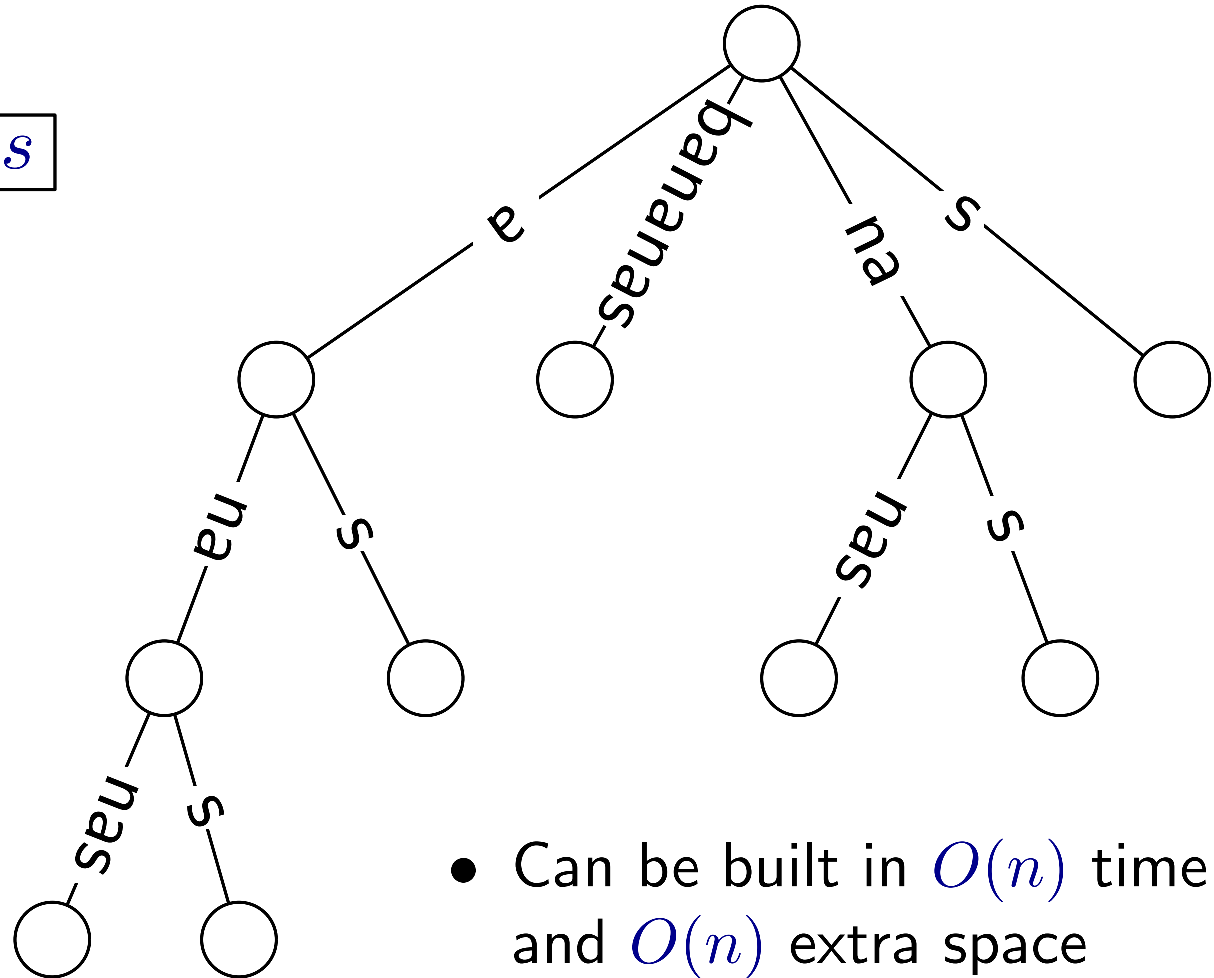


- Can be built in $O(n)$ time and $O(n)$ extra space

The sparse suffix tree (SST)

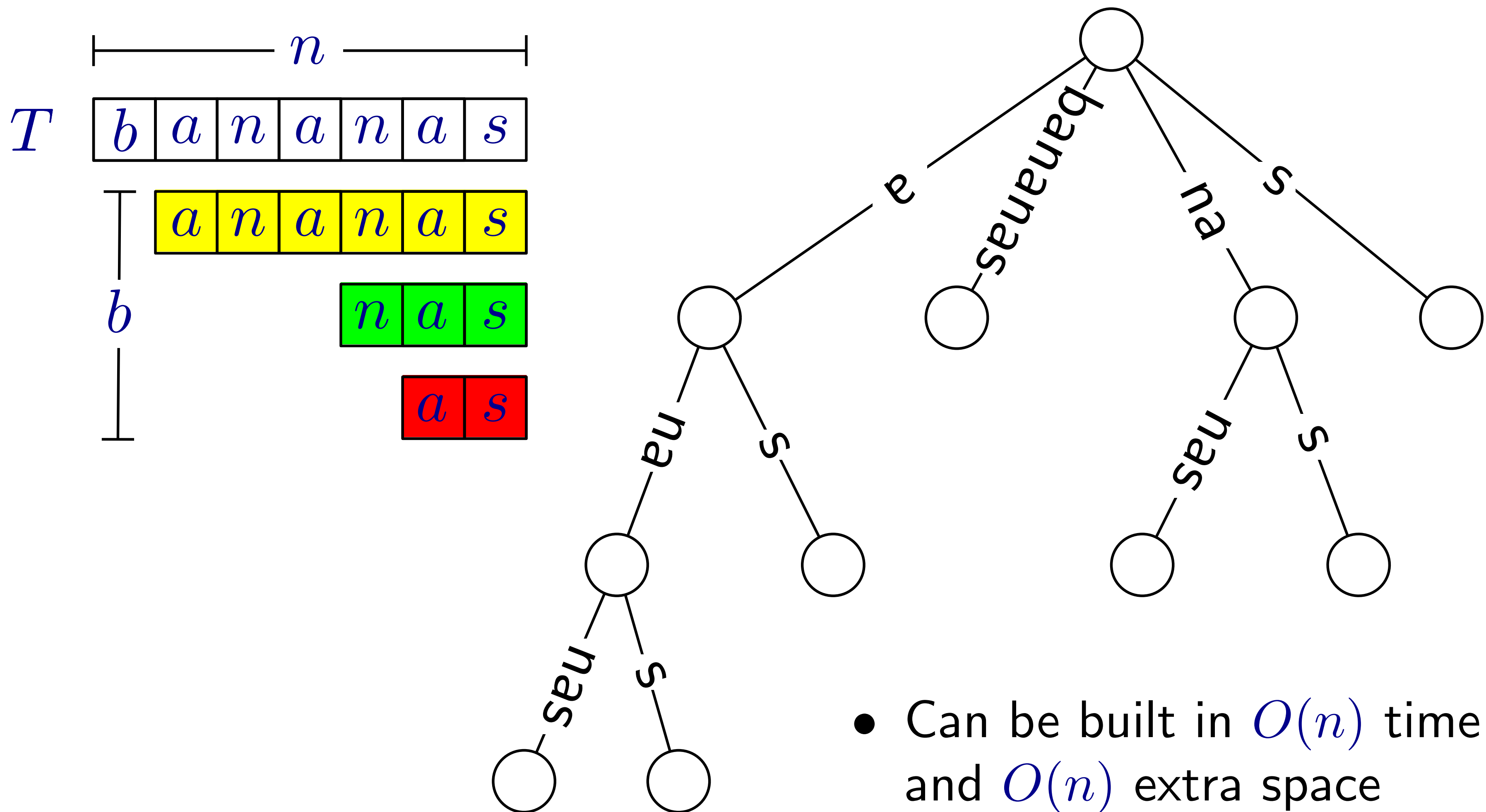
T

b	a	n	a	n	a	s



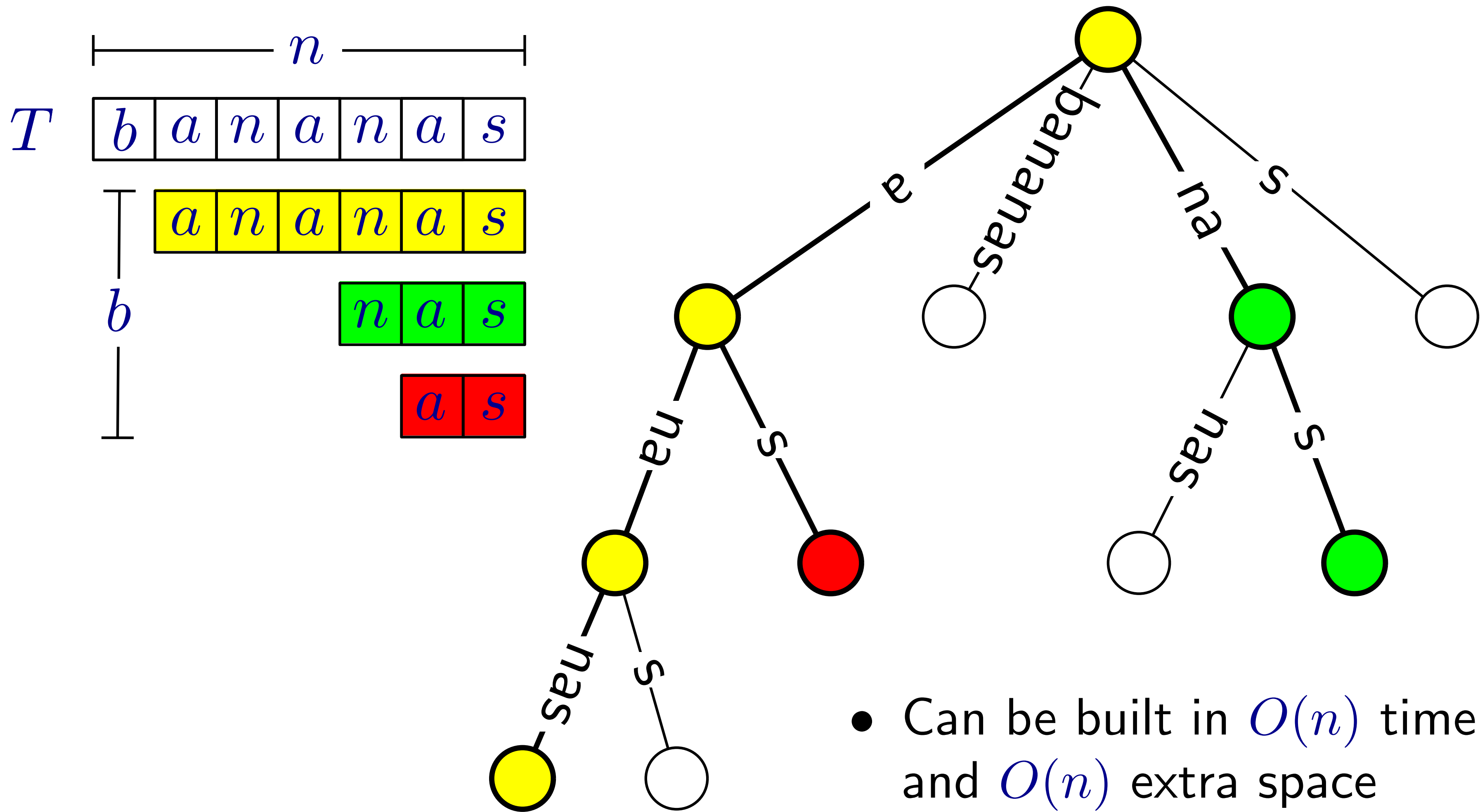
- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix tree (SST)



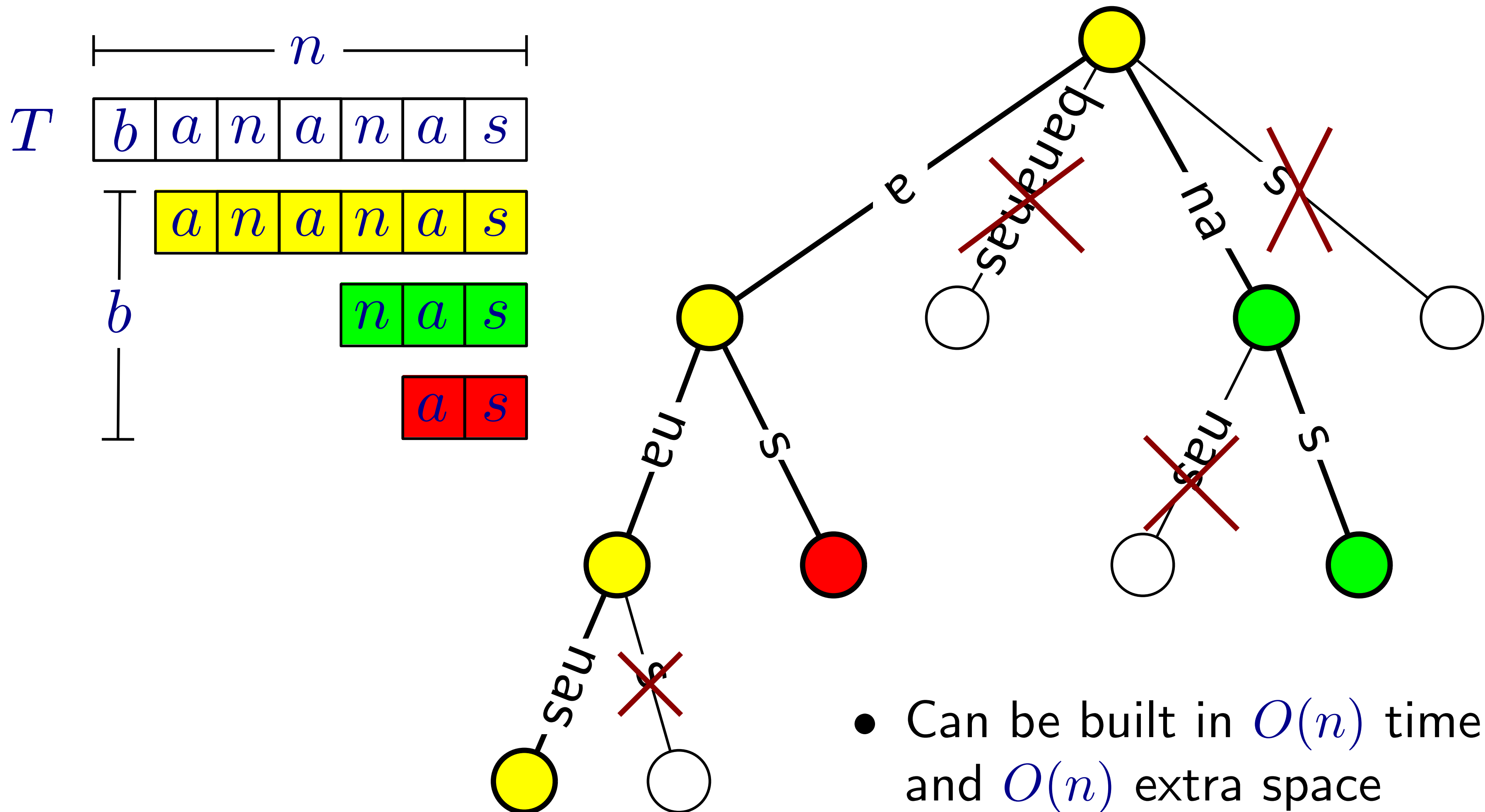
- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix tree (SST)



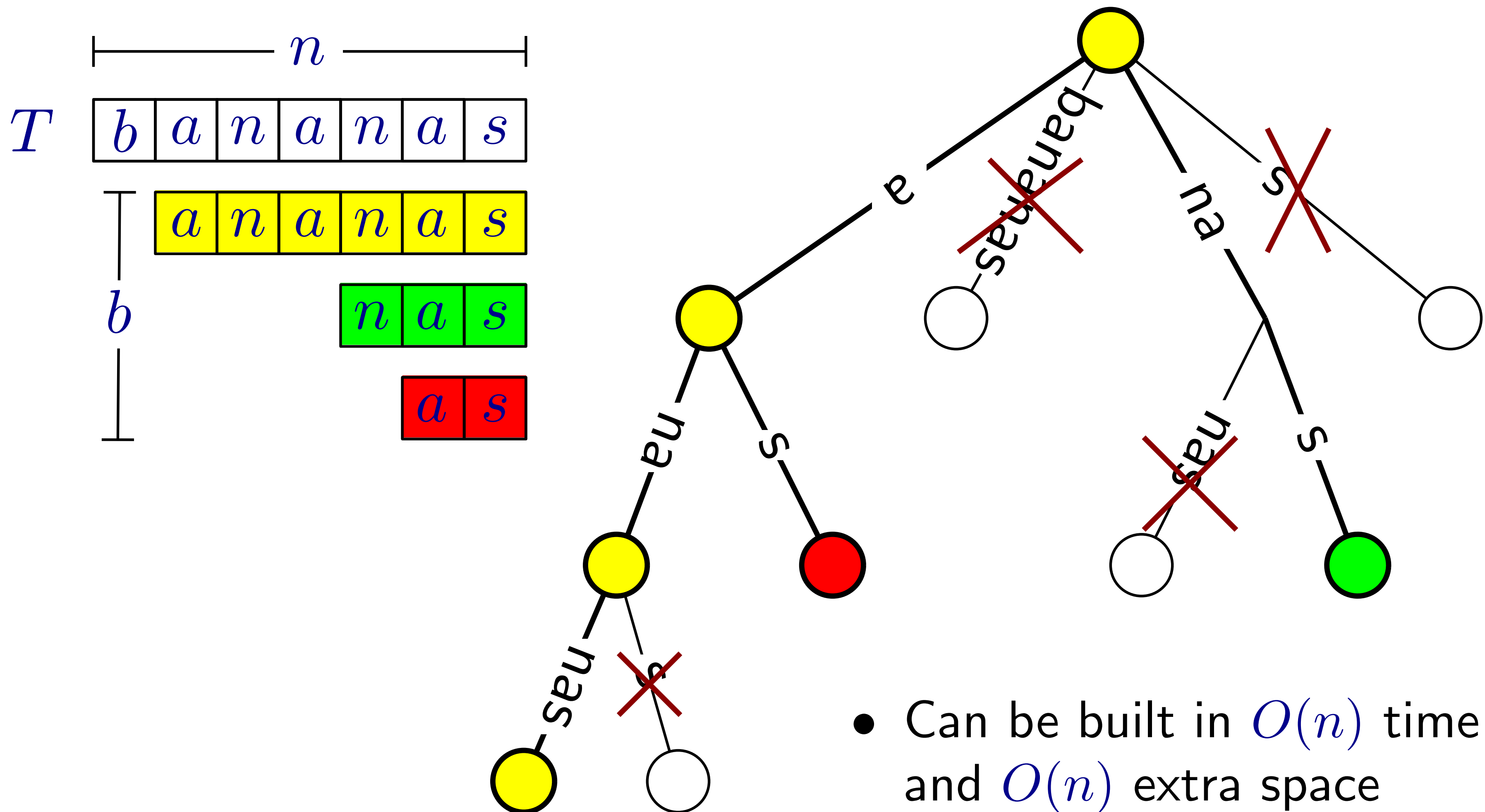
- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix tree (SST)



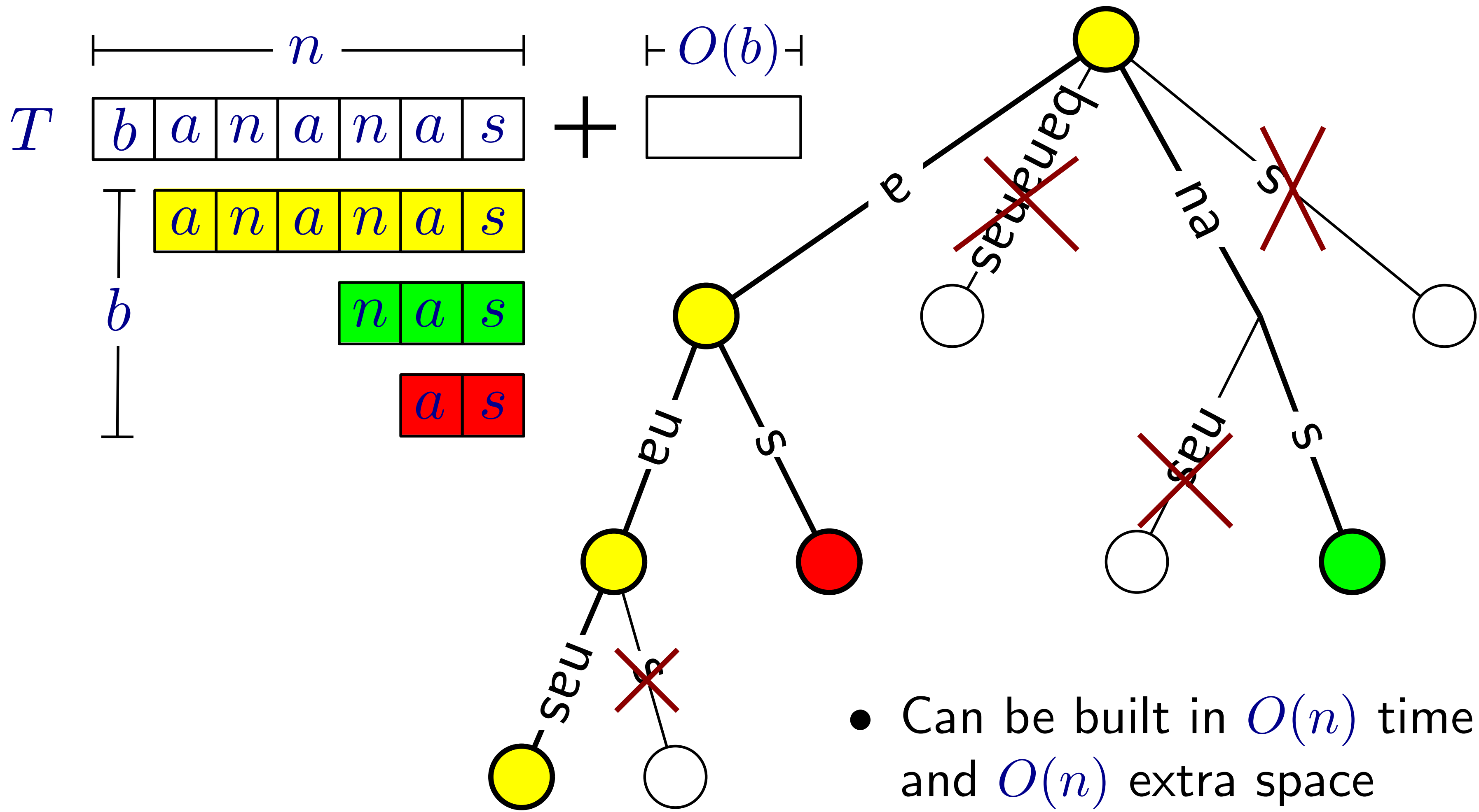
- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix tree (SST)



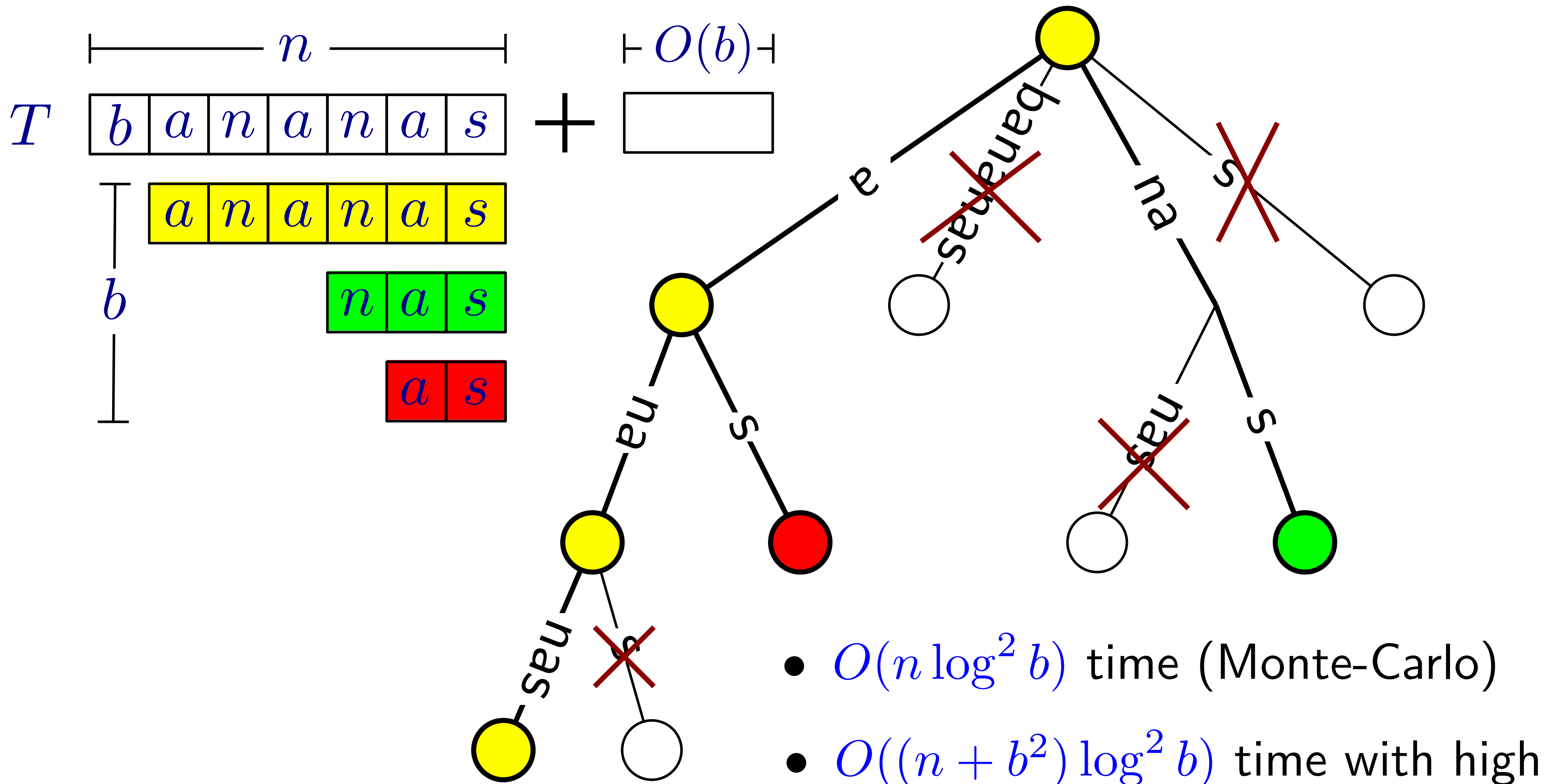
- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix tree (SST)



- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

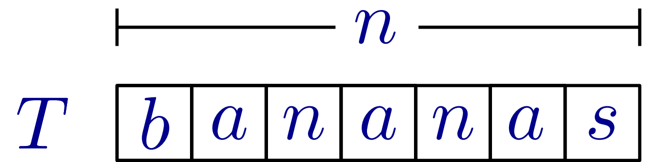
The sparse suffix tree (SST)



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

- Beating the naive bounds has been open since 1960s

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

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

1 $bananas$
2 $ananas$
3 $nanas$
4 $anas$
5 nas
6 as
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

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 |

The sparse suffix array (SSA)

T = | b | a | n | a | n | a | s | (length n)

Sort the suffixes lexicographically

2	$ananas$
4	$anas$
6	as
1	$bananas$
3	$nanas$
5	nas
7	s

Suffix Array (length n)

2	4	6	1	3	5	7

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

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

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)

T

b	a	n	a	n	a	s

(n)

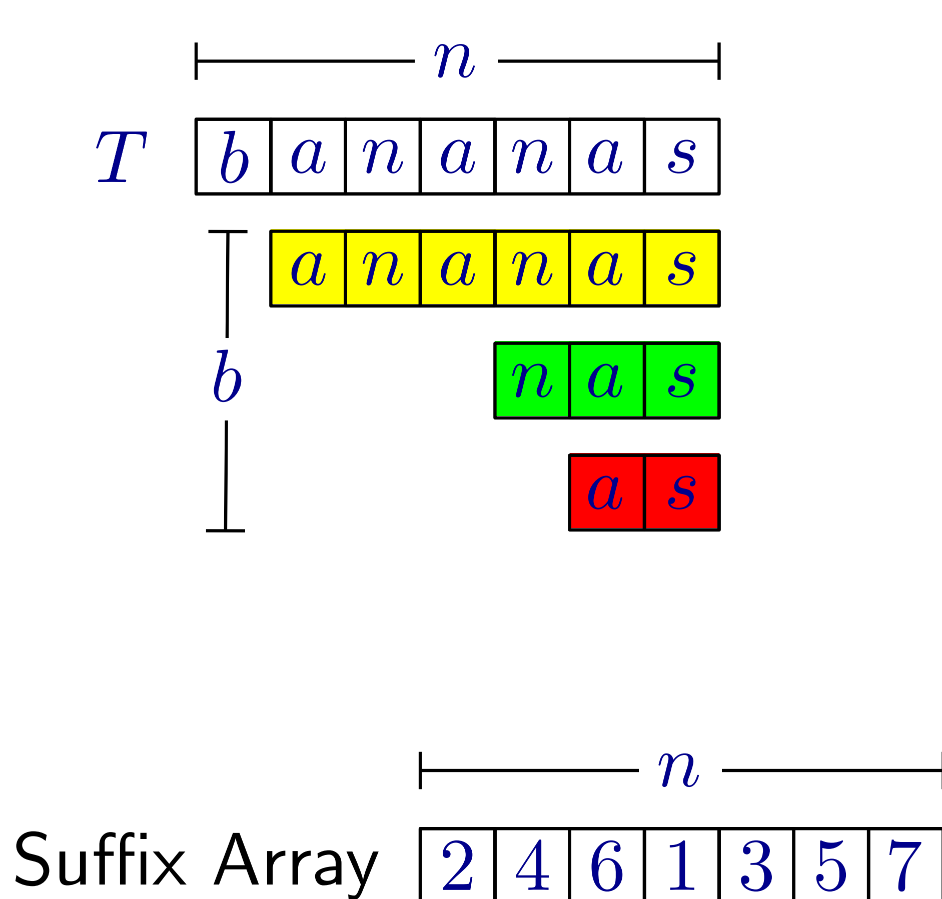
2	$a\,n\,a\,n\,a\,s$
4	$a\,n\,a\,s$
6	$a\,s$
1	$b\,a\,n\,a\,n\,a\,s$
3	$n\,a\,n\,a\,s$
5	$n\,a\,s$
7	s

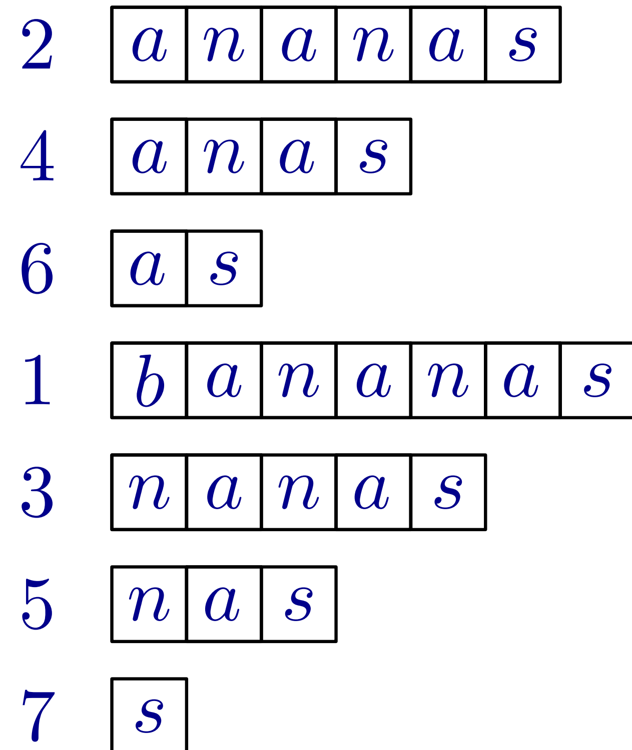
Suffix Array (n)

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?

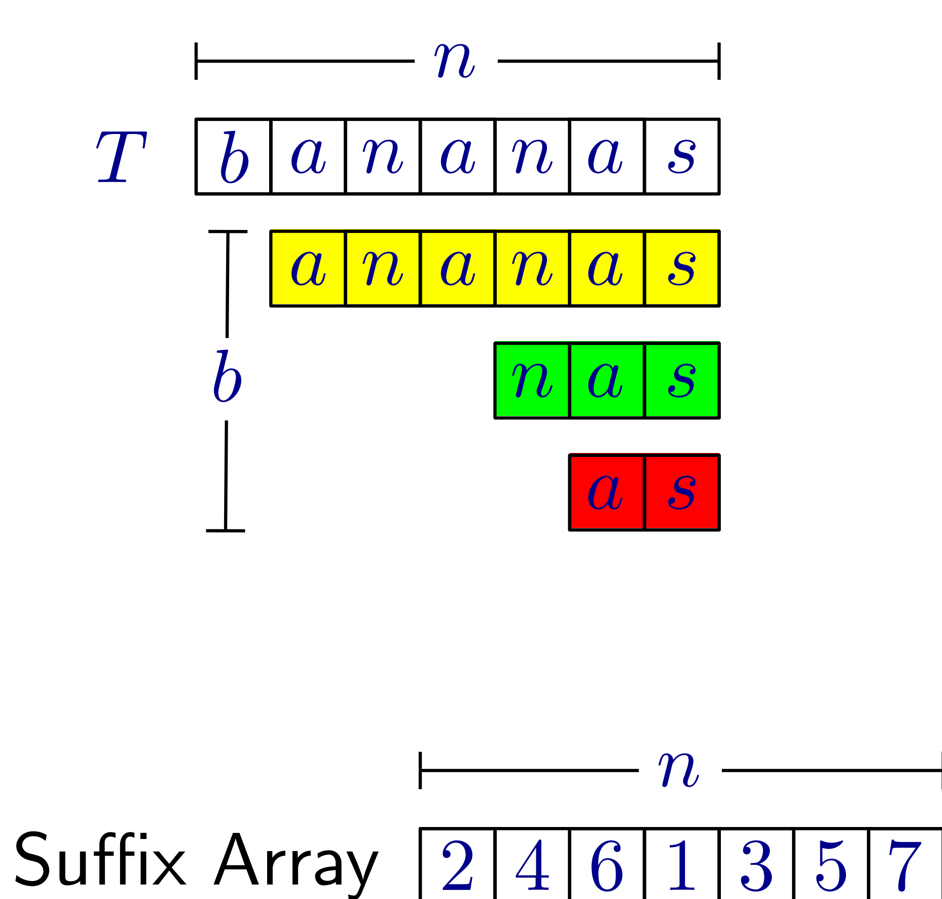
The sparse suffix array (SSA)

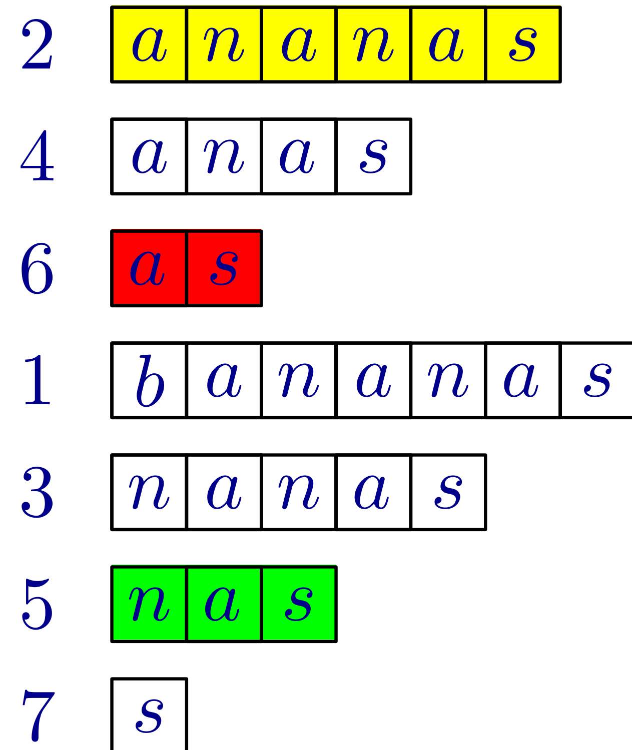




- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

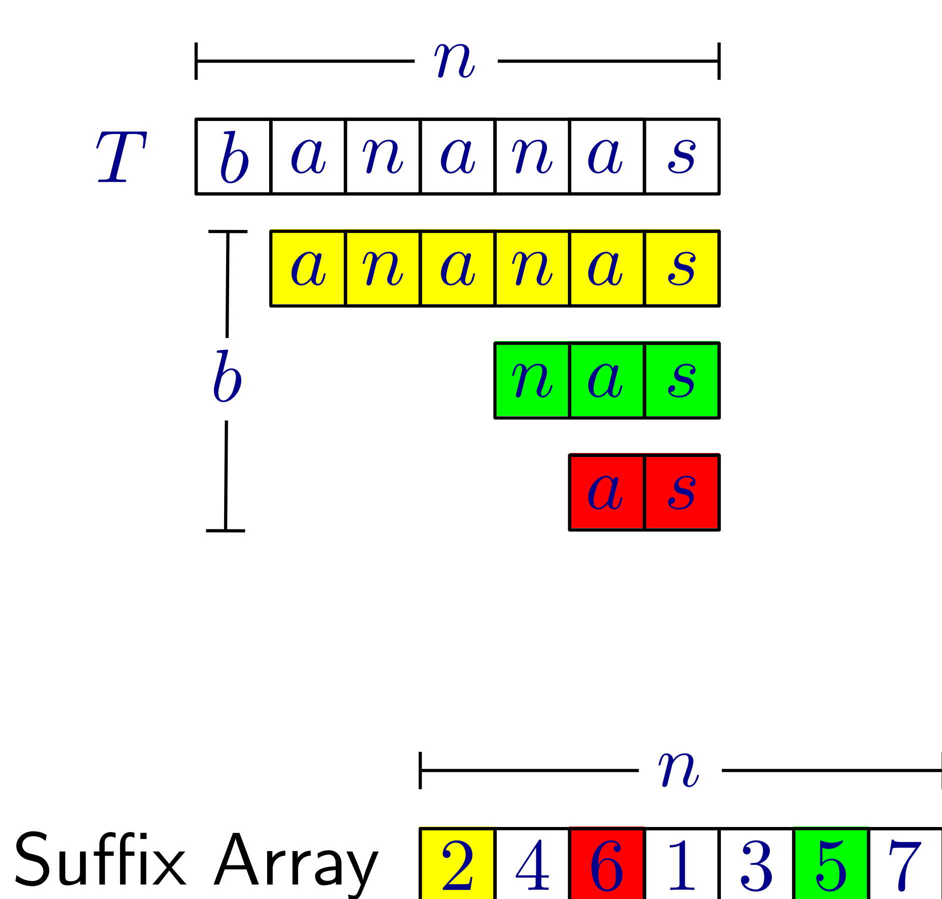
The sparse suffix array (SSA)

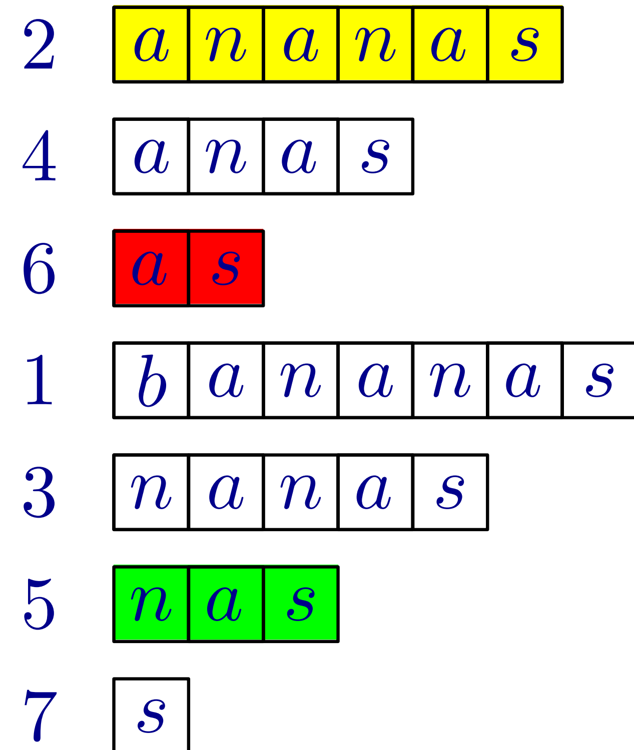




- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

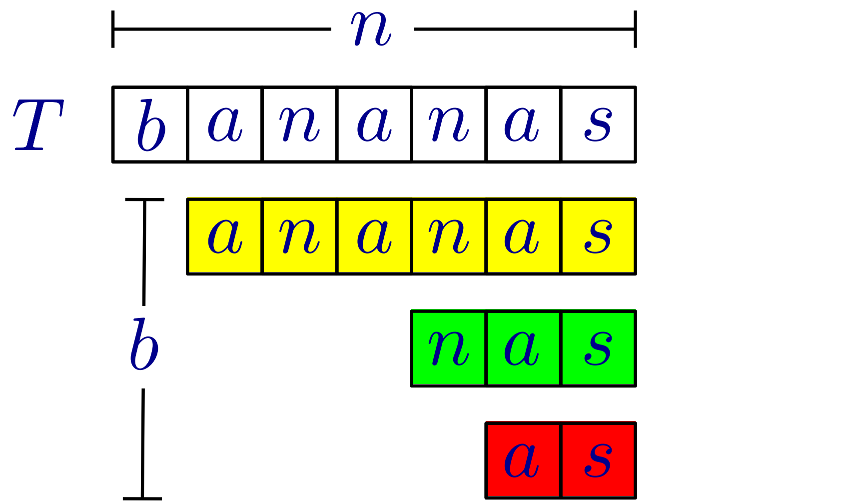
The sparse suffix array (SSA)

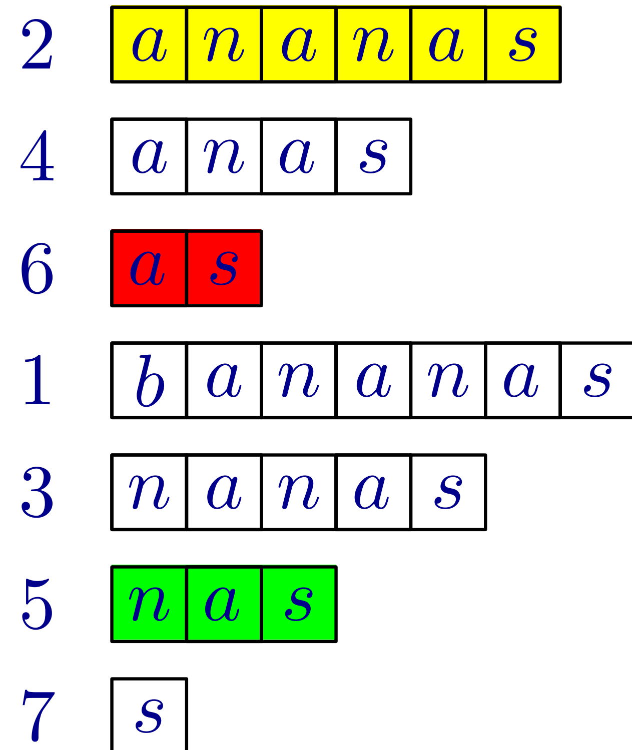


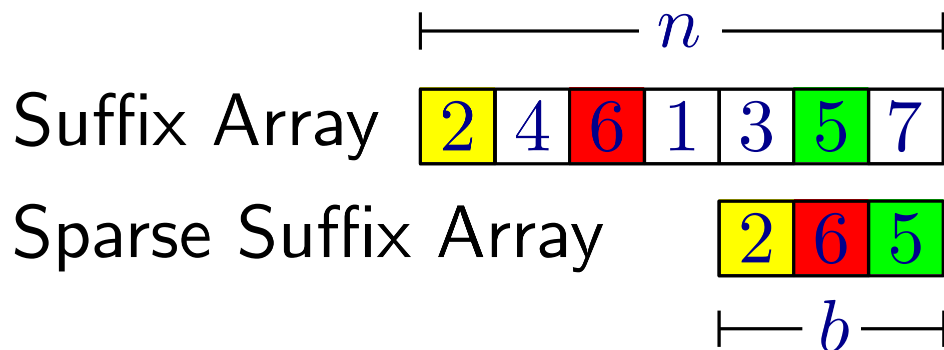


- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)

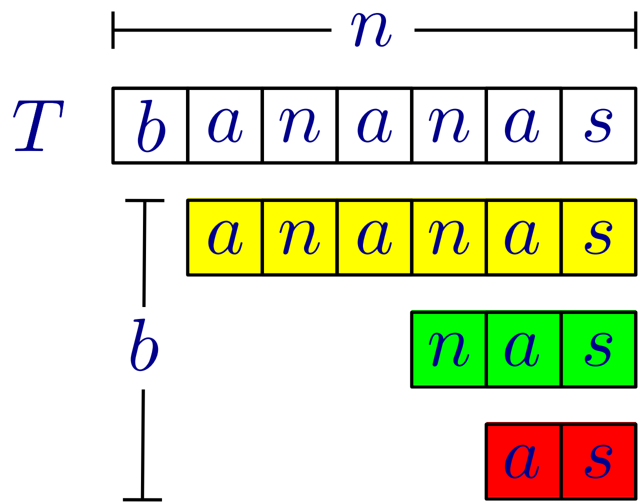


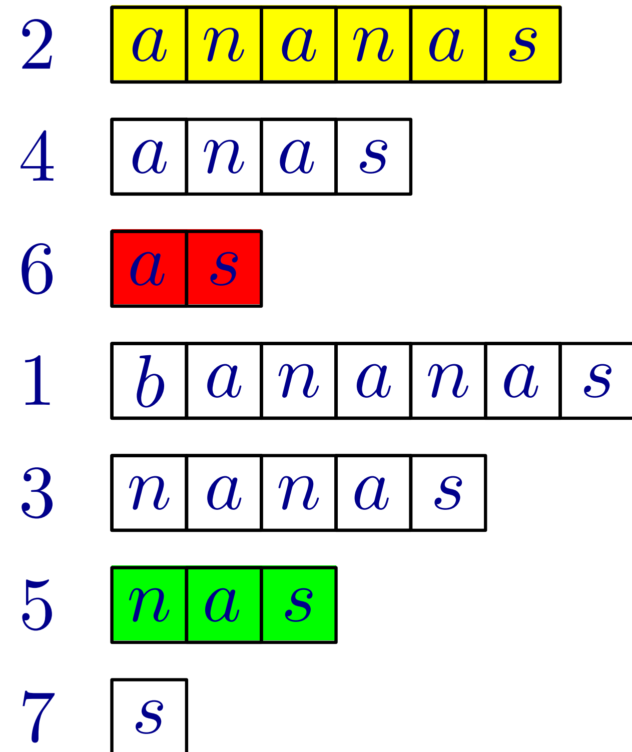


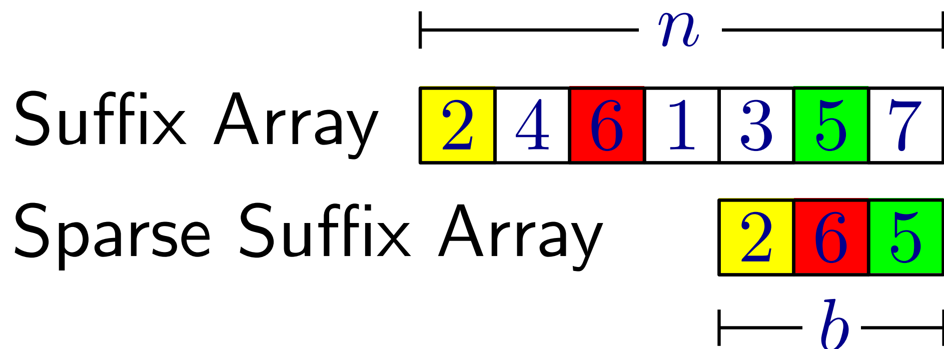


- Can be built in $O(n)$ time and $O(n)$ extra space
- What if we only care about a few of the suffixes?

The sparse suffix array (SSA)

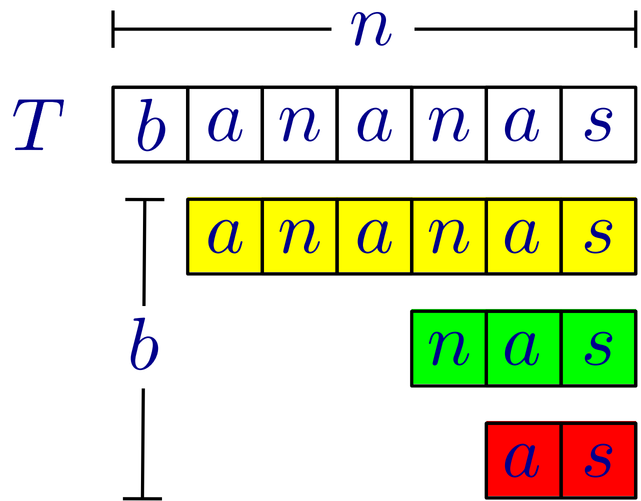


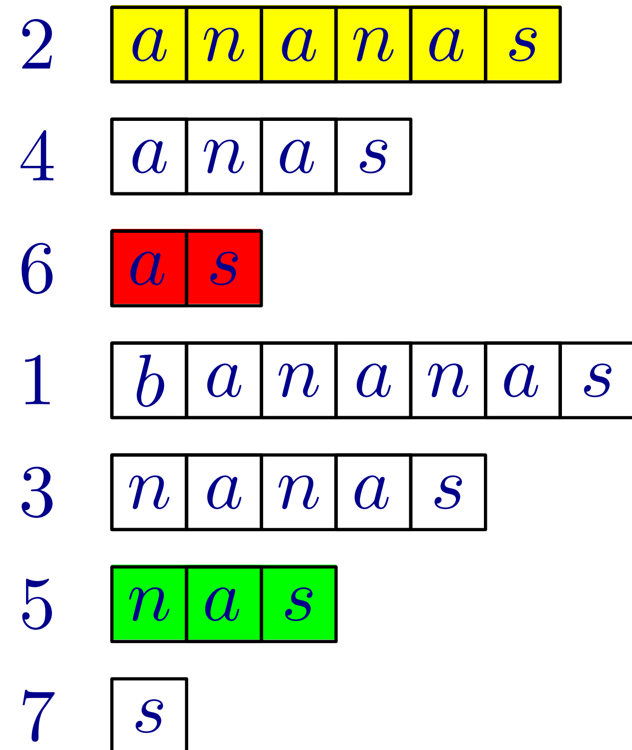


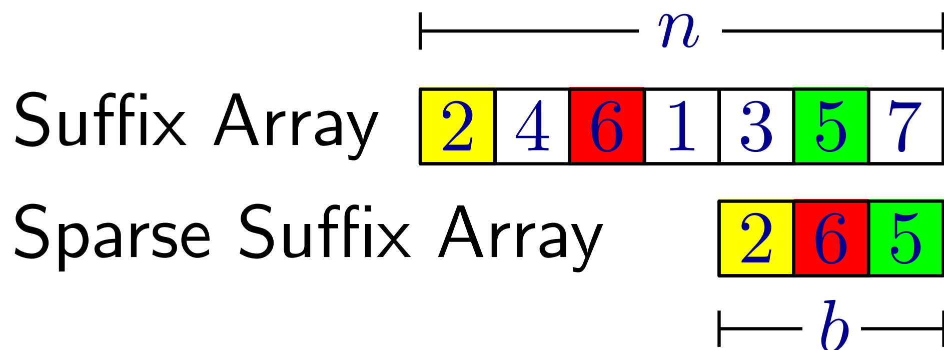


Conversion between SSA and SST is simple and takes $O(n \log b)$ *time*

The sparse suffix array (SSA)



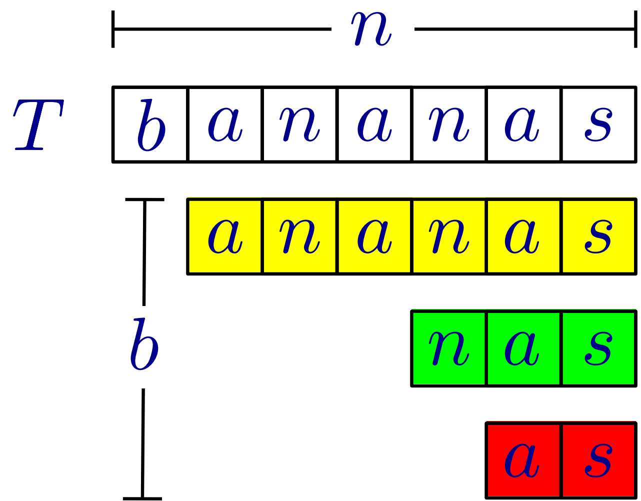




Conversion between SSA and SST is simple and takes $O(n \log b)$ time

. . . so we will focus on building the sparse suffix array

The sparse suffix array (SSA)



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

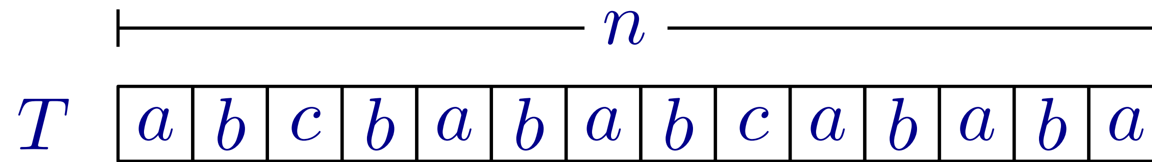
Suffix Array | 2 | 4 | 6 | 1 | 3 | 5 | 7

Sparse Suffix Array | 2 | 6 | 5

b

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

LCEs - a fundamental tool for pattern matching

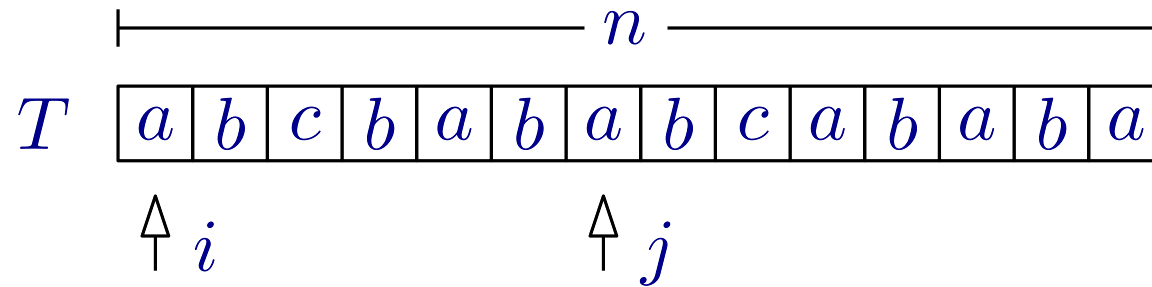


For any (i, j), the longest common extension is the largest ℓ such that

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

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

LCEs - a fundamental tool for pattern matching

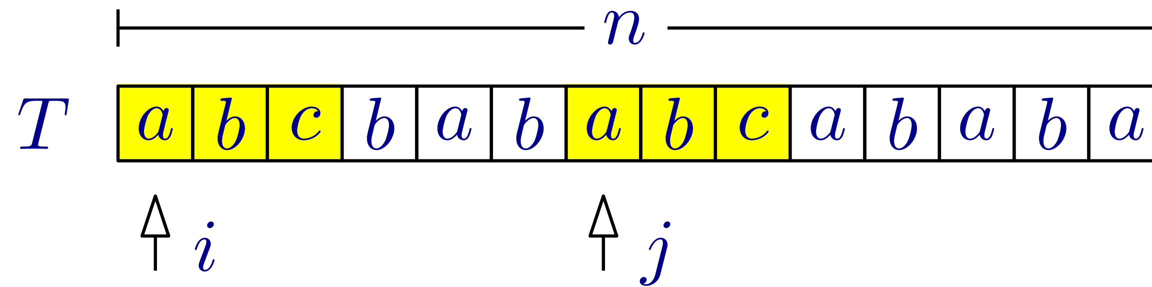


For any (i, j), the longest common extension is the largest ℓ such that

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

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

LCEs - a fundamental tool for pattern matching

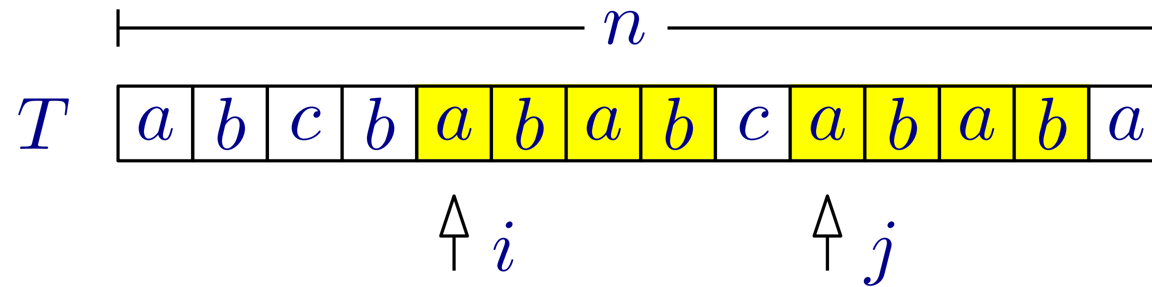


For any (i, j), the longest common extension is the largest ℓ such that

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

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

LCEs - a fundamental tool for pattern matching

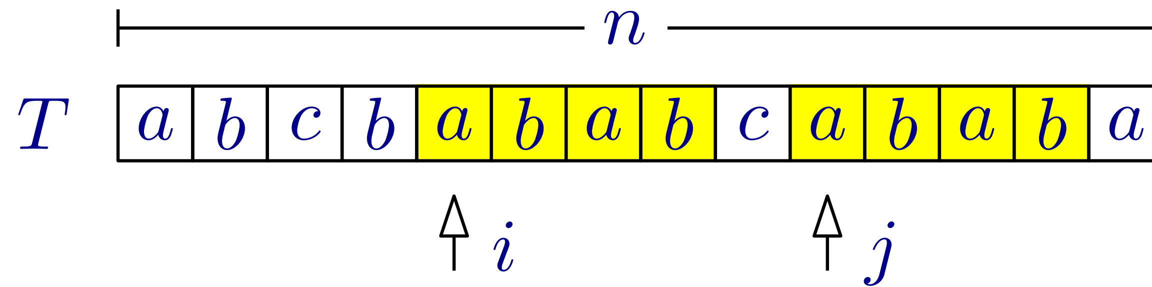


For any (i, j), the longest common extension is the largest ℓ such that

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

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

LCEs - a fundamental tool for pattern matching



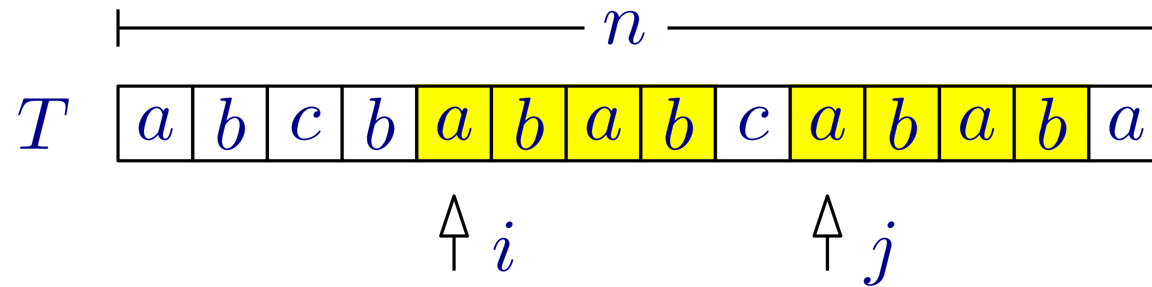
For any (i, j), the longest common extension is the largest ℓ such that

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

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

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

LCEs - a fundamental tool for pattern matching



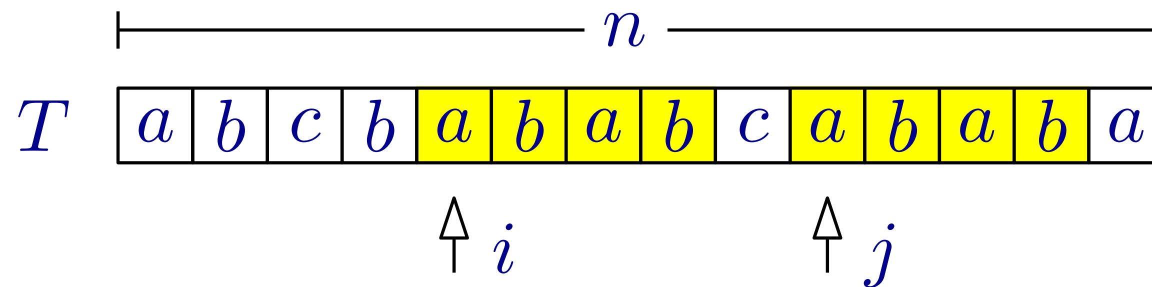
For any (i, j), the longest common extension is the largest ℓ such that

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

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

- LCE data structures are typically based on the suffix array or suffix tree.
- We do the opposite - we use *batched* LCE queries to construct the sparse suffix array

LCEs - a fundamental tool for pattern matching



For any (i, j), the longest common extension 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

- LCE data structures are typically based on the suffix array or suffix tree.
- We do the opposite - we use *batched* LCE queries to construct the sparse suffix array
- These LCE 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

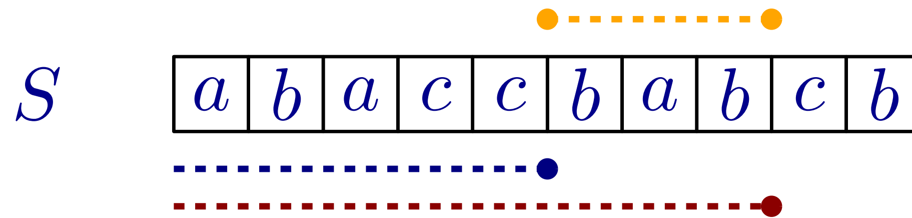
$$\phi(S) = \sum_{k=0}^{|S|-1} S[k] r^k \bmod p$$

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



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

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 LCE 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.

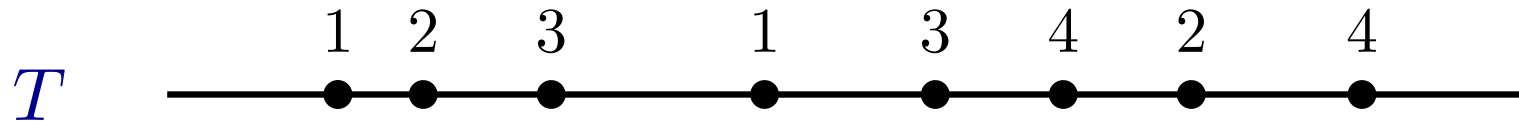
$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$

Simple, Monte-Carlo batched LCE 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.

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

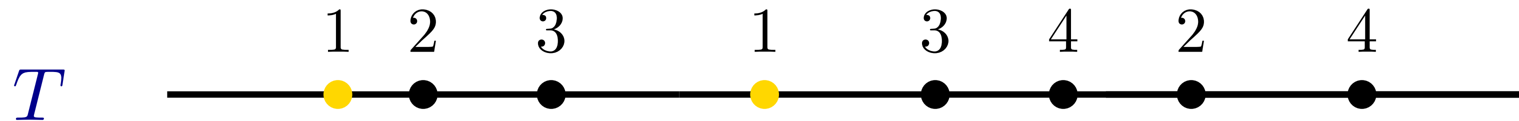


Simple, Monte-Carlo batched LCE queries

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

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

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

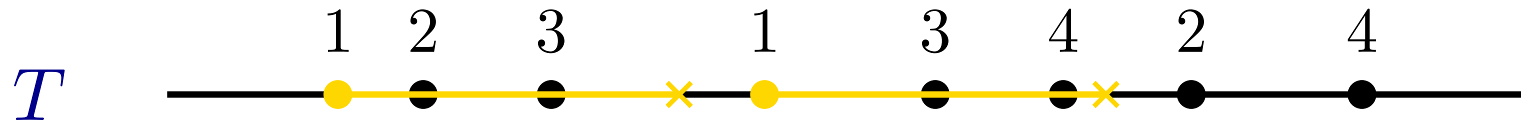


Simple, Monte-Carlo batched LCE queries

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

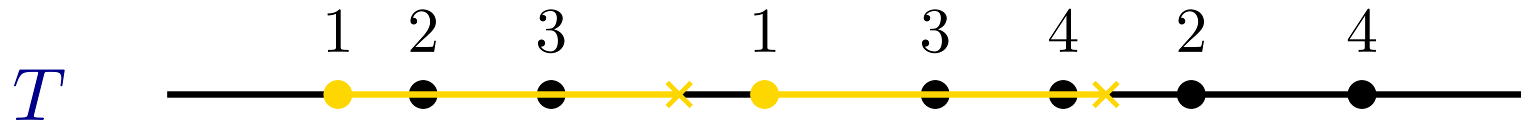
Output : for each pair (i, j) output the largest ℓ s.t.

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



Simple, Monte-Carlo batched LCE 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.
$$T[i \ldots i + \ell - 1] = T[j \ldots j + \ell - 1]$$



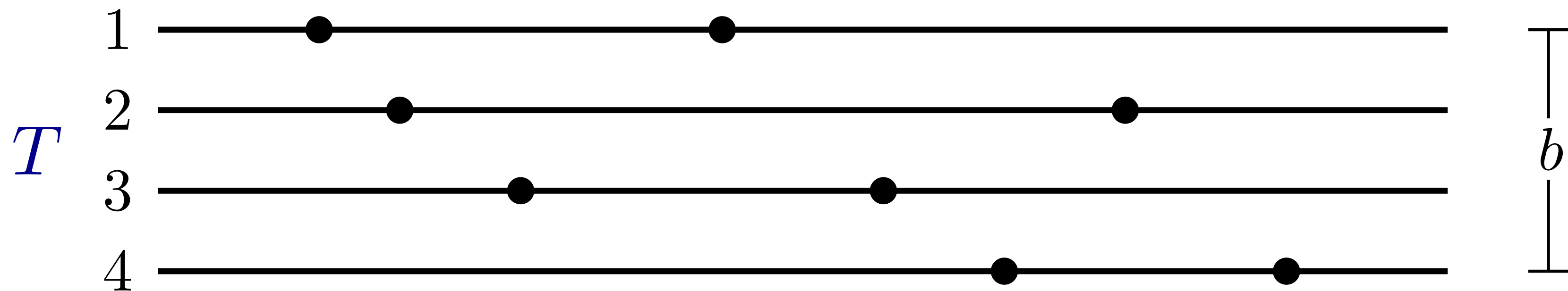
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



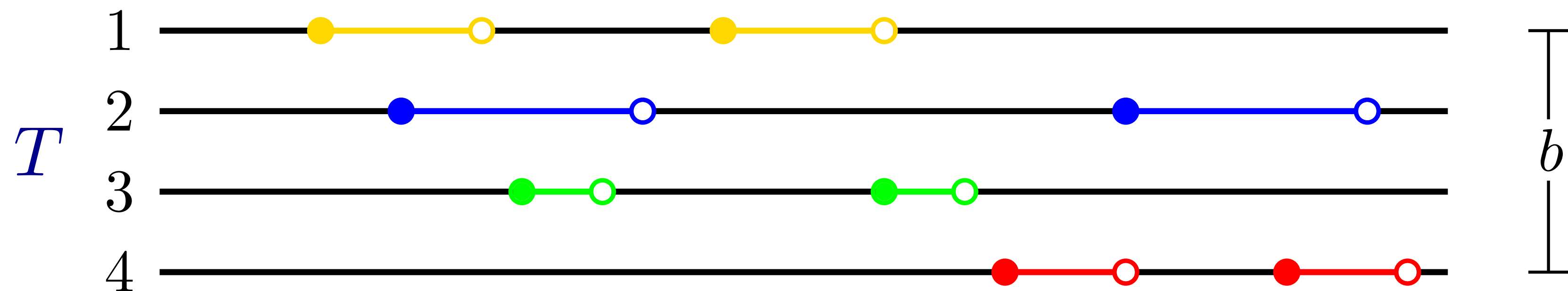
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



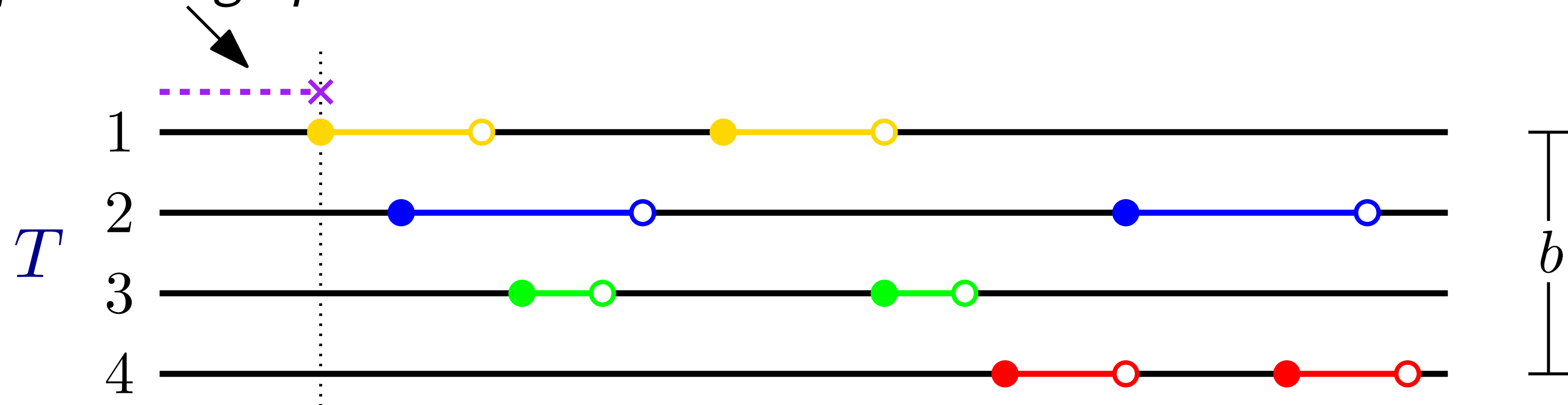
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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

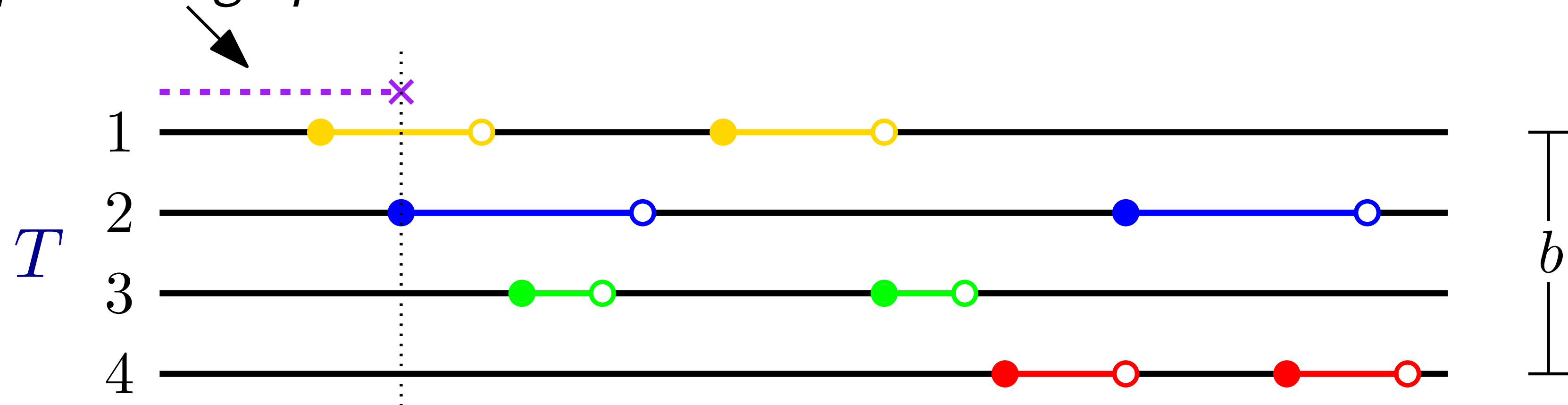


- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



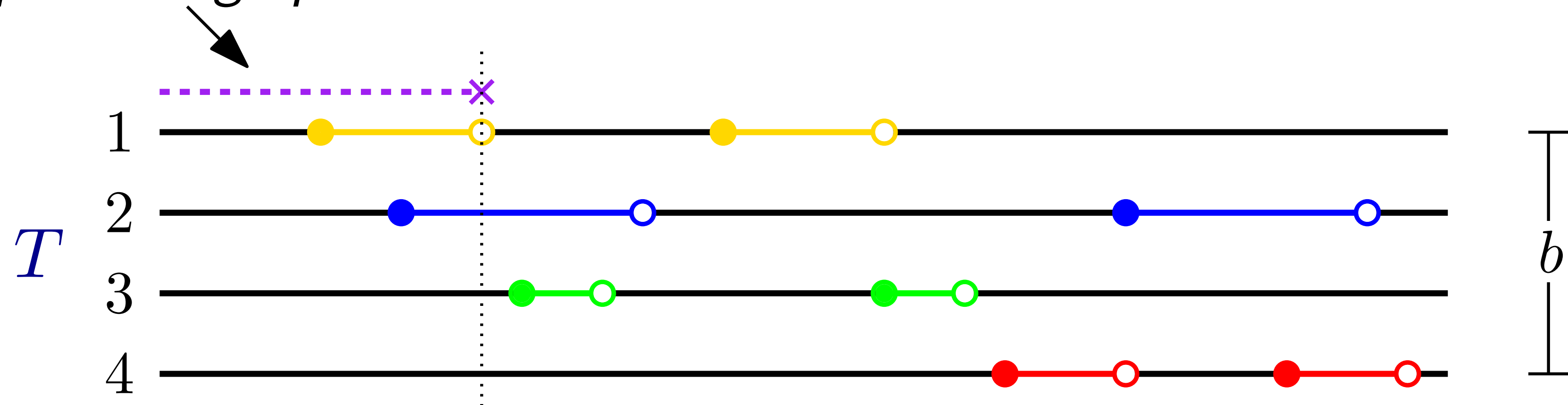
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



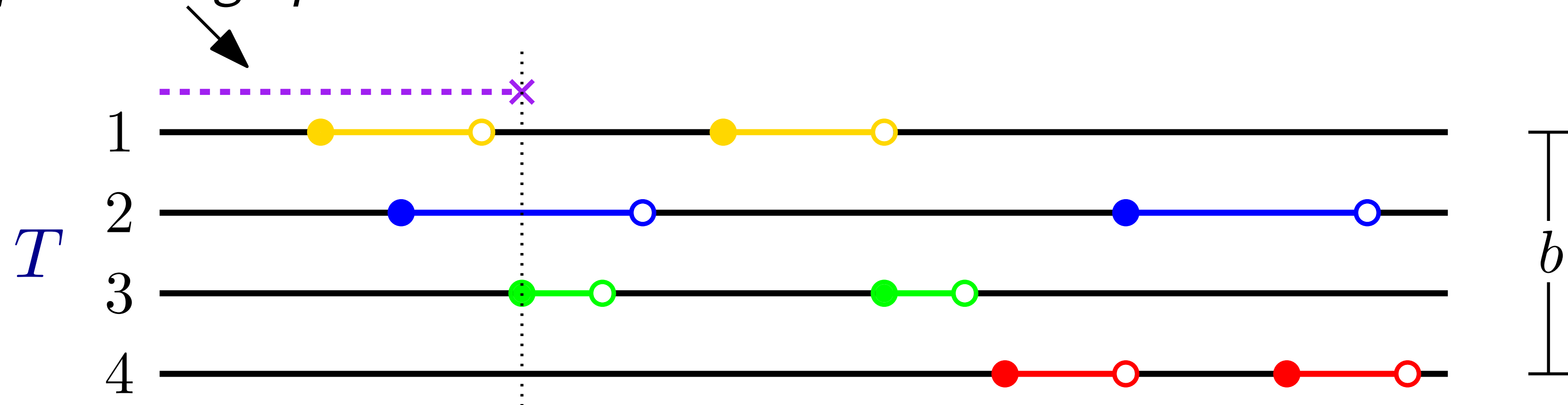
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



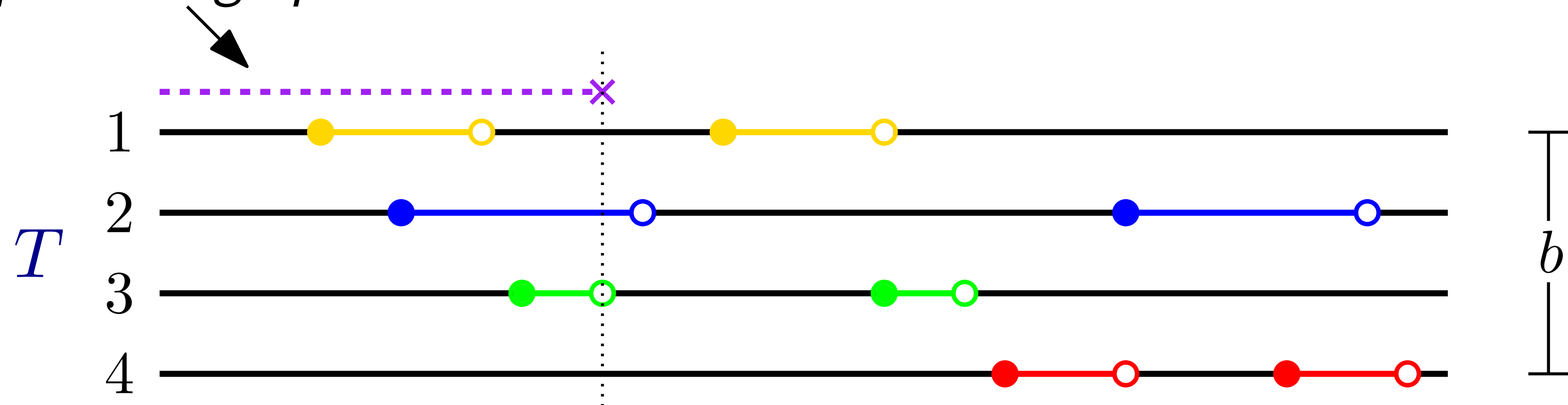
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



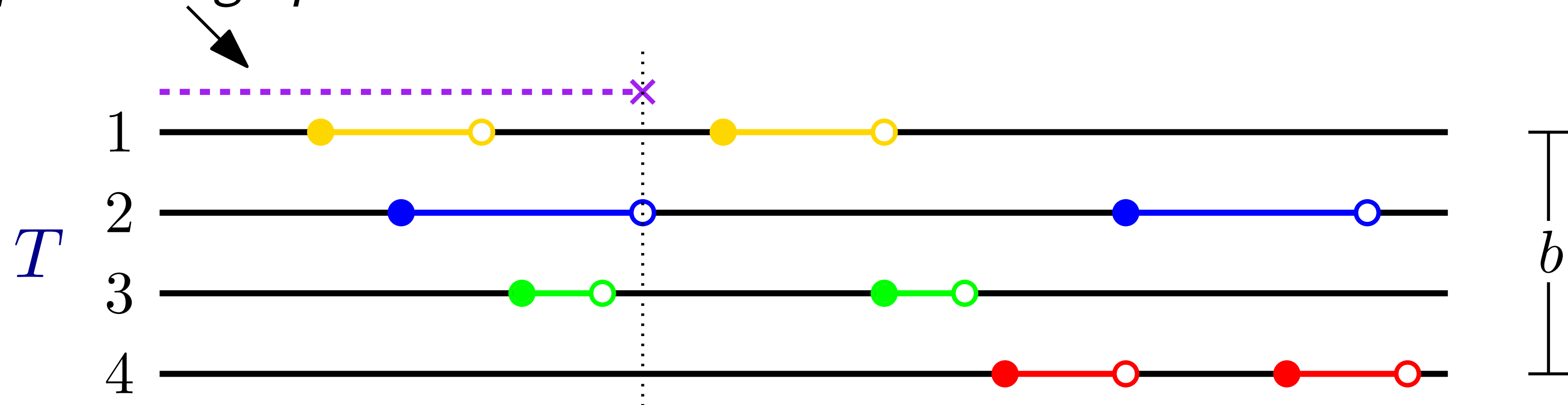
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



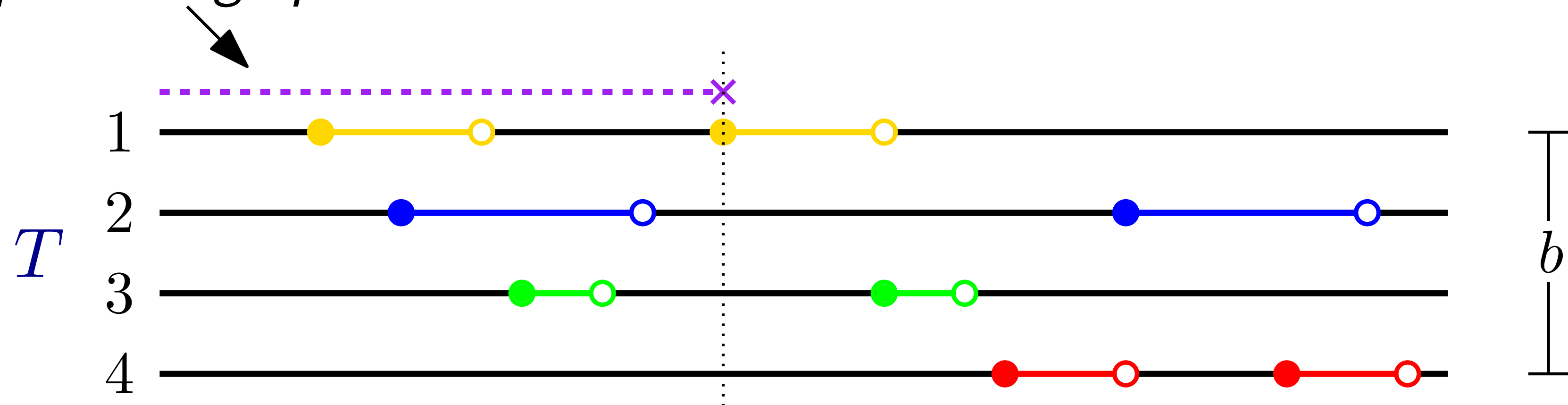
- We find the largest ℓ for each pair by binary search (in parallel) comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



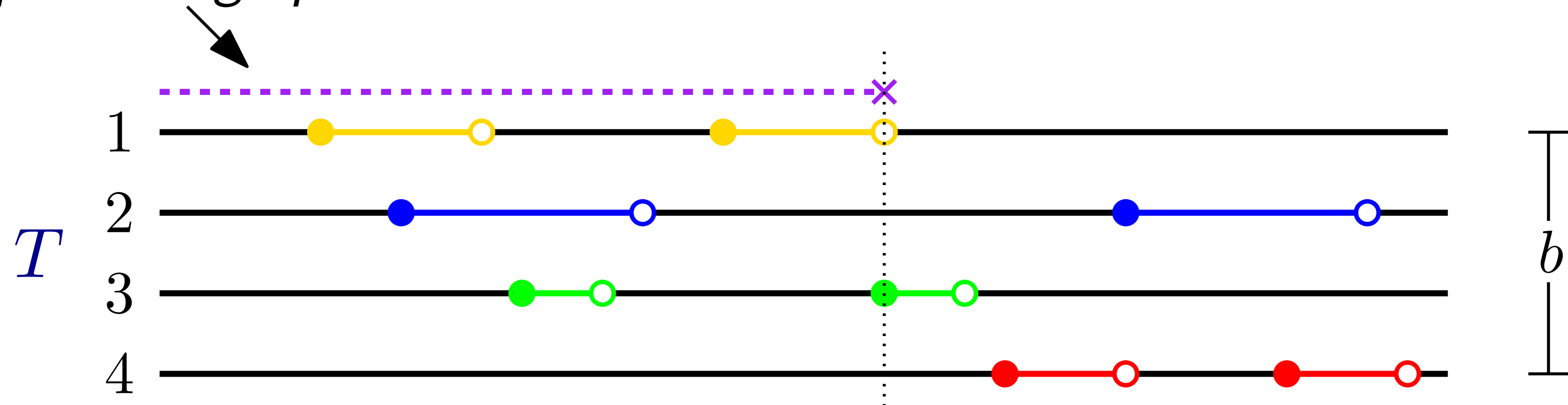
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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

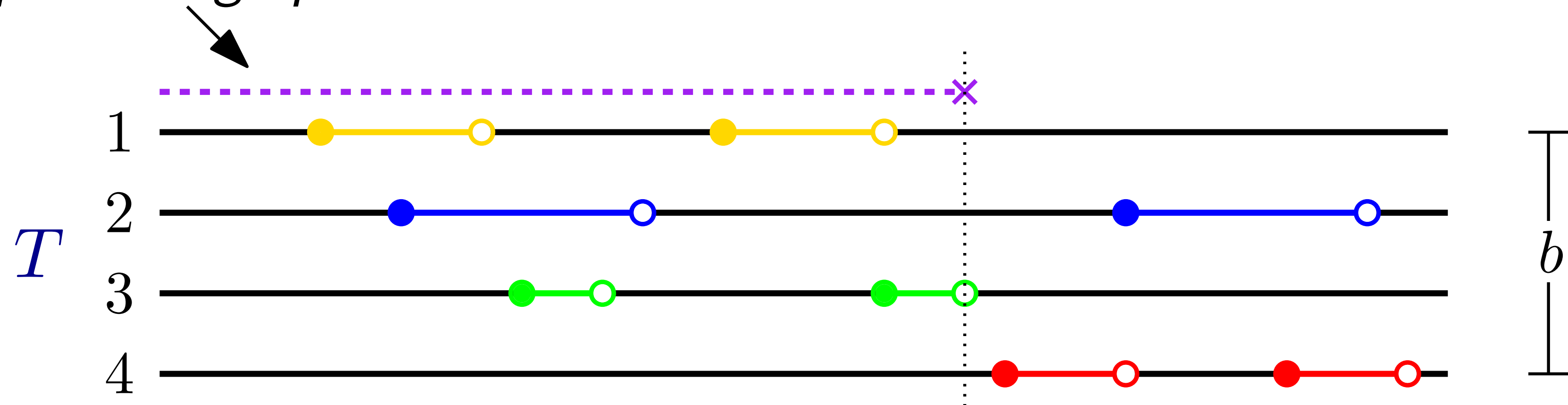


- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



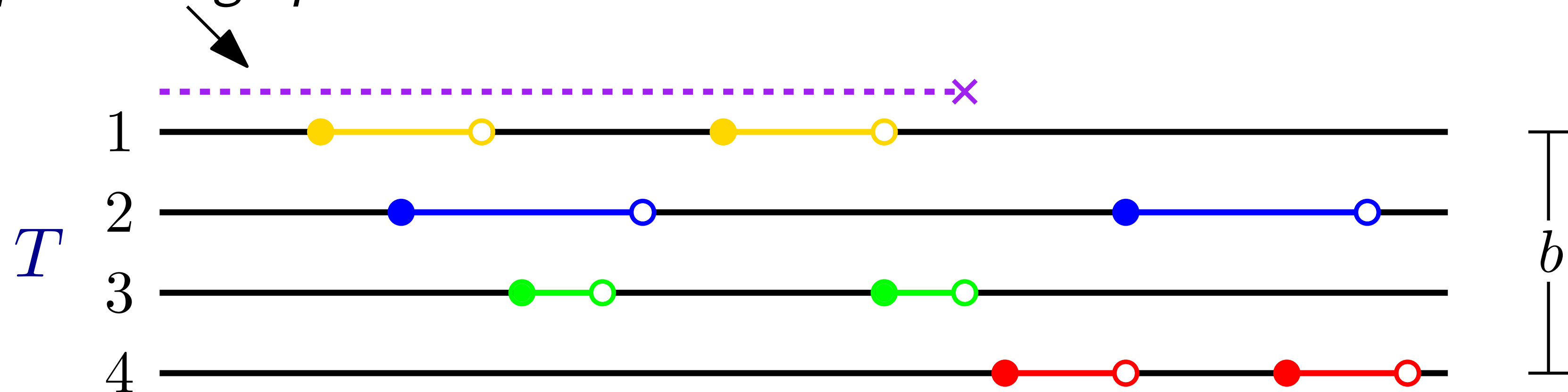
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints

Simple, Monte-Carlo batched LCE 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.

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



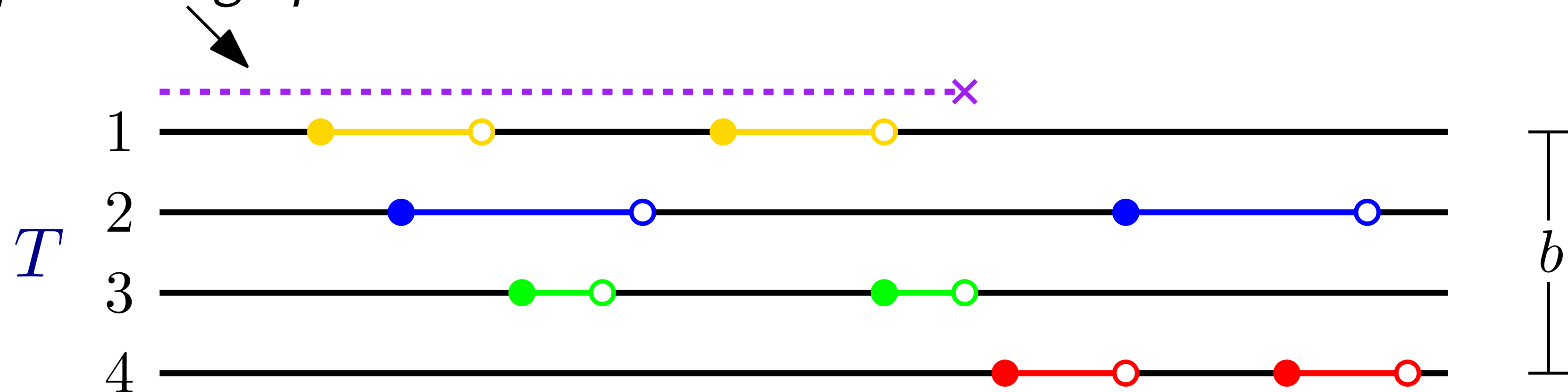
- We find the largest ℓ for each pair by binary search (in parallel)
  comparisons are performed using fingerprints
- In each pass we store (at most) $4b$ prefix fingerprints

Simple, Monte-Carlo batched LCE 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 LCEs

T	b	a	n	a	n	a	s

Building the sparse suffix array using batched LCEs

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 LCEs

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 LCEs

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

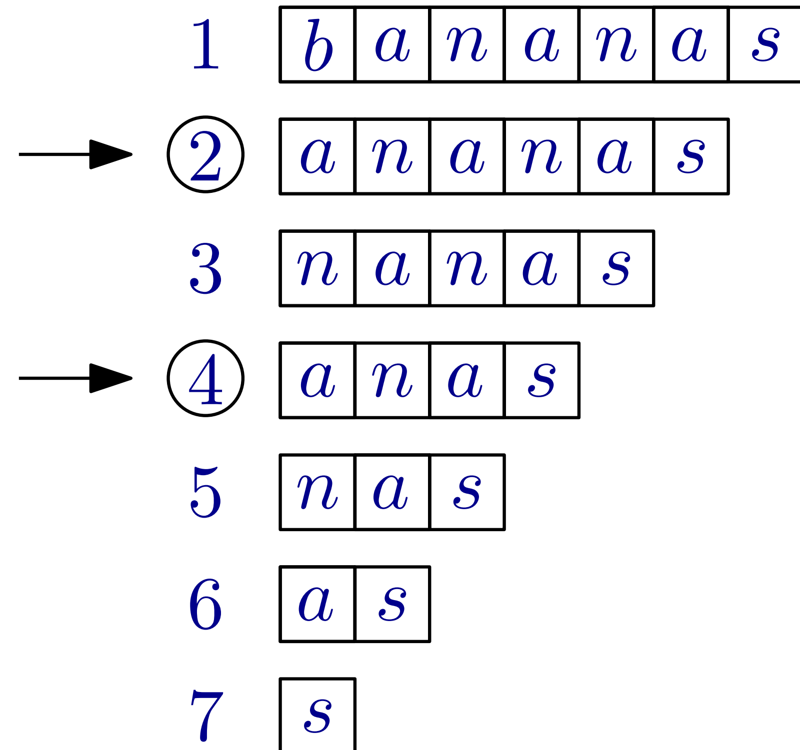
The LCE of two suffixes gives us their order

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 LCEs

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

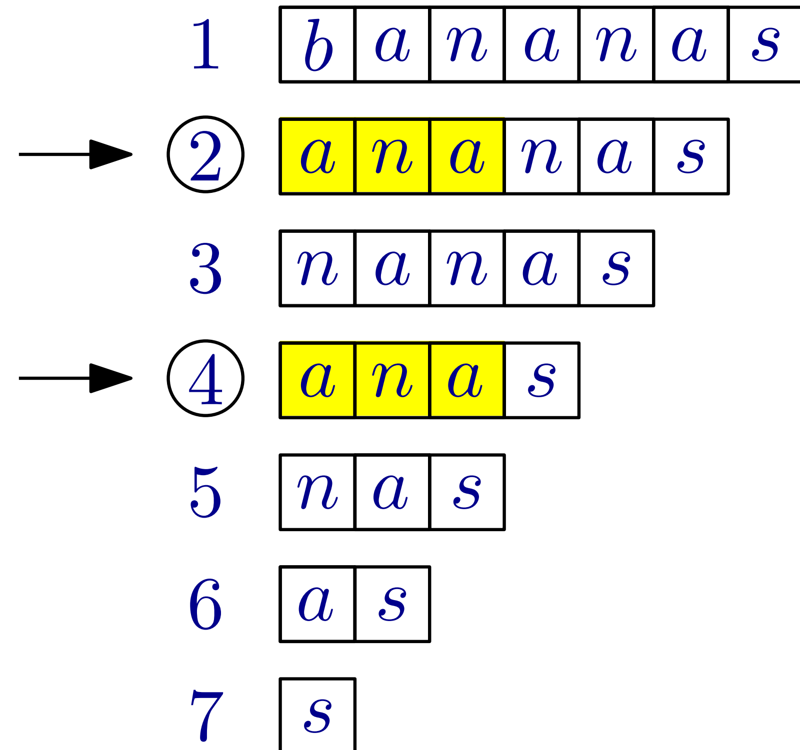
The LCE of two suffixes gives us their order



Building the sparse suffix array using batched LCEs

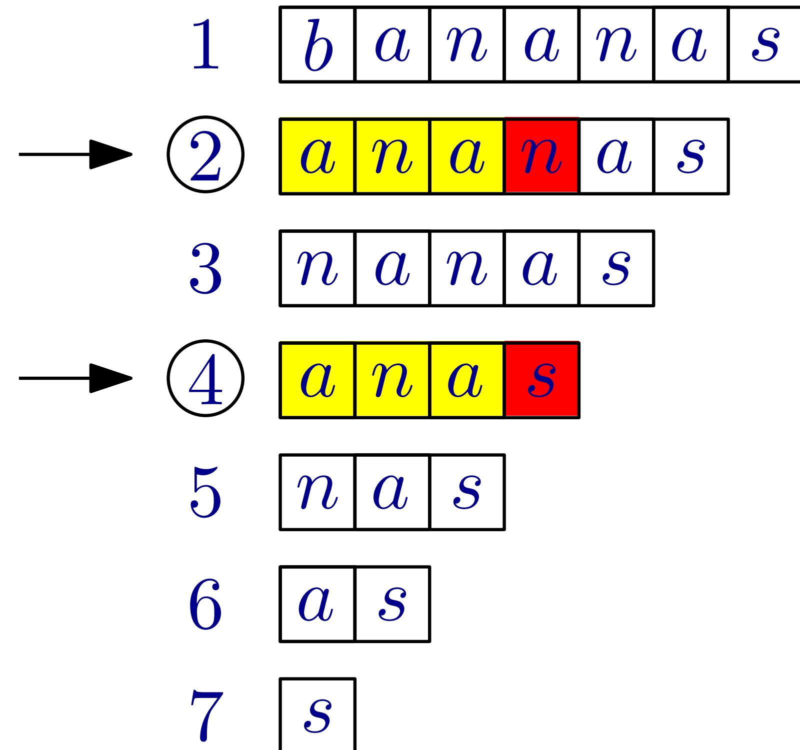
T | b | a | n | a | n | a | s

The LCE of two suffixes gives us their order



Building the sparse suffix array using batched LCEs

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



The LCE of two suffixes gives us their order

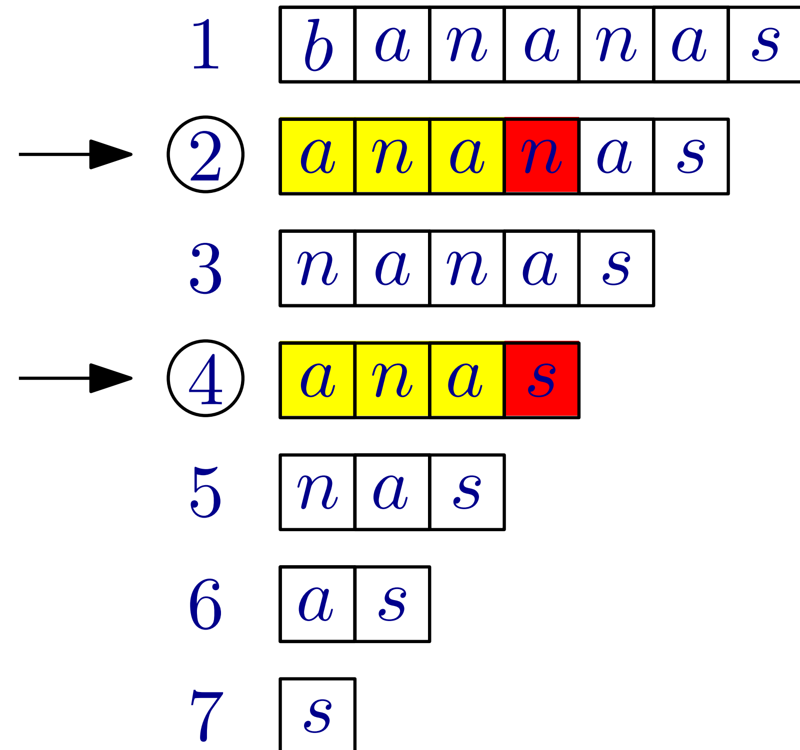
(2) < (4) because $n < s$

Building the sparse suffix array using batched LCEs

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

The LCE of two suffixes gives us their order

② < ④ because $n < s$

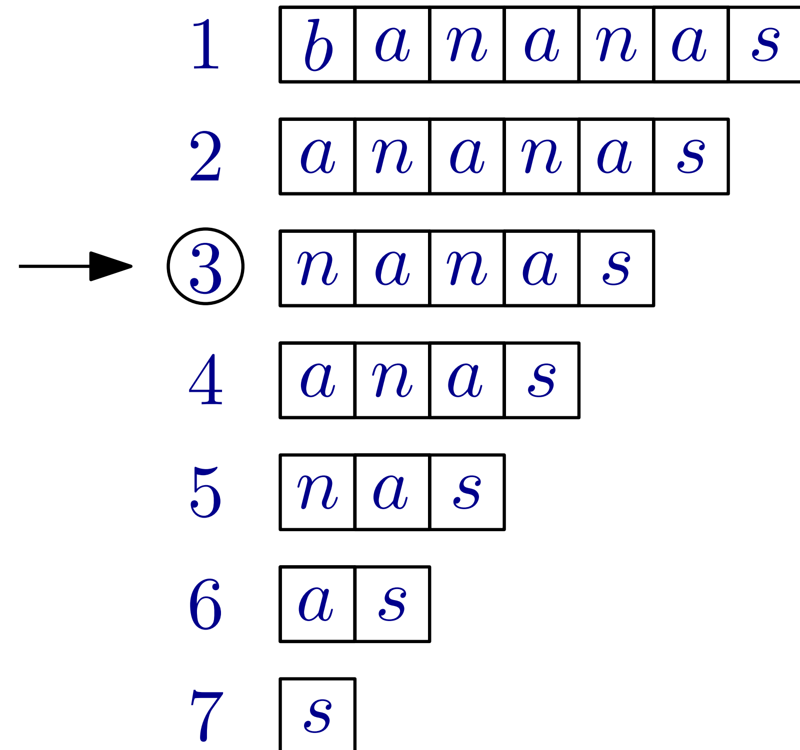


- We perform randomised quicksort on the b suffixes using batched LCEs for suffix comparisons

Building the sparse suffix array using batched LCEs

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

The LCE of two suffixes gives us their order



- We perform randomised quicksort on the b suffixes using batched LCEs 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 LCEs

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

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

The LCE of two suffixes gives us their order

- We perform randomised quicksort on the b suffixes using batched LCEs 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 LCEs

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

The LCE of two suffixes gives us their order

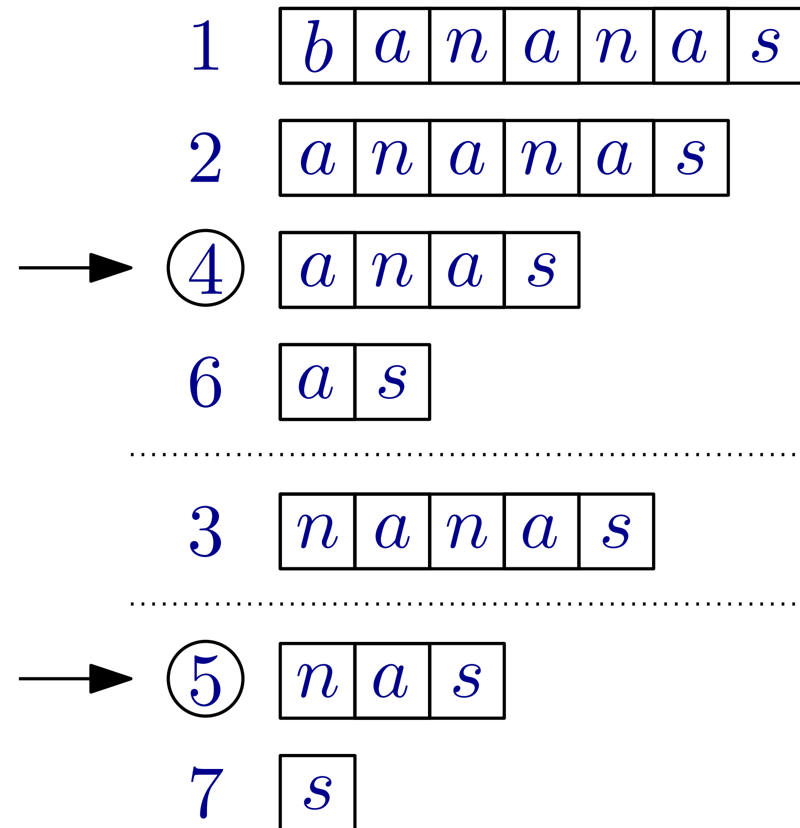
1 $b\ a\ n\ a\ n\ a\ s$
2 $a\ n\ a\ n\ a\ s$
4 $a\ n\ a\ s$
6 $a\ s$
→ ③ $n\ a\ n\ a\ s$
5 $n\ a\ s$
7 s

- We perform randomised quicksort on the b suffixes using batched LCEs 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 LCEs)*

Building the sparse suffix array using batched LCEs

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

The LCE of two suffixes gives us their order

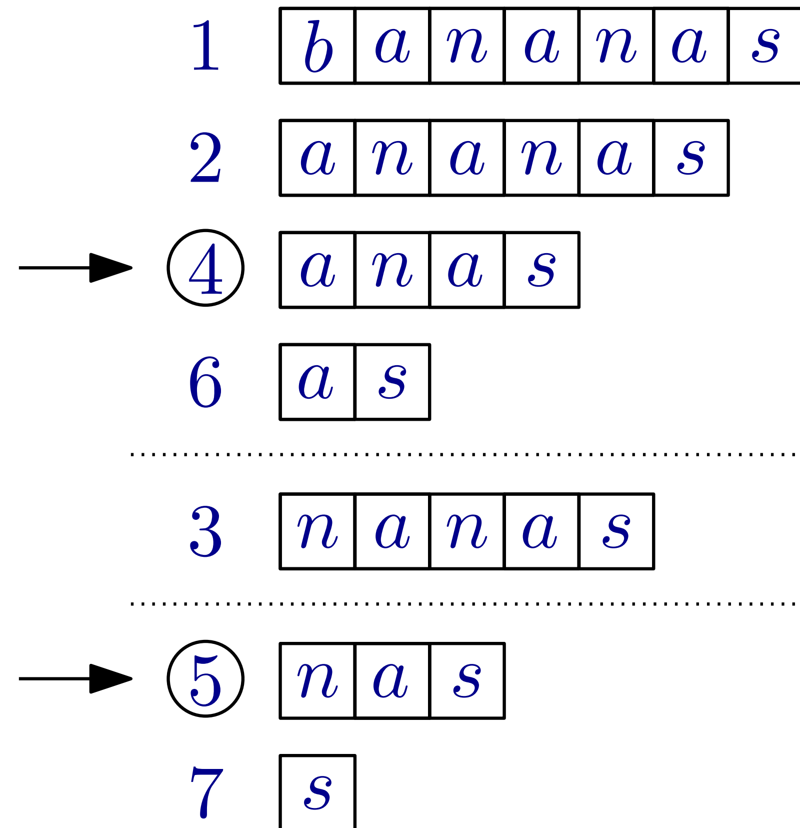


- We perform randomised quicksort on the b suffixes using batched LCEs for suffix comparisons
- Pick a random pivot and compare each other suffix to it
  - This partitions the suffixes in $O(n \log b)$ time and $O(b)$ space
  - *Recurse on each partition (the batch still contains b LCEs)*

Building the sparse suffix array using batched LCEs

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

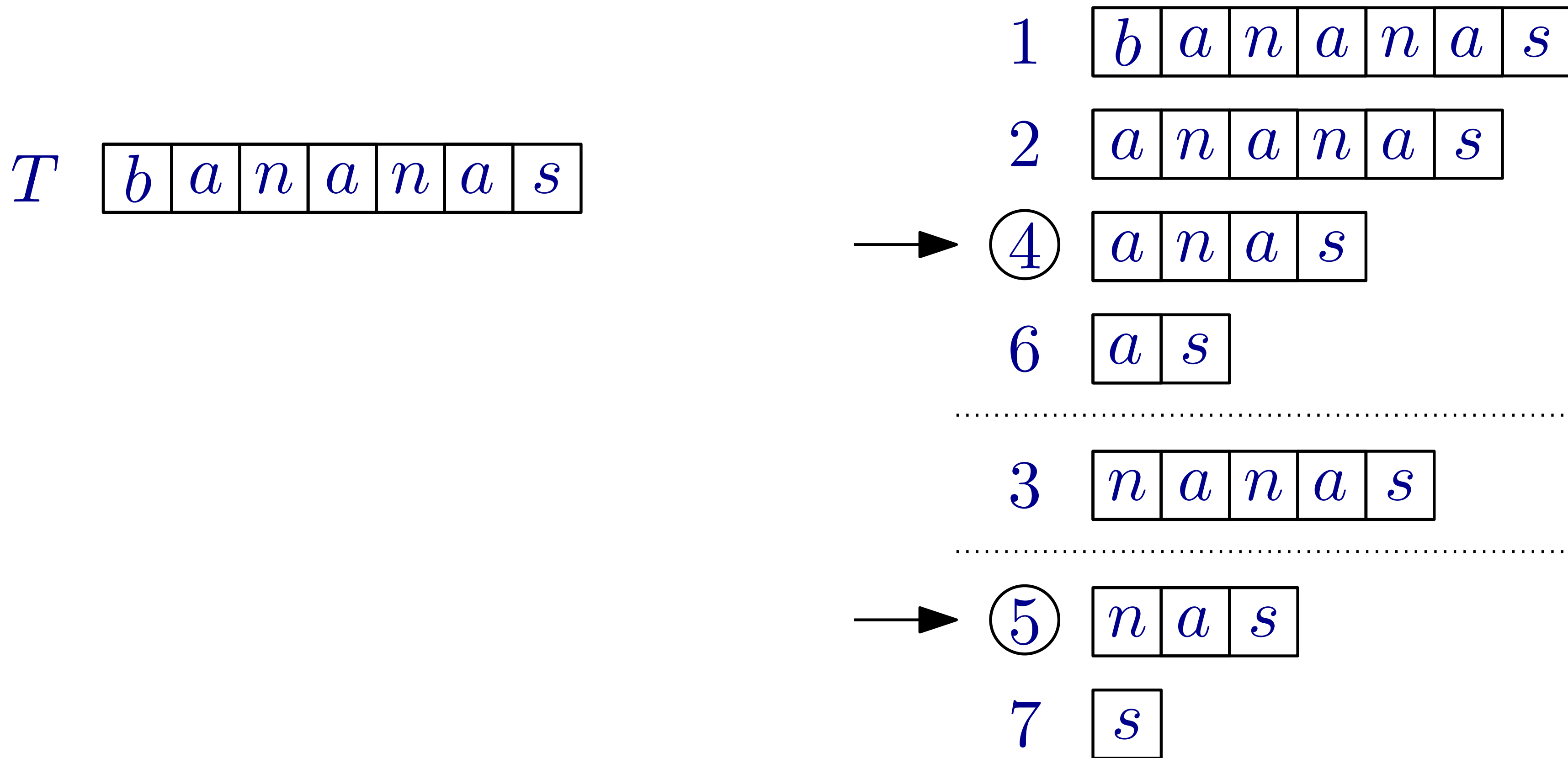
The LCE of two suffixes gives us their order



- We perform randomised quicksort on the b suffixes using batched LCEs 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 LCEs

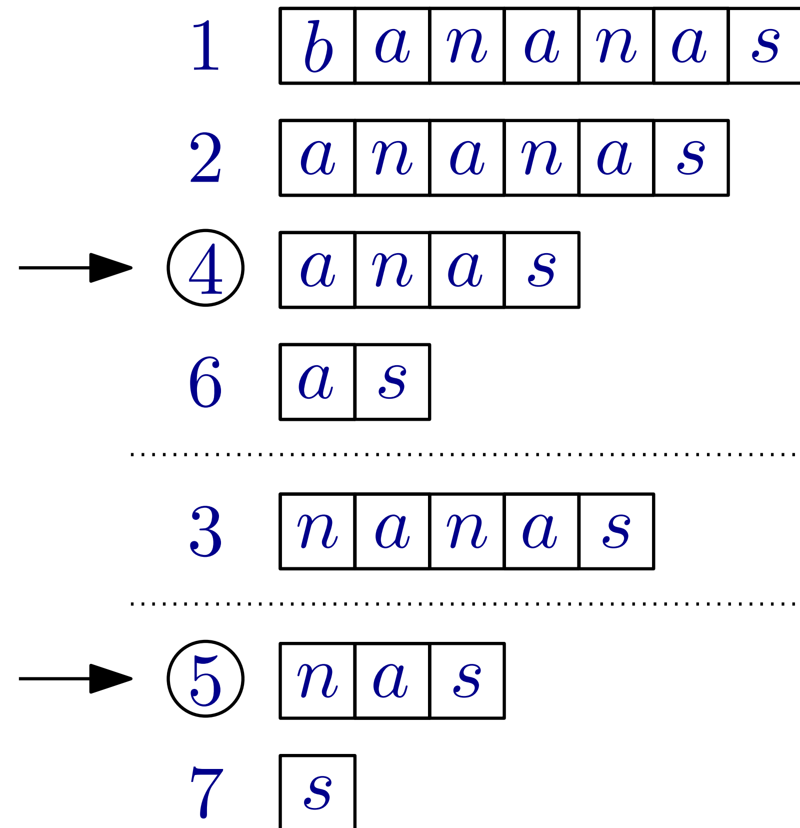


- We perform randomised quicksort on the b suffixes using batched LCEs for suffix comparisons
- The depth of the recursion is $O(\log b)$ whp. so...

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

Building the sparse suffix array using batched LCEs

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

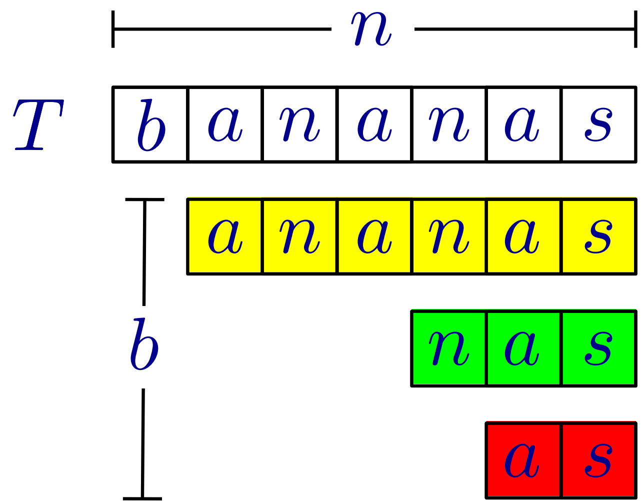


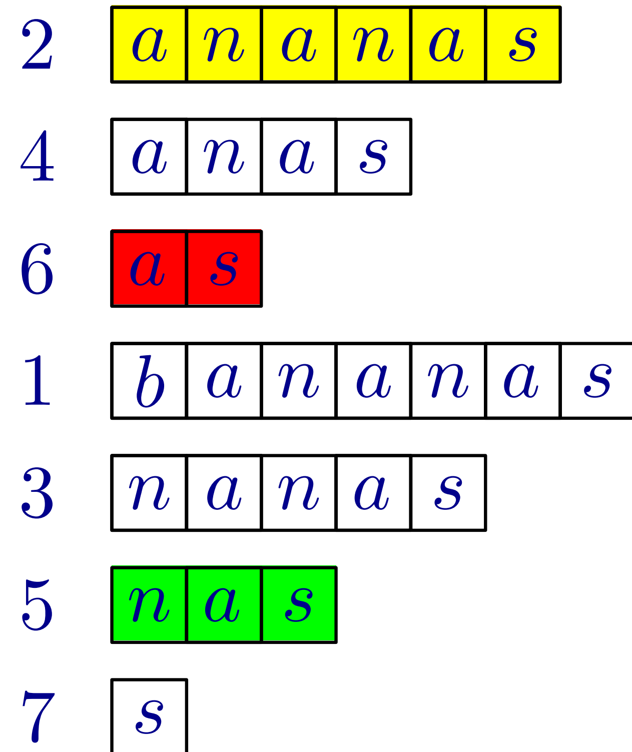
This algorithm is Monte-Carlo *and* Las-Vegas. It can be made Monte-Carlo only by aborting the quicksort early

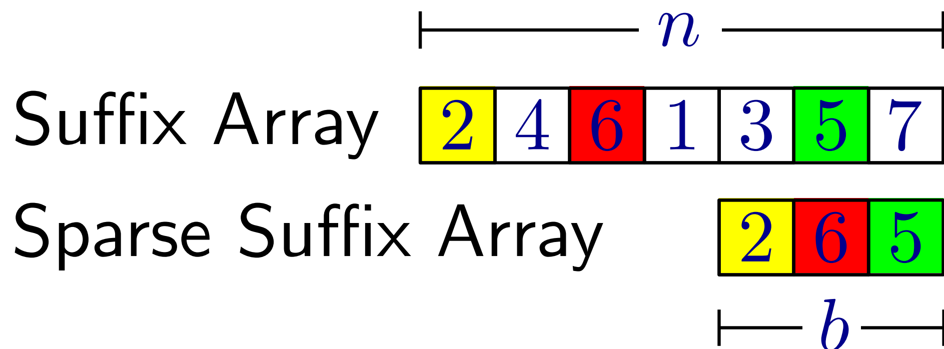
- We perform randomised quicksort on the b suffixes using batched LCEs for suffix comparisons
- The depth of the recursion is $O(\log b)$ whp. so...

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

The sparse suffix array (SSA)

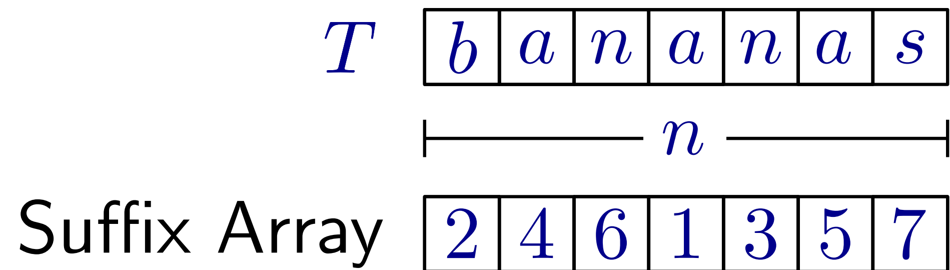






- $O(n \log^2 b)$ time (Monte-Carlo)
- $O((n + b^2) \log^2 b)$ time with high probability (Las-Vegas)
- both in $O(b)$ space

Verifying the sparse suffix array



How can we tell if this suffix array is correct?

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

Verifying the sparse suffix array

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

(length of T: n)

How can we tell if this suffix array is correct?

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

This suffices because lexicographical ordering is transitive

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

This suffices because lexicographical ordering is transitive

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

Verifying the sparse suffix array

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

$\longleftarrow n \longrightarrow$

How can we tell if this suffix array is correct?

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

This suffices because lexicographical ordering is transitive

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

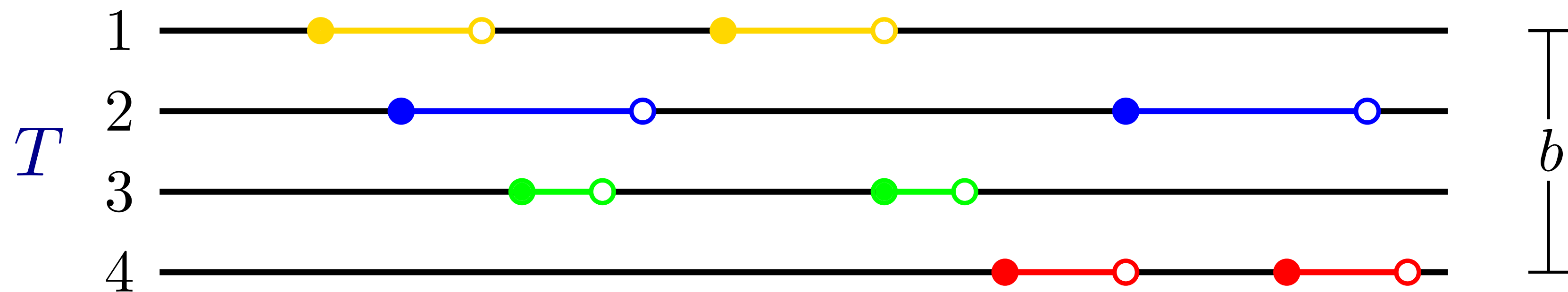
So it suffices to verify a batch of b LCE queries

Verifying batched LCE queries

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

Output : for each triple (i, j, ℓ) check that

$$T[i+\ell] \neq T[j+\ell] \textit{ and } T[i \ldots i+\ell-1] = T[j \ldots j+\ell-1]$$



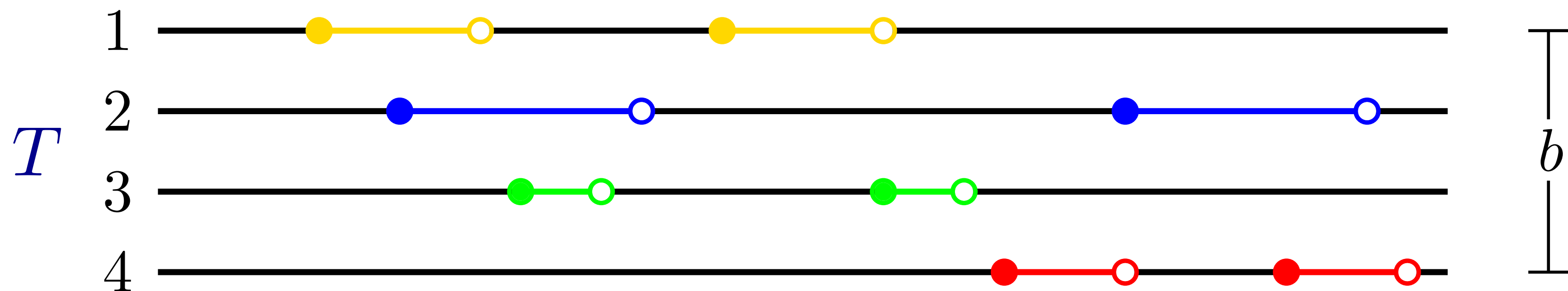
Verifying batched LCE queries

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

Output : for each triple (i, j, ℓ) check that

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

easy! O(n) time

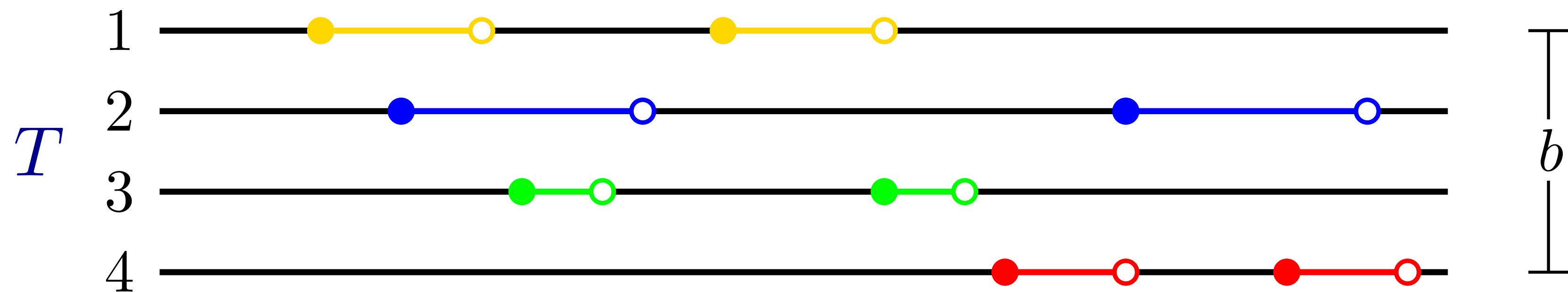


Verifying batched LCE queries

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

Output : for each triple (i, j, ℓ) check that

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

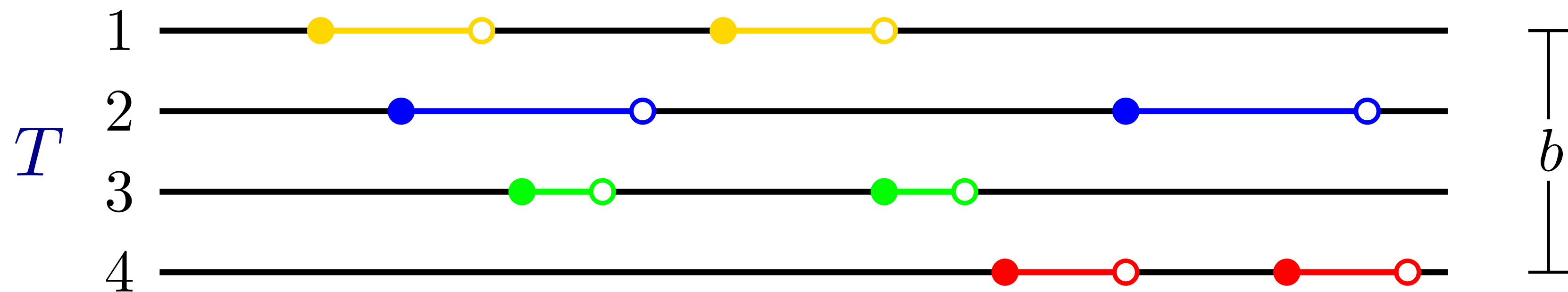


Verifying batched LCE queries

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

Output : for each triple (i, j, ℓ) check that

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



- We proceed in rounds in descending order...

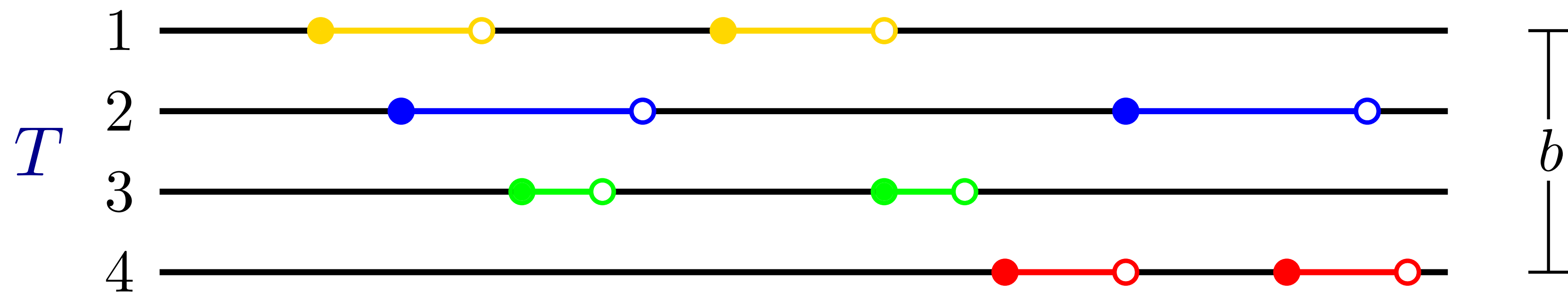
  in the k-th round every $\ell = 2^k$

Verifying batched LCE queries

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

Output : for each triple (i, j, ℓ) check that

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

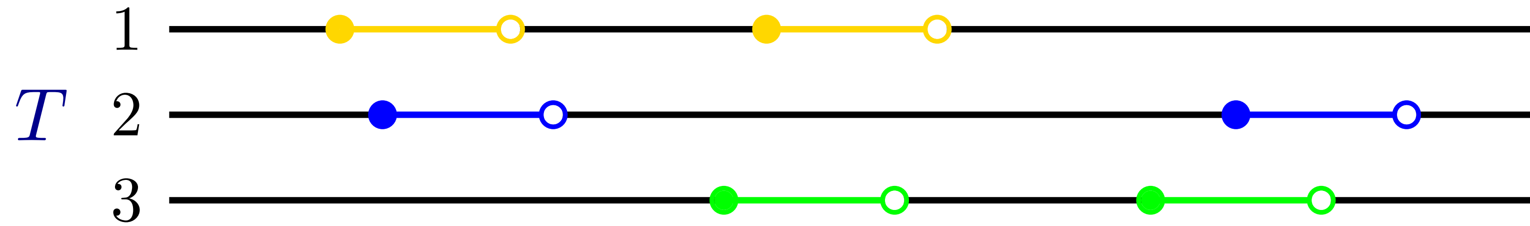


- We proceed in rounds in descending order...

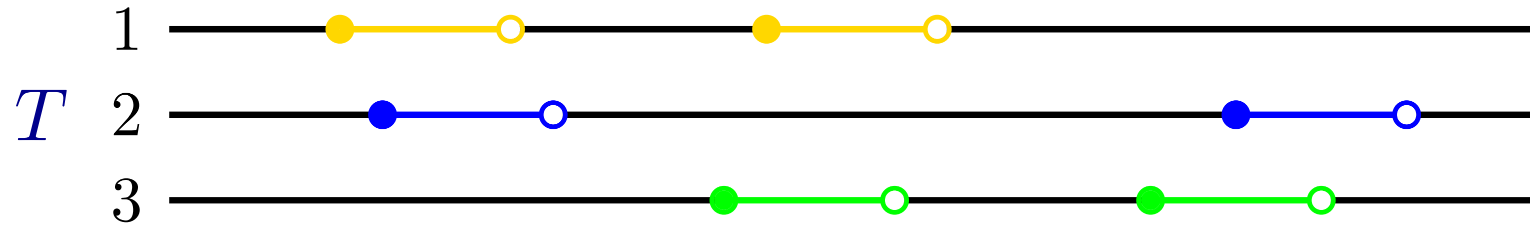
  in the k-th round every $\ell = 2^k$

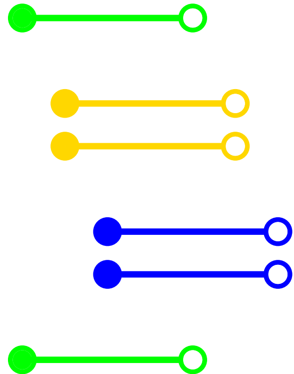
Let's focus on a single round

A first example

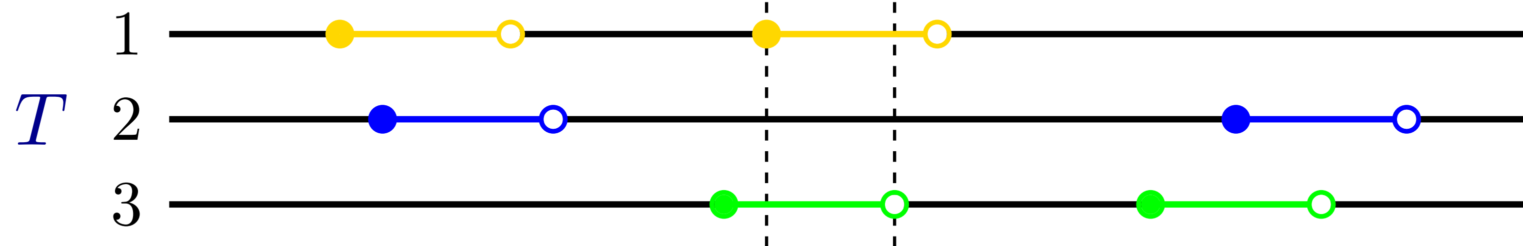


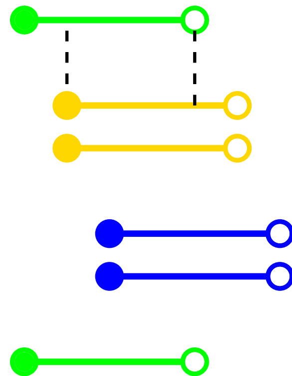
A first example



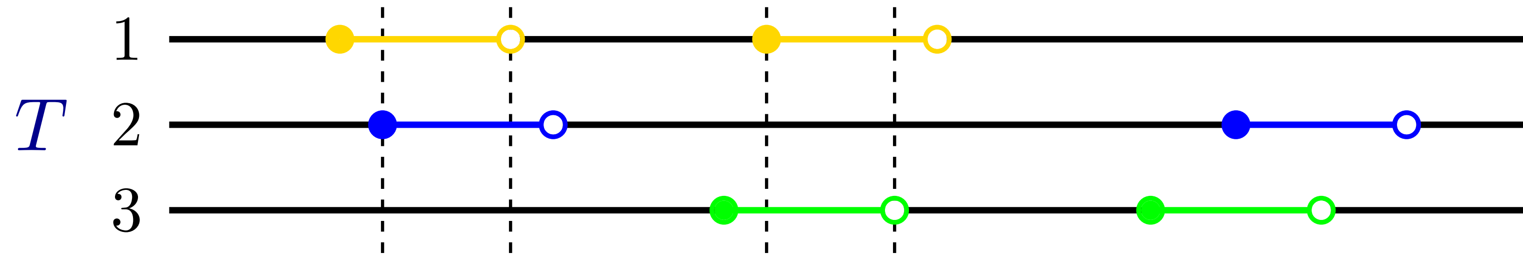


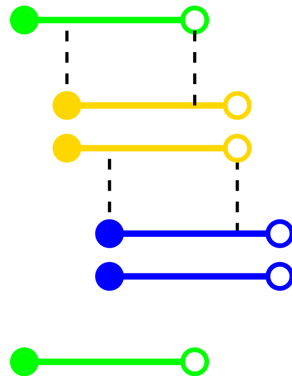
A first example



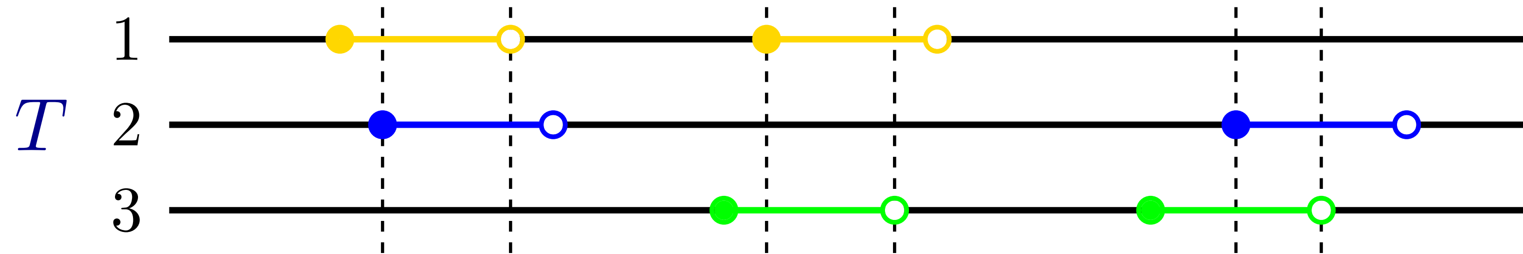


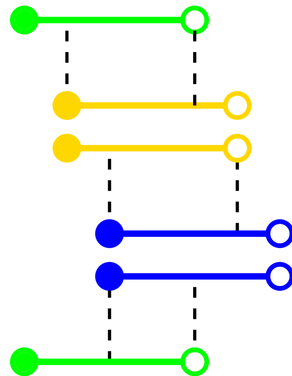
A first example



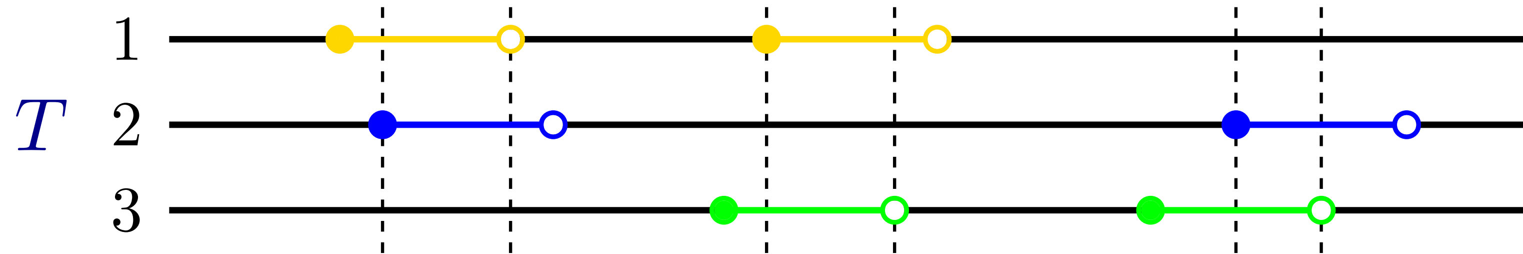


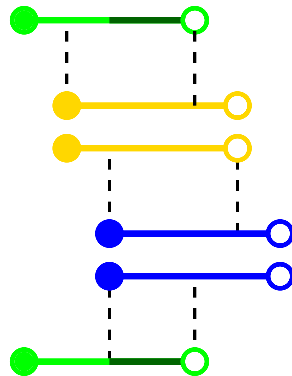
A first example





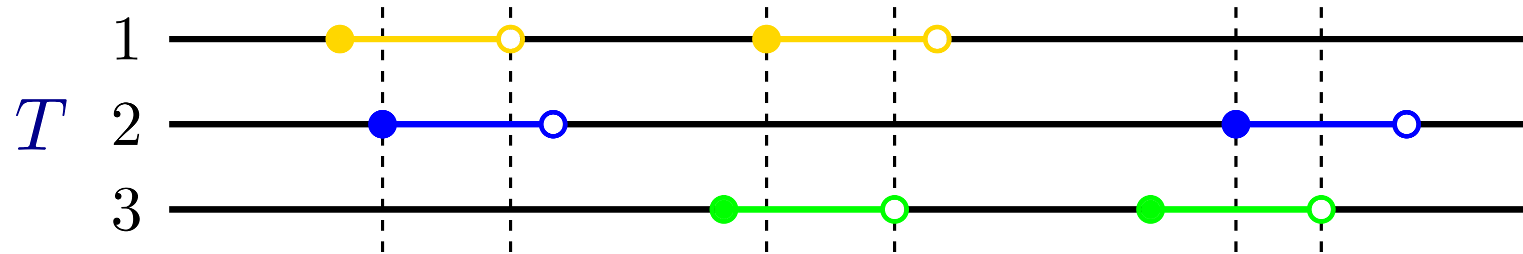
A first example

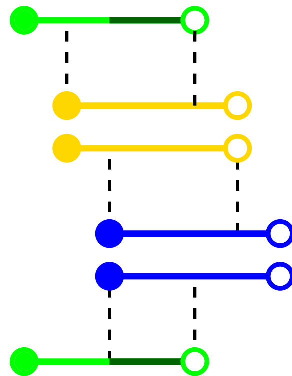




If yellow (1) and blue (2) match then the right half of green (3) matches

A first example

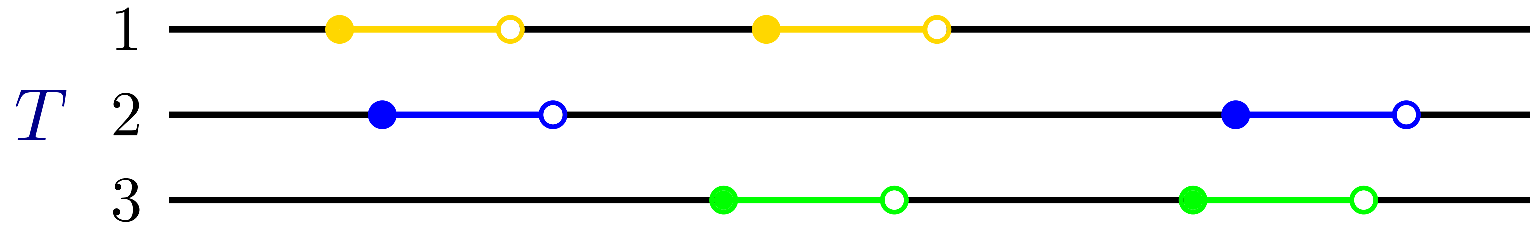




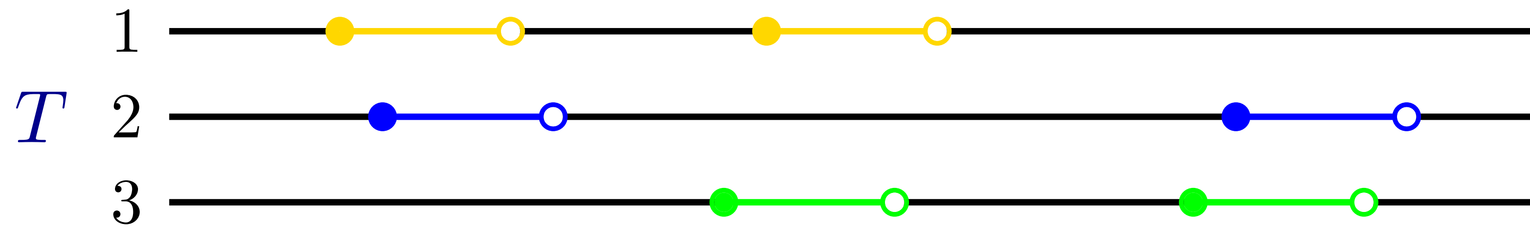
If yellow (1) and blue (2) match then the right half of green (3) matches

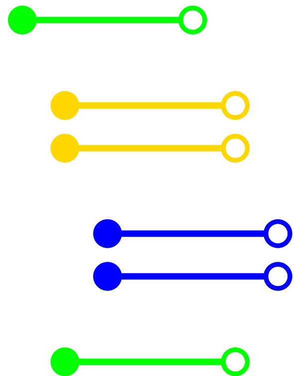
This is a *lock-stepped* cycle

A second example

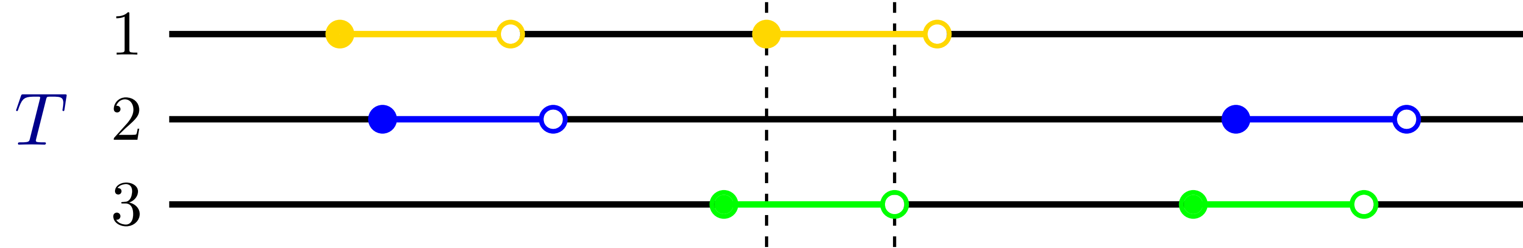


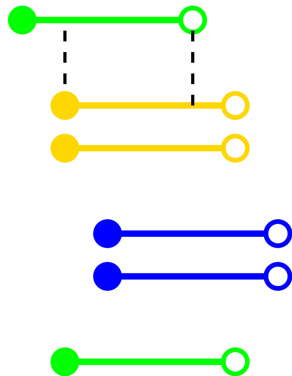
A second example



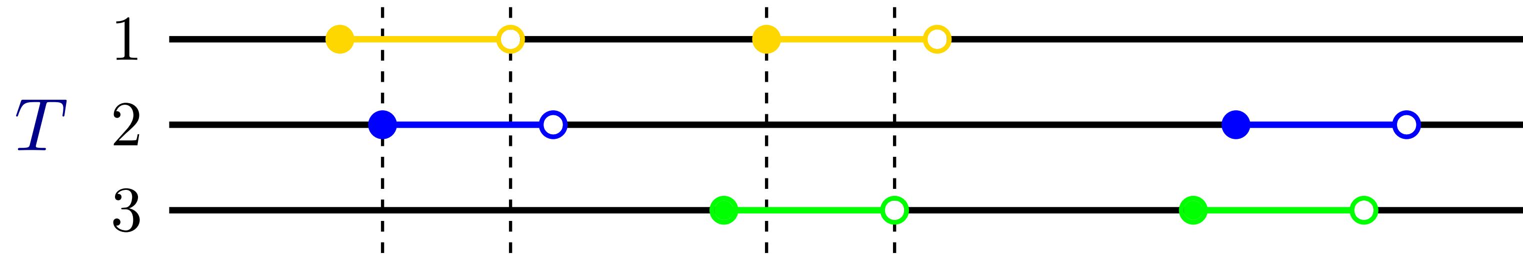


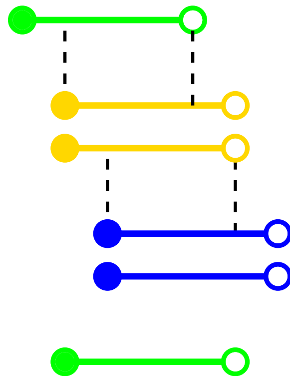
A second example



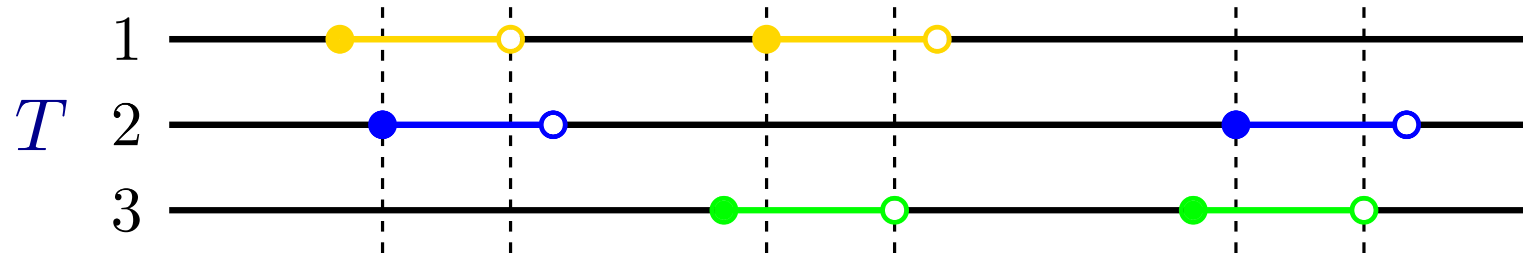


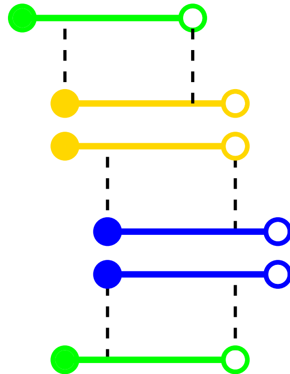
A second example



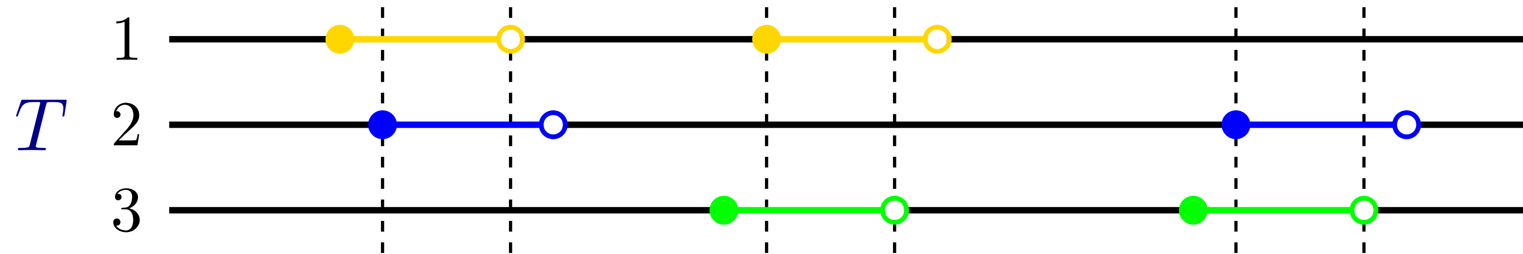


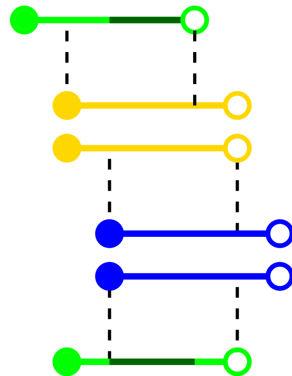
A second example





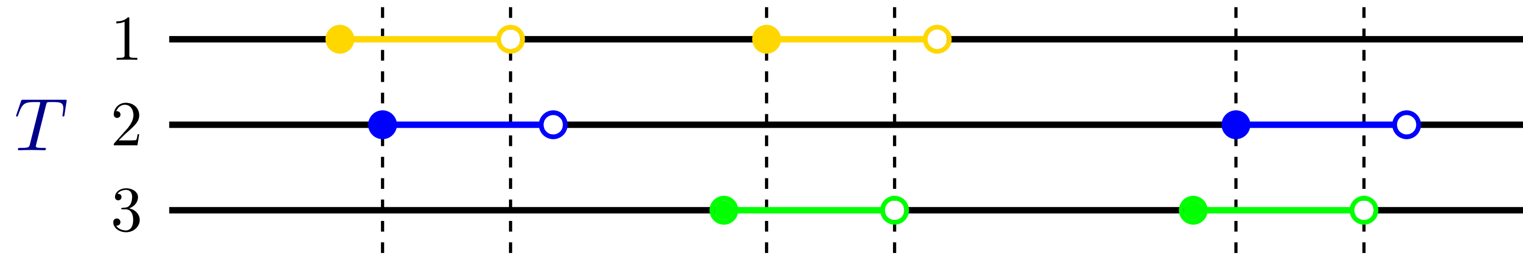
A second example

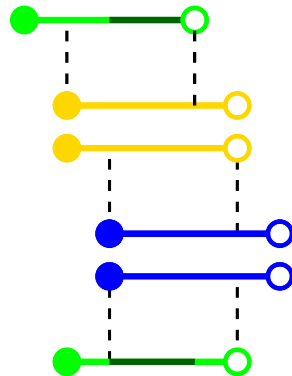




If yellow (1),blue (2) and green (3) match then the right $3/4$ of green (3) is periodic

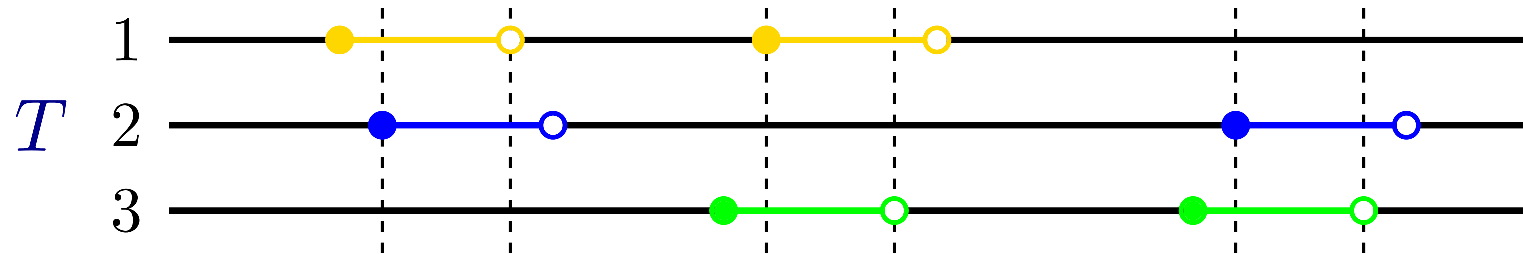
A second example

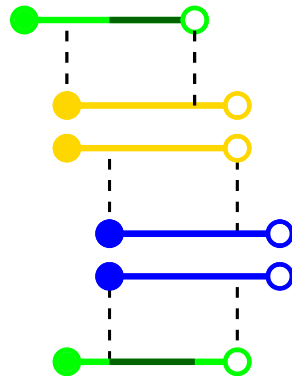




If yellow (1),blue (2) and green (3) match
then the right $^3/_4$ of green (3) is periodic

A second example

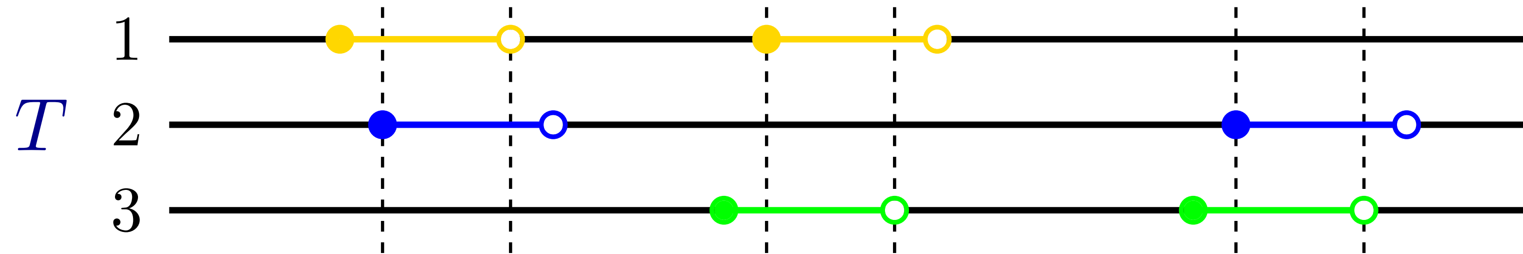


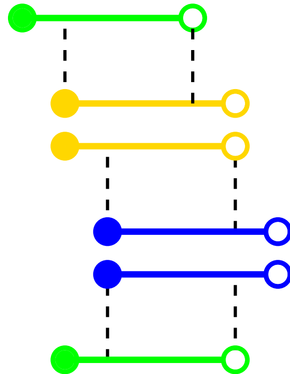


If yellow (1),blue (2) and green (3) match then the right $^3/_4$ of green (3) is periodic

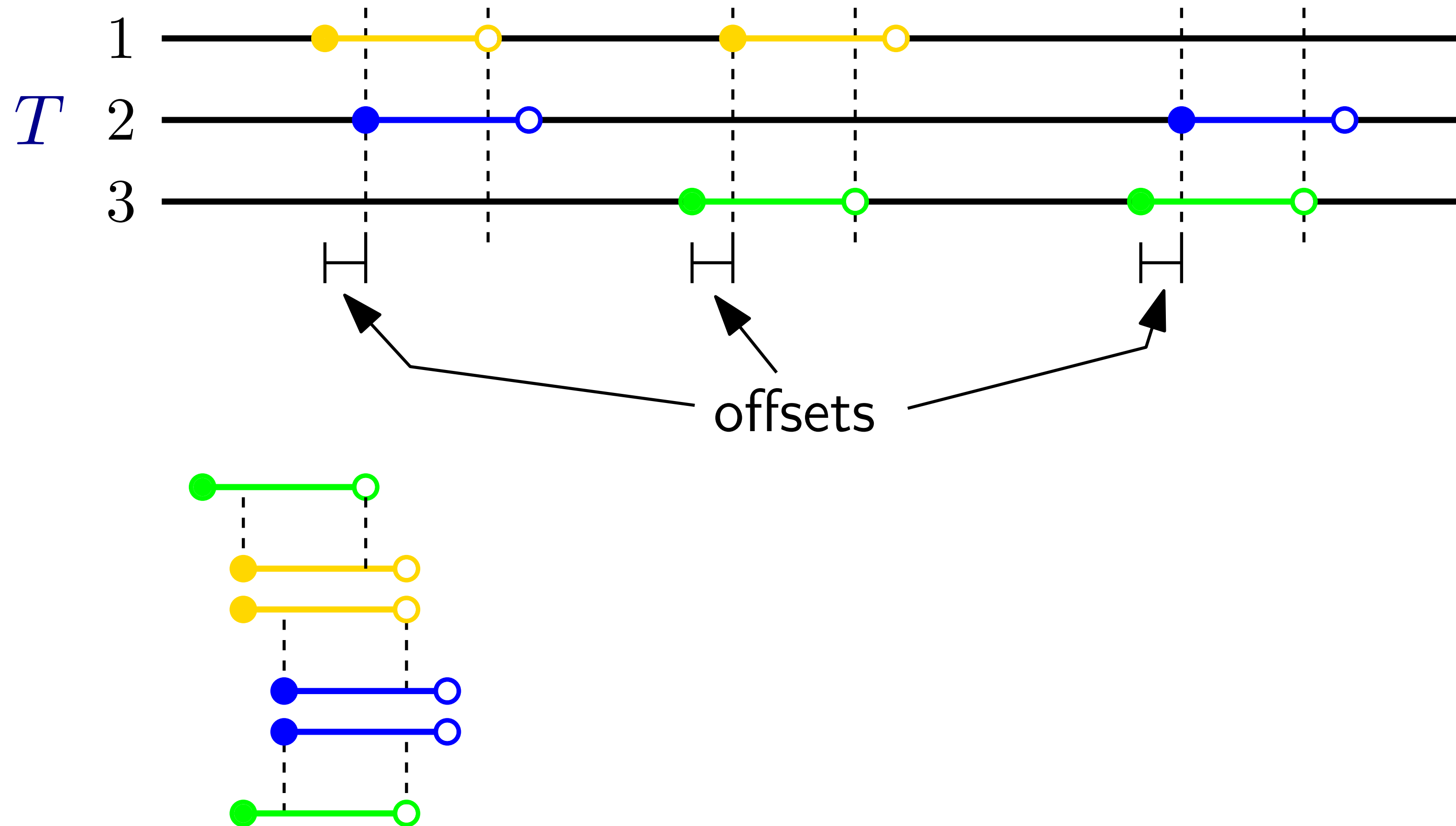
This is an *unlocked* cycle

A second example



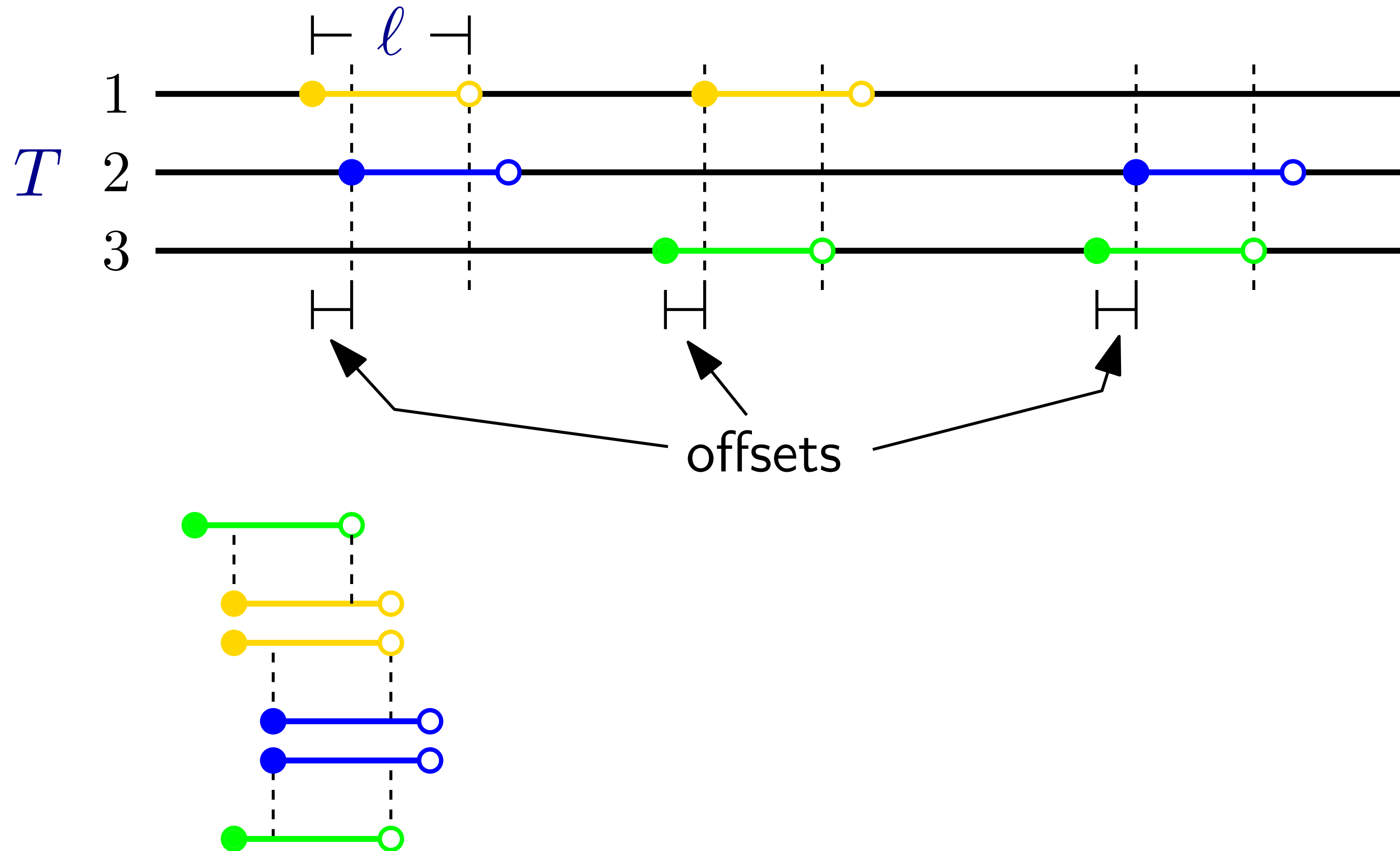


A second example



These tricks only work when the offsets are *small*

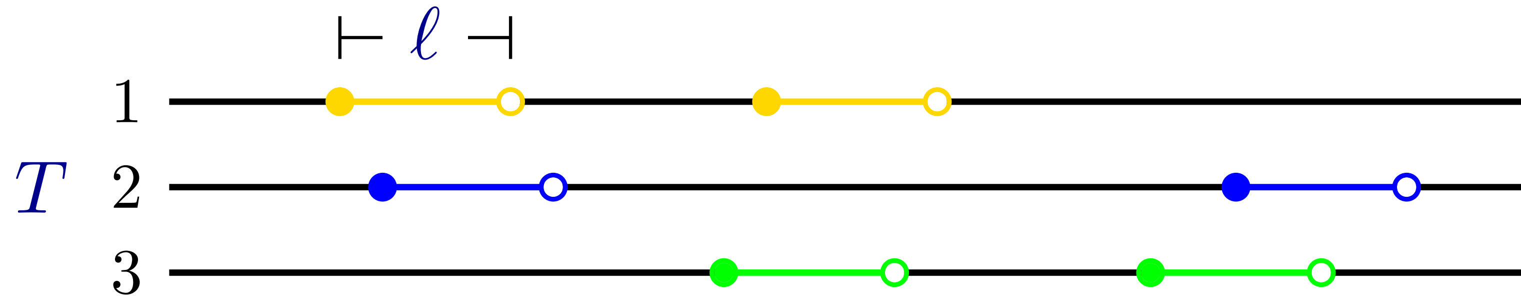
A second example



These tricks only work when the offsets are *small*

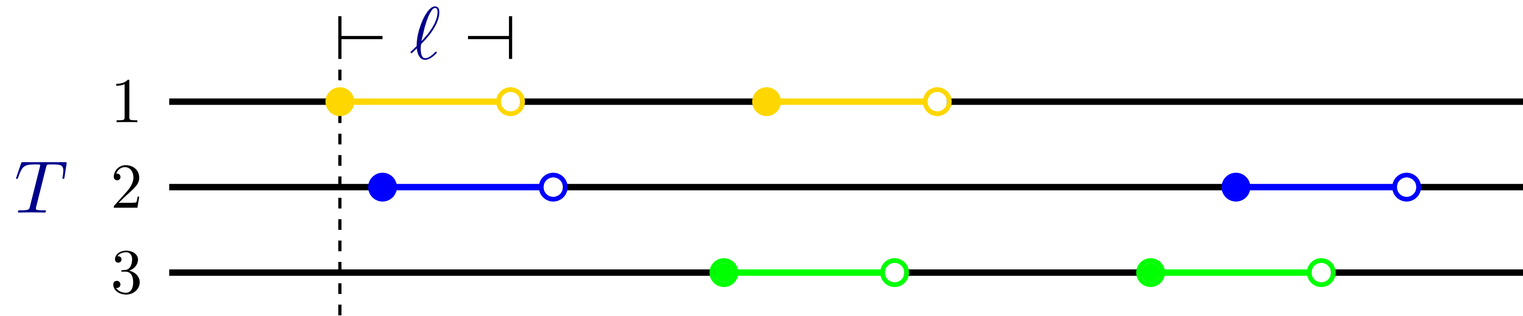
in particular when they sum to at most $\ell/4$

The overall idea

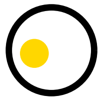


- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

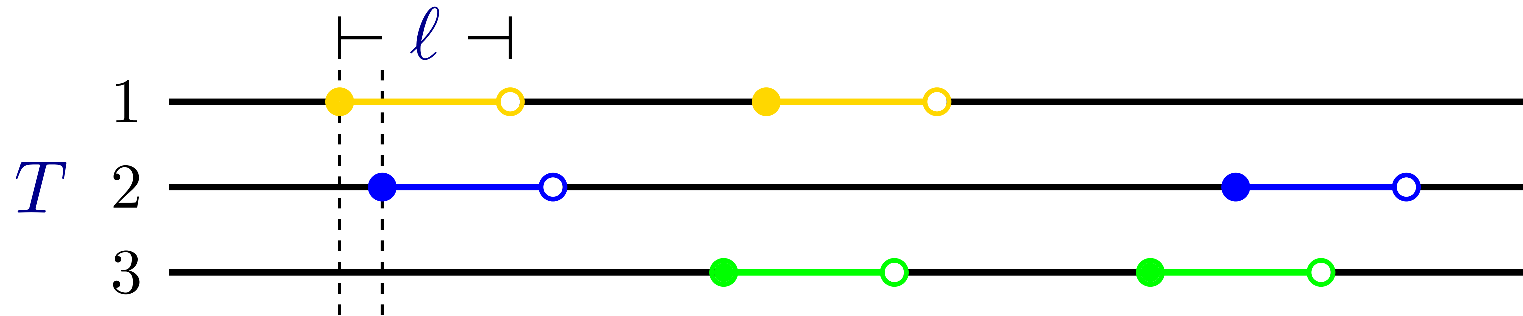
The overall idea



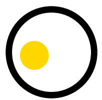
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



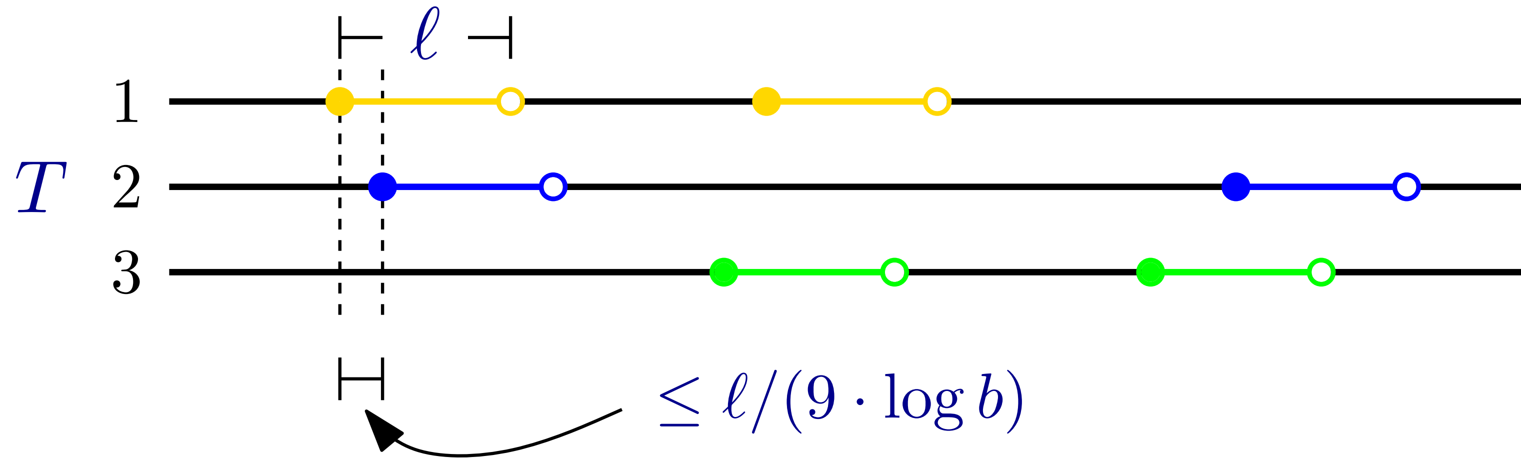
The overall idea



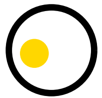
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



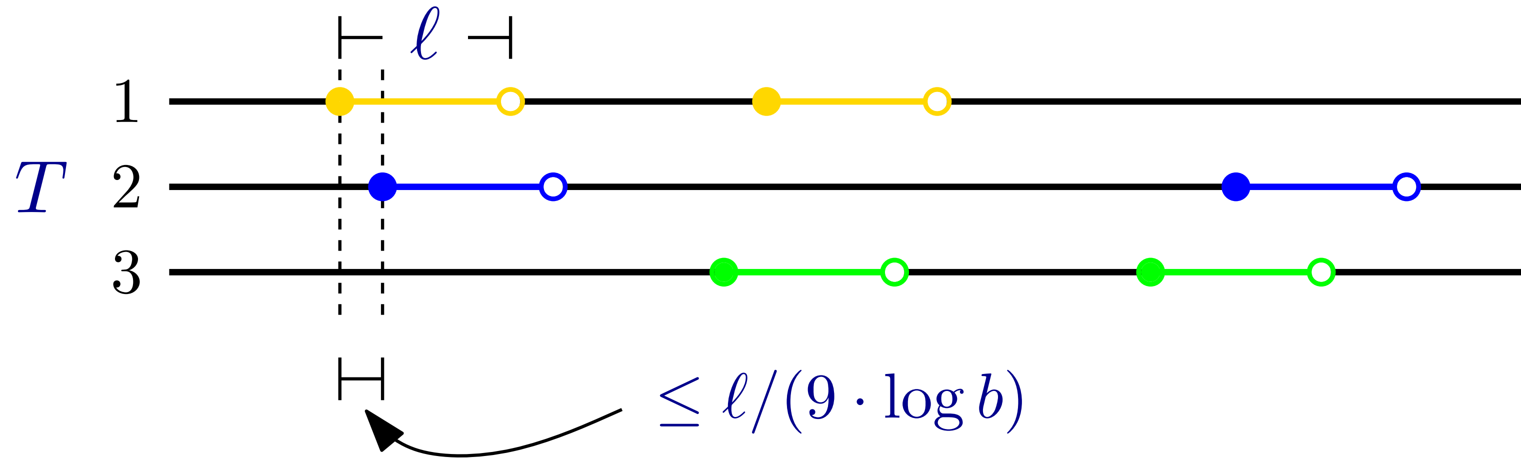
The overall idea



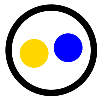
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



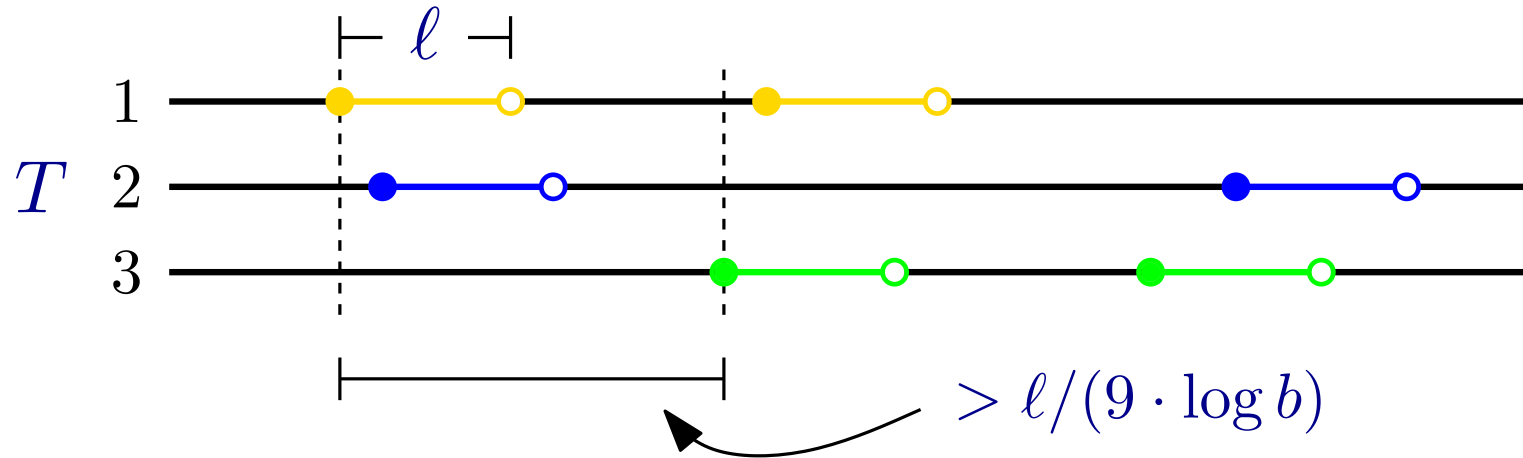
The overall idea



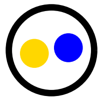
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



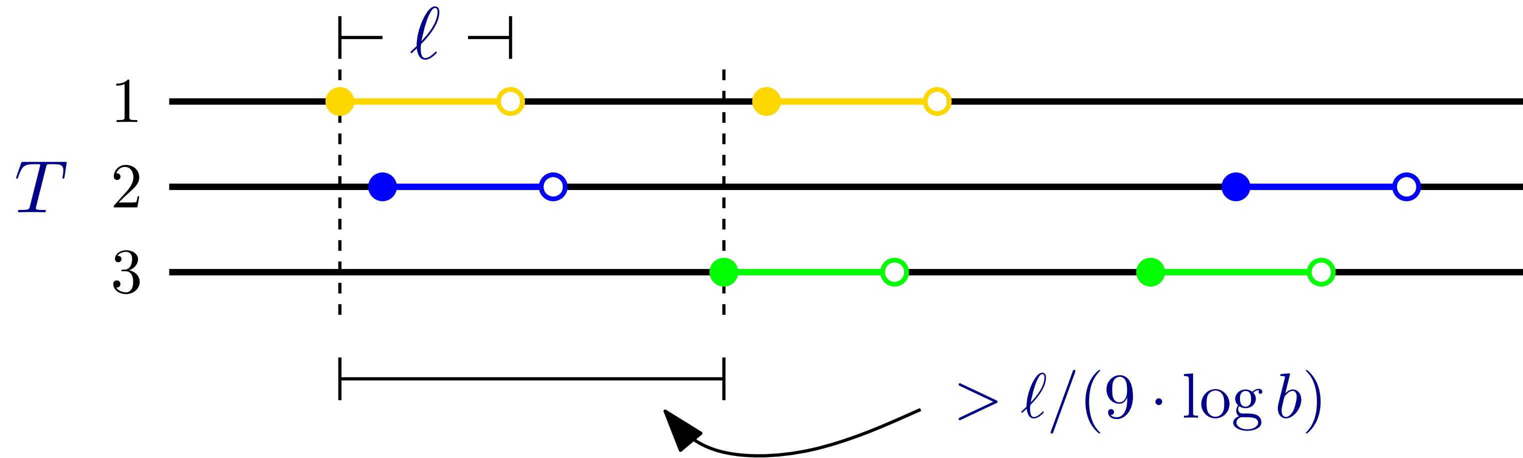
The overall idea



- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

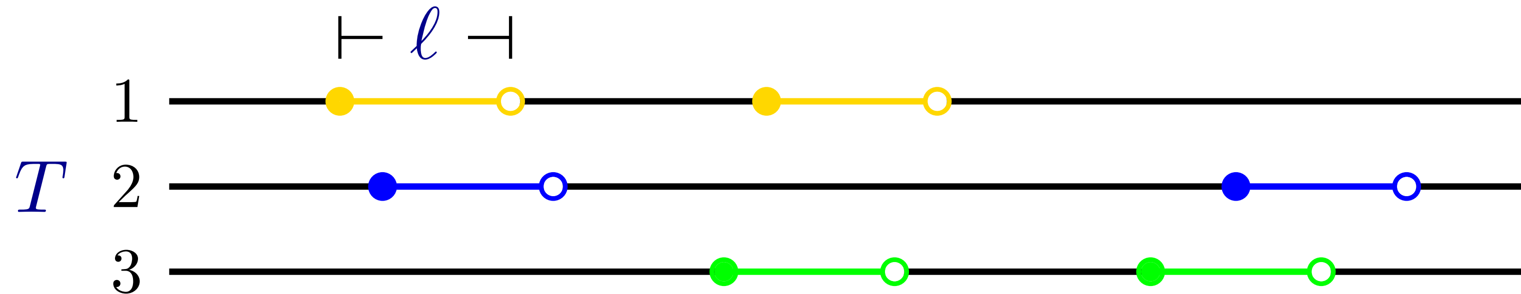


The overall idea



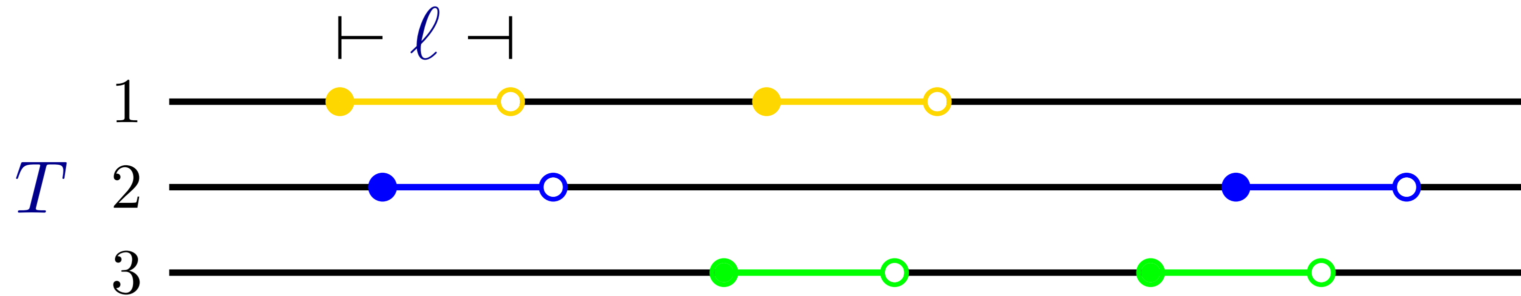
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

The overall idea



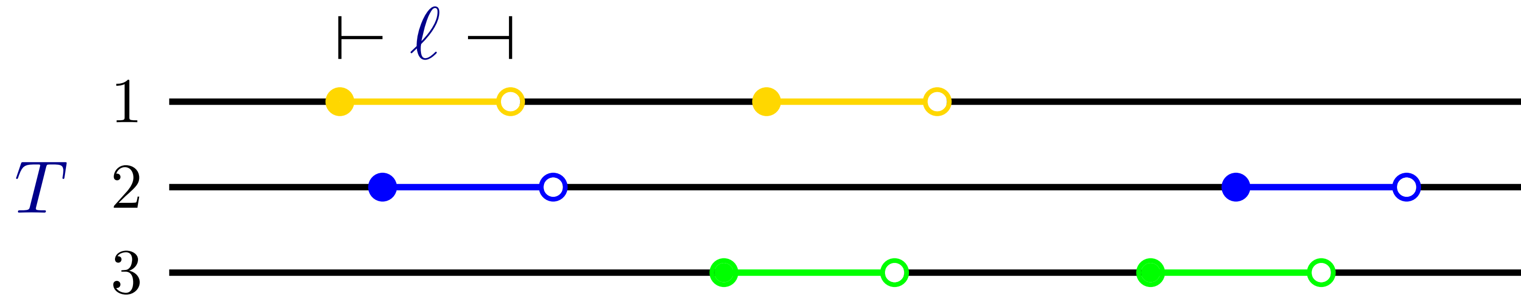
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

The overall idea

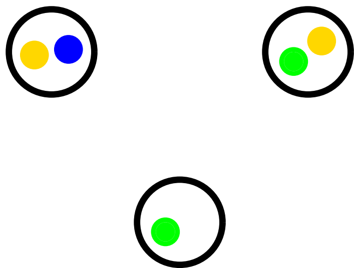


- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

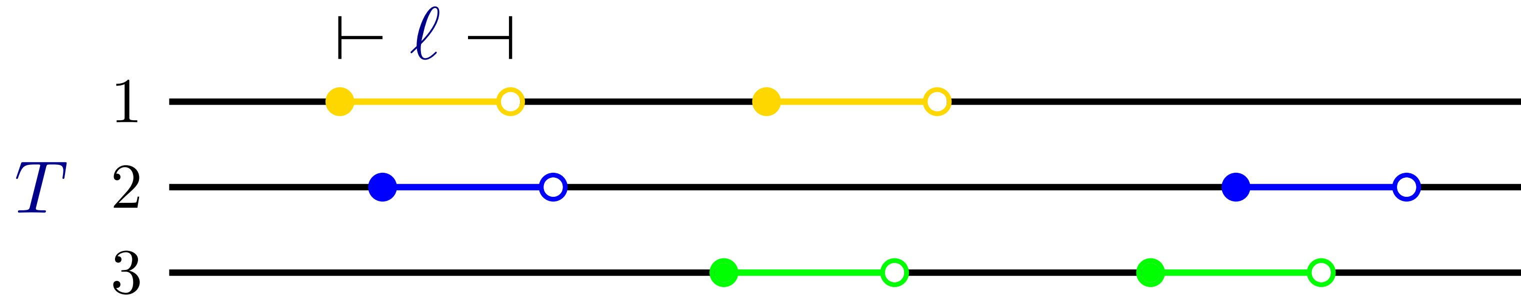
The overall idea



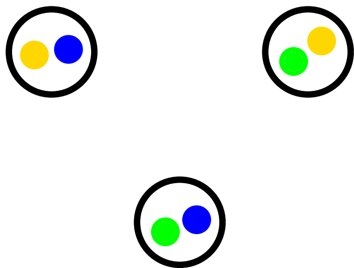
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



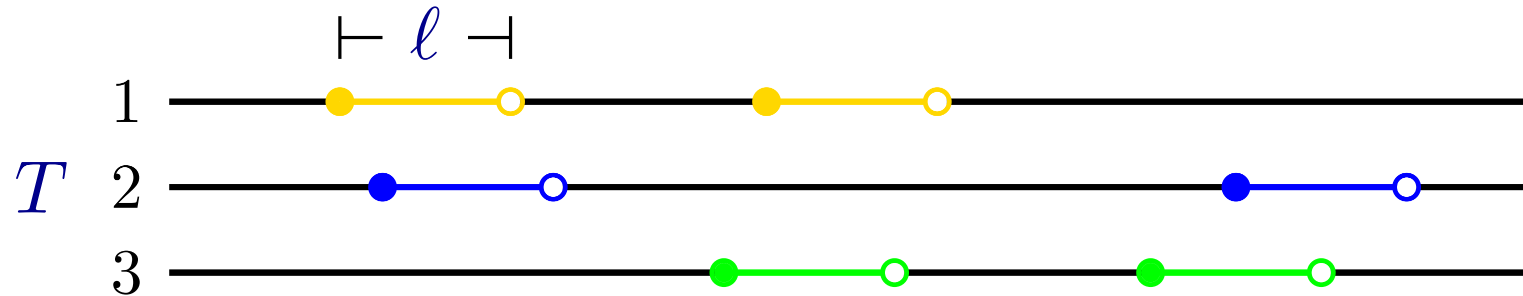
The overall idea



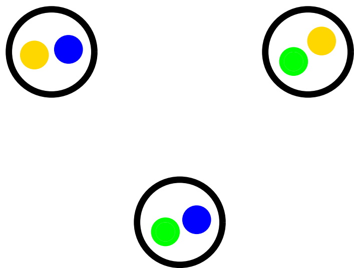
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...



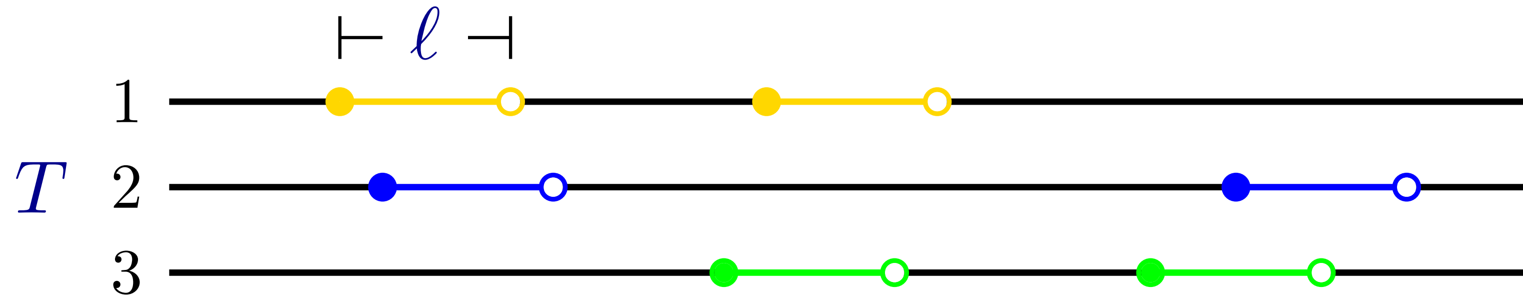
The overall idea



- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query

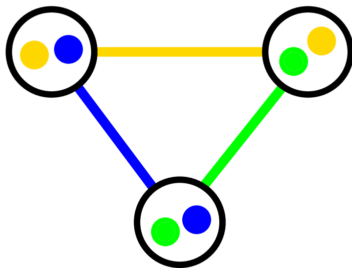


The overall idea

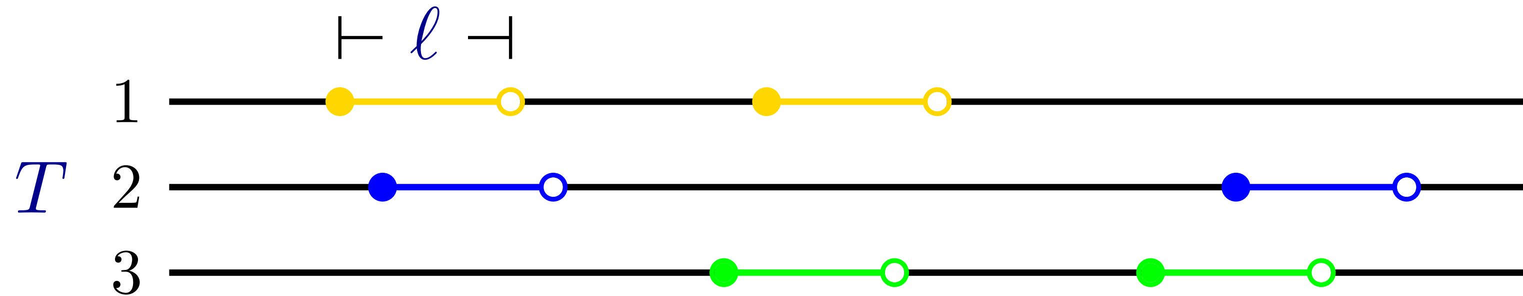


- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...

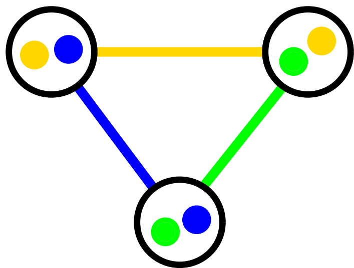
  and one edge per LCE query



The overall idea

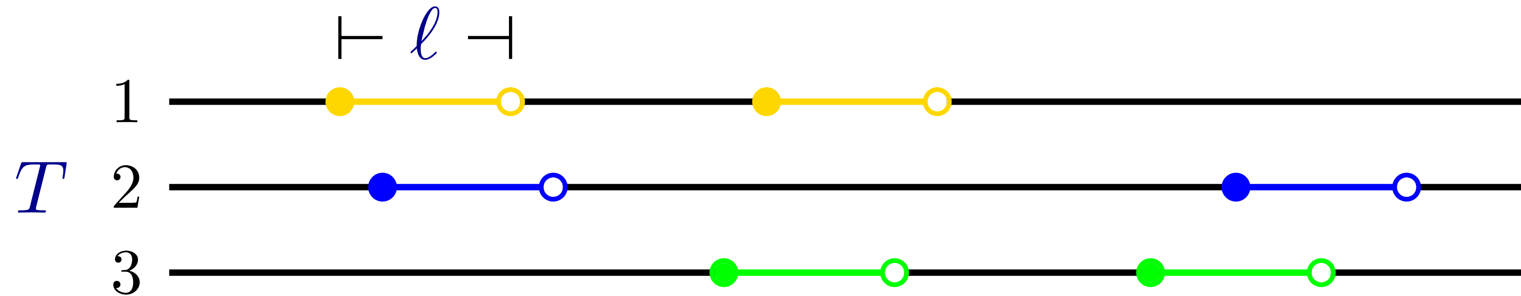


- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes... and one edge per LCE query

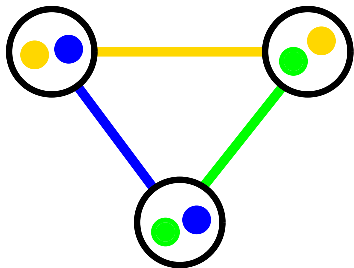


- We can apply one of the two tricks to any *short* cycle

The overall idea

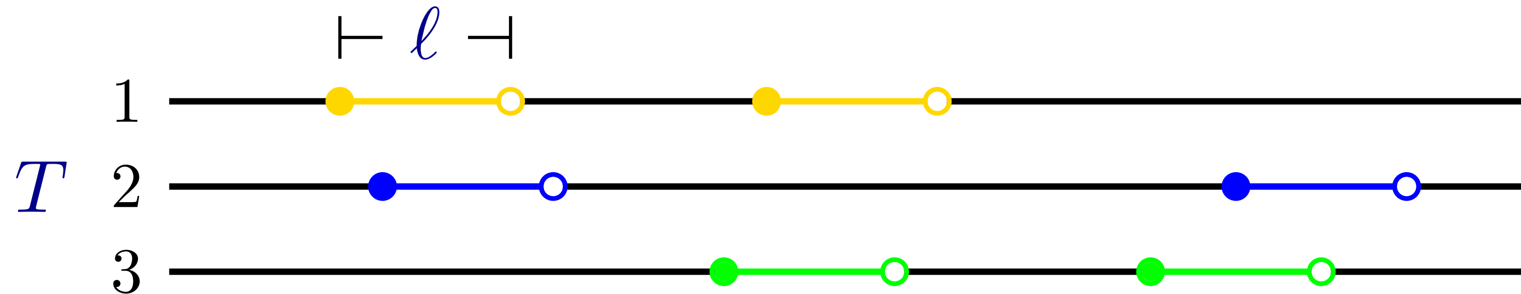


- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query

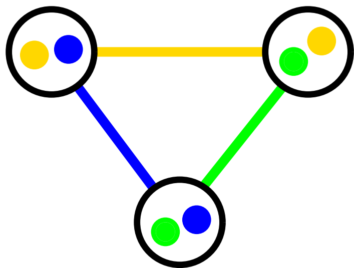


- We can apply one of the two tricks
  to any *short* cycle (length at most $2 \log b + 1$)

The overall idea

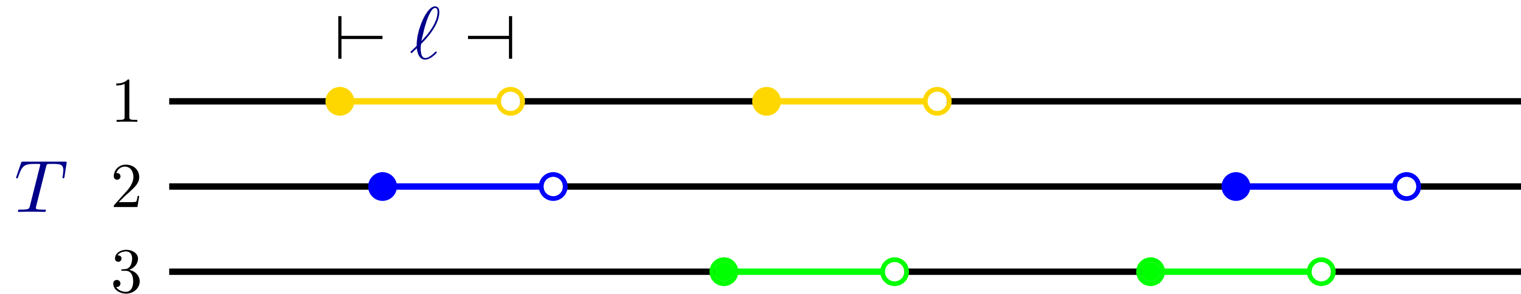


- We build a graph containing $|V| \le 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query

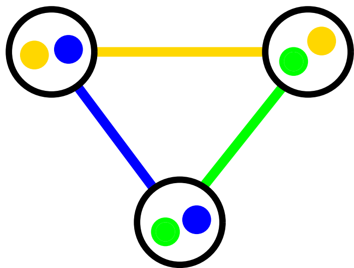


- We can apply one of the two tricks to any *short* cycle (length at most $2 \log b + 1$)
- This breaks the cycle (because we delete an edge)

The overall idea



- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query

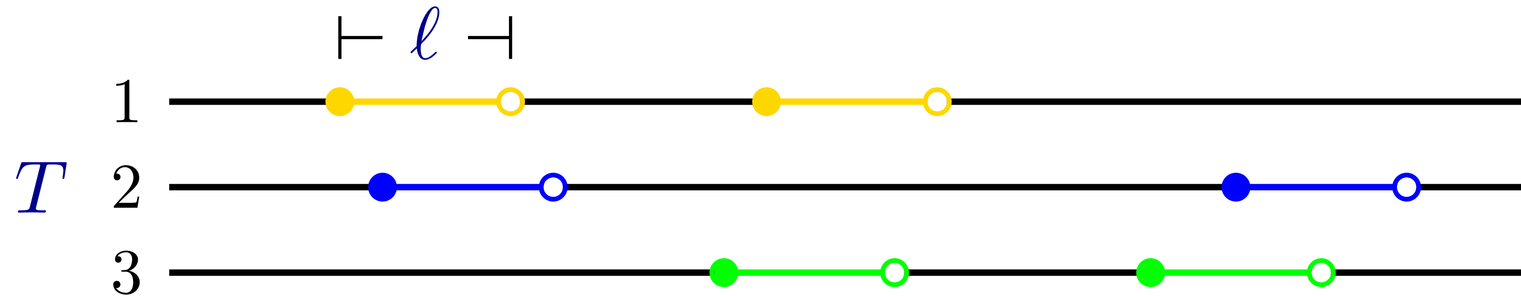


- We can apply one of the two tricks
  to any *short* cycle (length at most $2 \log b + 1$)

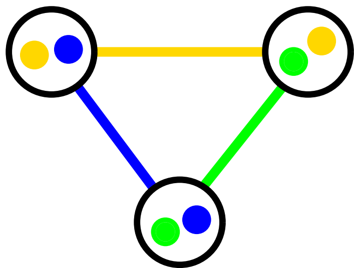
- This breaks the cycle (because we delete an edge)

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

The overall idea



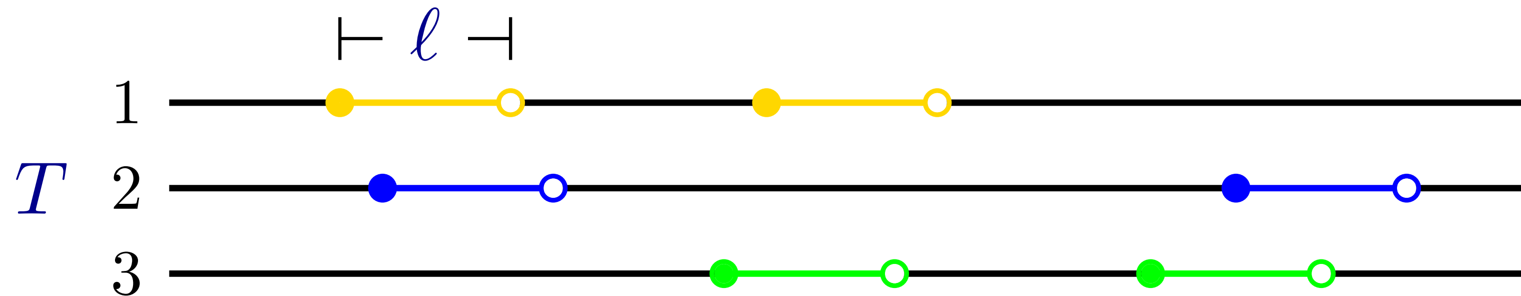
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query



- 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
(low degree nodes are easily handled)

The overall idea

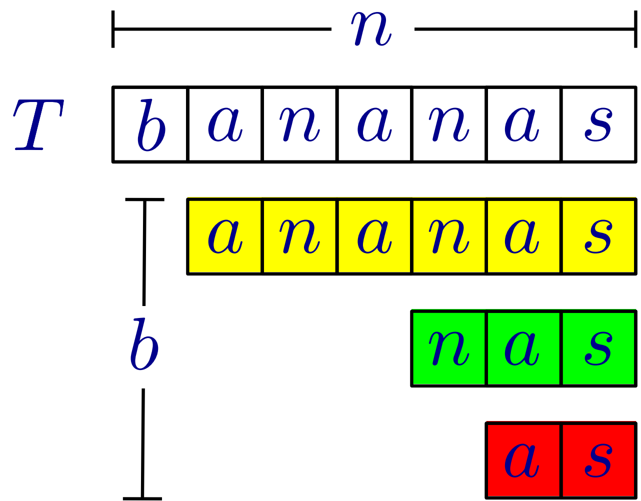


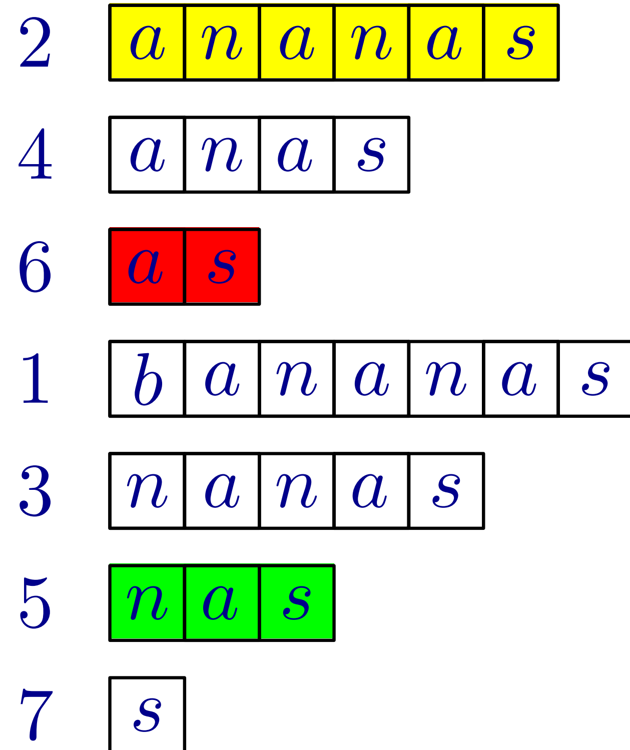
- We build a graph containing $|V| \leq 9 \cdot n/(\ell \cdot \log b)$ nodes...
  and one edge per LCE query

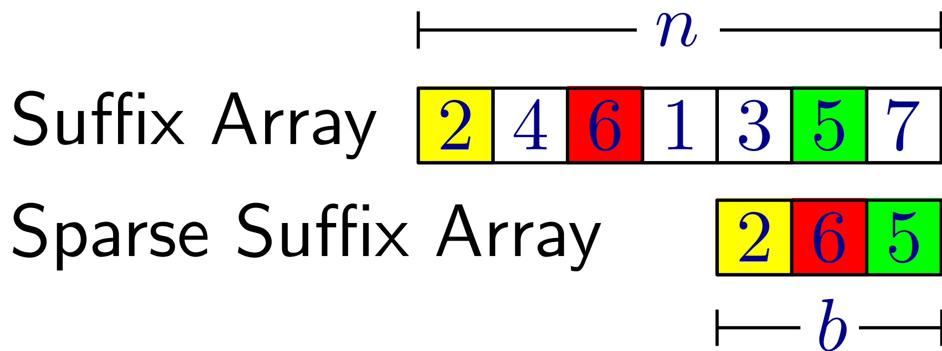
Fact If every node has degree at least three there is a short cycle

- We can find a short cycle in the graph via a BFS in $O(|E|) = O(b)$ 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