Parameterized Matching in the Streaming Model

Markus Jalsenius[1], Benny Porat[2] and *Benjamin Sach*[3]

(1) University of Bristol, UK
(2) Bar-Ilan University, Israel
(3) University of Warwick, UK

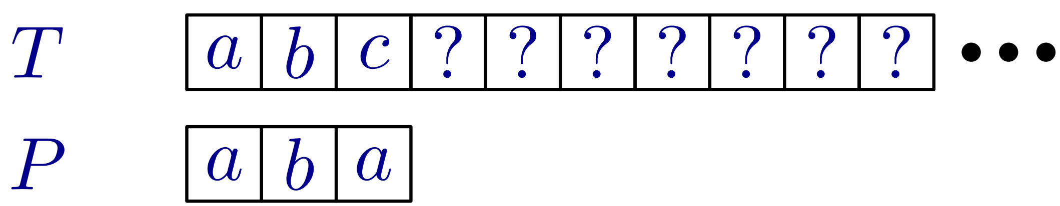
Parameterized Matching in the Streaming Model

Markus Jalsenius[1], Benny Porat[2] and *Benjamin Sach*[3]

(1) University of Bristol, UK
(2) Bar-Ilan University, Israel
(3) University of Warwick, UK

Pattern matching in the streaming model

- Consider a text string, T and a pattern P
- We assume we have P in advance but T arrives online. . .



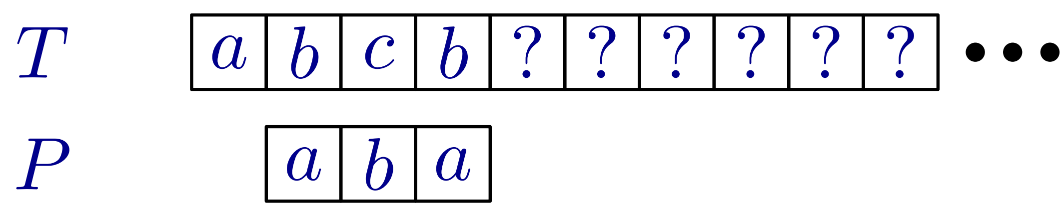
Goal: Decide whether P **matches** the last $|P|$ text characters. . .
before the next character arrives

- The definition of a **match** depends on the problem

- We care about **worst-case** time per text character
and using as little space as possible

Pattern matching in the streaming model

- Consider a text string, T and a pattern P
- We assume we have P in advance but T arrives online...



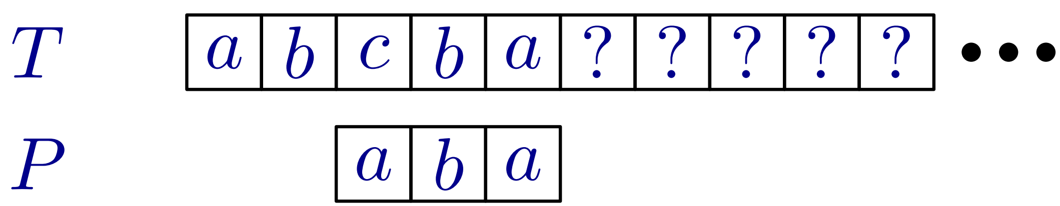
Goal: Decide whether P **matches** the last $|P|$ text characters...
before the next character arrives

- The definition of a **match** depends on the problem

- We care about **worst-case** time per text character
and using as little space as possible

Pattern matching in the streaming model

- Consider a text string, T and a pattern P
- We assume we have P in advance but T arrives online...



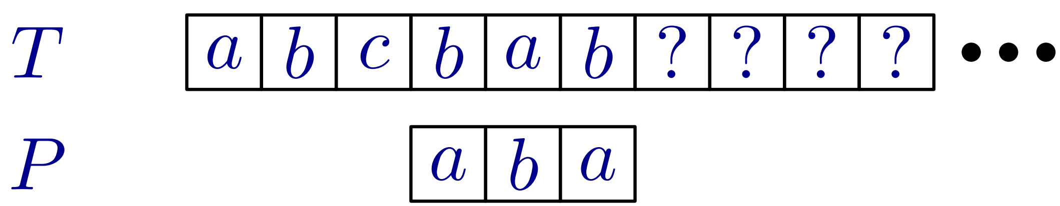
Goal: Decide whether P **matches** the last $|P|$ text characters...
before the next character arrives

- The definition of a **match** depends on the problem

- We care about **worst-case** time per text character
and using as little space as possible

Pattern matching in the streaming model

- Consider a text string, T and a pattern P
- We assume we have P in advance but T arrives online. . .



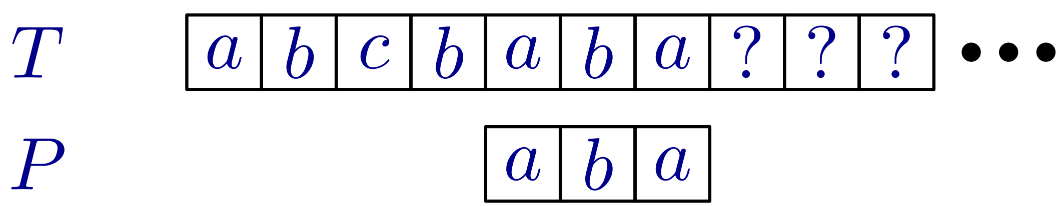
Goal: Decide whether P **matches** the last $|P|$ text characters. . .
before the next character arrives

- The definition of a **match** depends on the problem

- We care about **worst-case** time per text character
and using as little space as possible

Pattern matching in the streaming model

- Consider a text string, T and a pattern P
- We assume we have P in advance but T arrives online. . .



Goal: Decide whether P **matches** the last $|P|$ text characters. . .
before the next character arrives

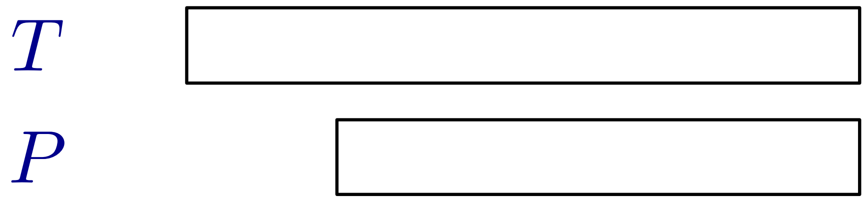
- The definition of a **match** depends on the problem

- We care about **worst-case** time per text character
and using as little space as possible

Streaming pattern matching in small space

What's the least space we can get away with?

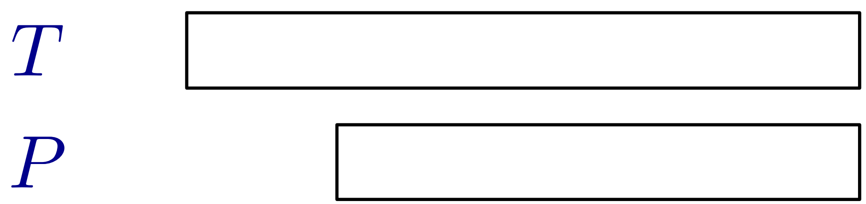
- *Surely*, we have to store the pattern?



Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



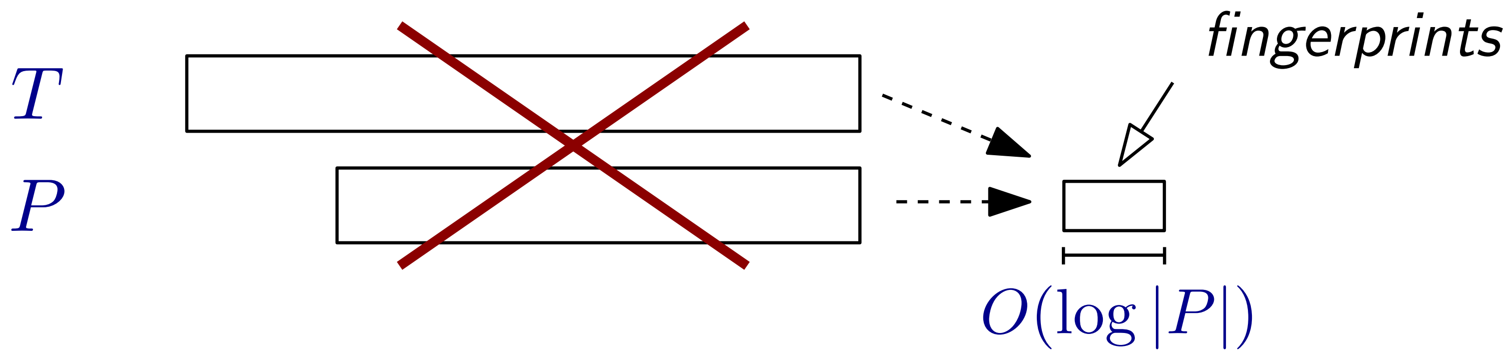
Porat, Porat FOCS'09

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(\log |P|)$ time per character

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



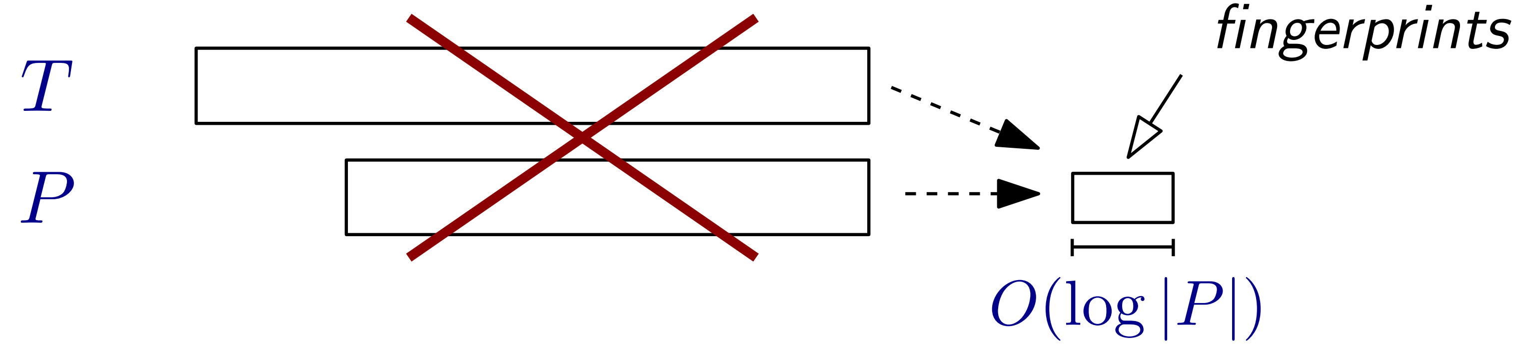
Porat, Porat FOCS'09

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(\log |P|)$ time per character

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(\log |P|)$ time per character

The algorithm is randomised,

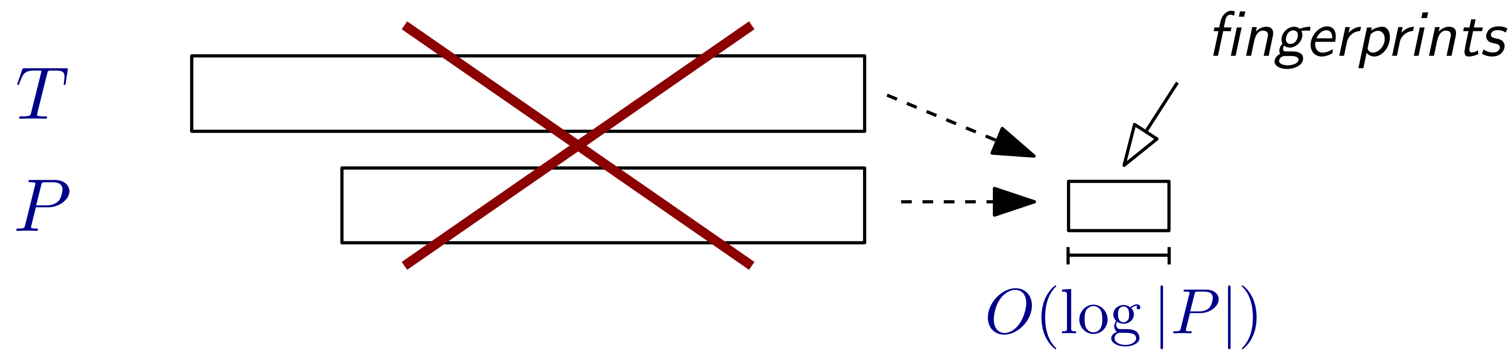
- it might make mistakes but it's correct with high probability

(at least $1 - 1/|T|^3$)

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(\log |P|)$ time per character

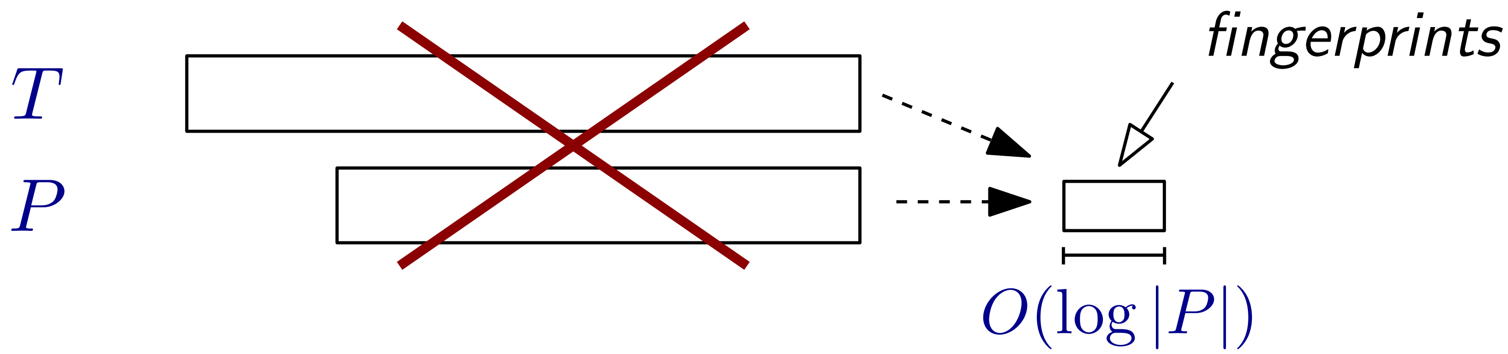
The algorithm is randomised,

- it might make mistakes but it's correct with high probability

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

Breslauer, Galil CPM'11

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(1)$ time per character

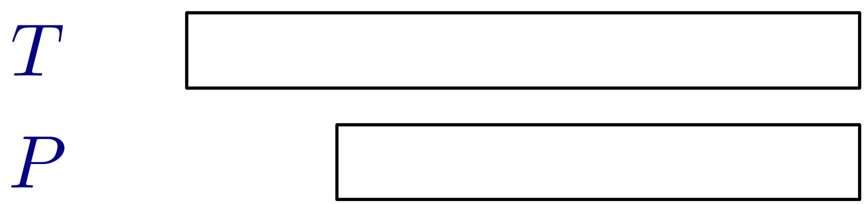
The algorithm is randomised,

- it might make mistakes but it's correct with high probability

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

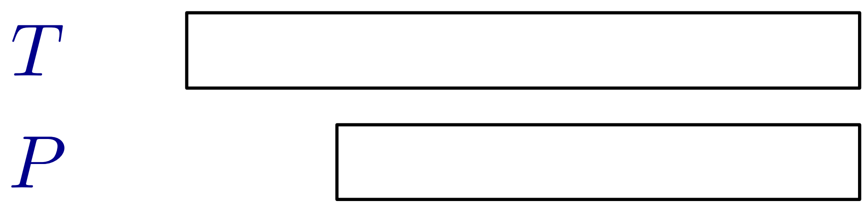
Breslauer, Galil CPM'11

- Exact pattern matching can be solved in $O(\log |P|)$ space and $O(1)$ time per character

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

Breslauer, Galil CPM'11

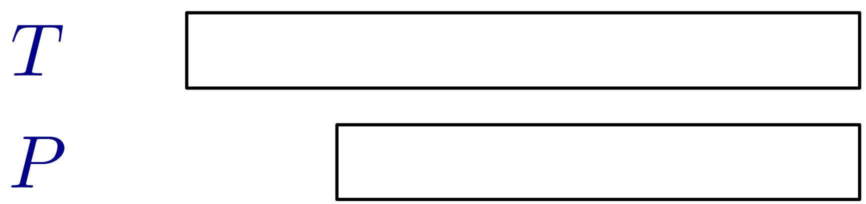
- Exact pattern matching can be solved in
  $O(\log |P|)$ space and $O(1)$ time per character

- Pattern matching with k mismatches can be solved in
  $O(k^3 \text{polylog}|P|)$ space and $O(k^2 \text{polylog}|P|)$ time per character

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely,* we have to store the pattern?



Porat, Porat FOCS'09

Breslauer, Galil CPM'11

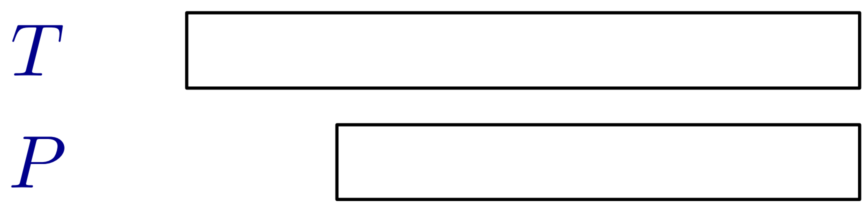
- Exact pattern matching can be solved in
  $O(\log |P|)$ space and $O(1)$ time per character

- Pattern matching with k mismatches can be solved in
  $O(k^3 \text{polylog}|P|)$ space and $O(k^2 \text{polylog}|P|)$ time per character
  (this algorithm is also randomised)

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

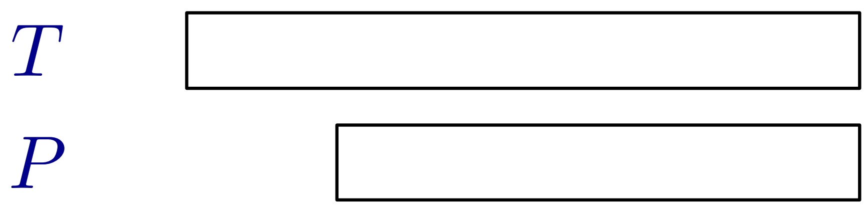
Breslauer, Galil CPM'11

- Exact pattern matching can be solved in $O(\log|P|)$ space and $O(1)$ time per character
- Pattern matching with k mismatches can be solved in $O(k^3 \text{polylog}|P|)$ space and $O(k^2 \text{polylog}|P|)$ time per character

Streaming pattern matching in small space

What's the least space we can get away with?

- *Surely*, we have to store the pattern?



Porat, Porat FOCS'09

Breslauer, Galil CPM'11

- Exact pattern matching can be solved in $O(\log|P|)$ space and $O(1)$ time per character
- Pattern matching with k mismatches can be solved in $O(k^3 \text{polylog}|P|)$ space and $O(k^2 \text{polylog}|P|)$ time per character

What else can be solved in small space?

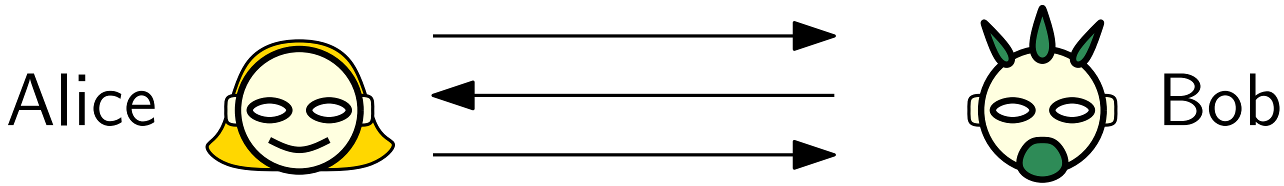
Streaming pattern matching in small space

Clifford, Jalsenius, Porat, **S.** *CPM'11*

We showed randomised space lower bounds of $\Omega(|P|)$ bits for:

Hamming distance, *Exact matching with wildcards*, L_1, L_2 *and* L_∞ *distances*, *Edit distance and Swap matching*

as well as any algorithm performing convolutions

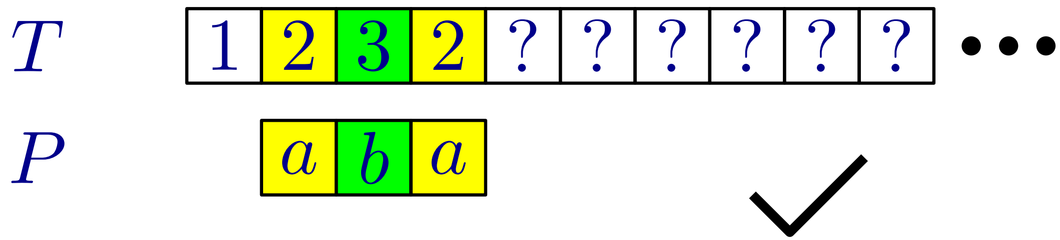


We did this by showing that better algorithms would give impossibly good communication protocols

Parameterized matching in a stream

- In parameterized matching (p-matching),

  the alphabet can be relabelled



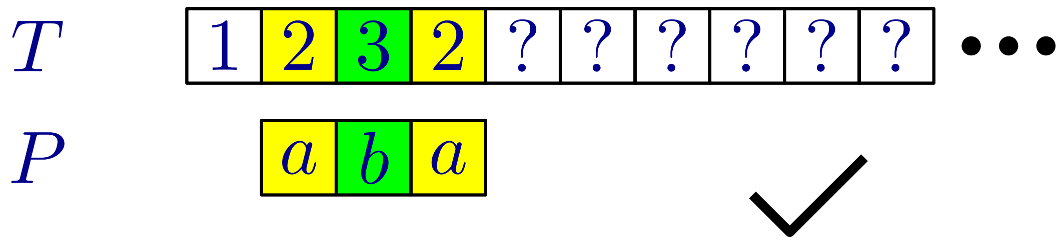
- The alphabet mapping can differ for each pattern/text alignment
- The mapping must be one-to-one (injective)

P p-matches $T[i, i + |P| - 1]$ iff there is
a one-to-one f s.t. $f(P[j]) = T[i + j]$ for all j

Parameterized matching in a stream

- In parameterized matching (p-matching),

  the alphabet can be relabelled



$a \to 2$, $b \to 3$ gives a mapping

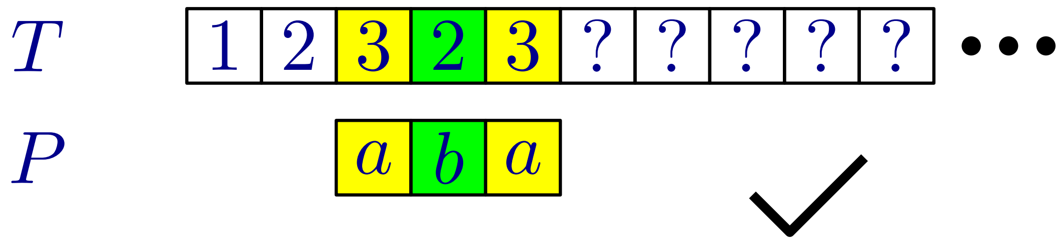
- The alphabet mapping can differ for each pattern/text alignment
- The mapping must be one-to-one (injective)

P p-matches $T[i, i + |P| - 1]$ iff there is
a one-to-one f s.t. $f(P[j]) = T[i + j]$ for all j

Parameterized matching in a stream

- In parameterized matching (p-matching),

  the alphabet can be relabelled



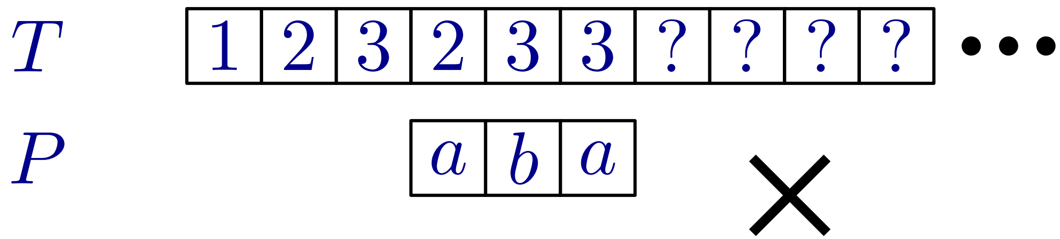
$a \to 3$, $b \to 2$ gives a mapping

- The alphabet mapping can differ for each pattern/text alignment
- The mapping must be one-to-one (injective)

P p-matches $T[i, i + |P| - 1]$ iff there is
a one-to-one f s.t. $f(P[j]) = T[i + j]$ for all j

Parameterized matching in a stream

- In parameterized matching (p-matching),
  the alphabet can be relabelled



there is no mapping - a can't map to 2 *and* 3

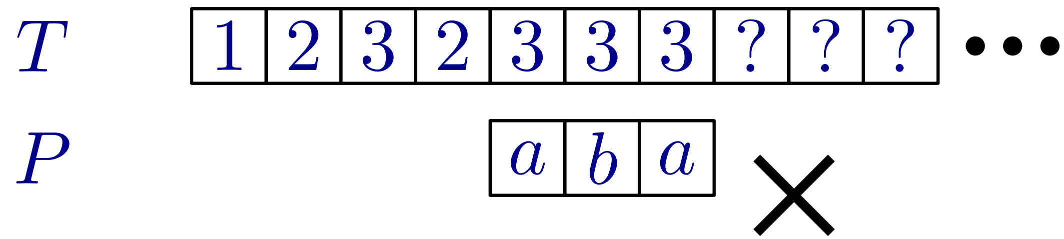
- The alphabet mapping can differ for each pattern/text alignment
- The mapping must be one-to-one (injective)

P p-matches $T[i, i + |P| - 1]$ iff there is
a one-to-one f s.t. $f(P[j]) = T[i + j]$ for all j

Parameterized matching in a stream

- In parameterized matching (p-matching),

  the alphabet can be relabelled



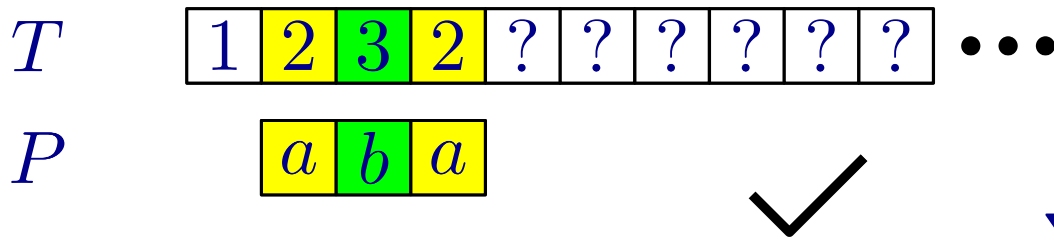
there is no mapping - it has to be one-to-one

- The alphabet mapping can differ for each pattern/text alignment
- The mapping must be one-to-one (injective)

P p-matches $T[i, i + |P| - 1]$ iff there is
a one-to-one f s.t. $f(P[j]) = T[i + j]$ for all j

Parameterized matching in a stream

- In parameterized matching (p-matching),
  the alphabet can be relabelled



Σ is the alphabet

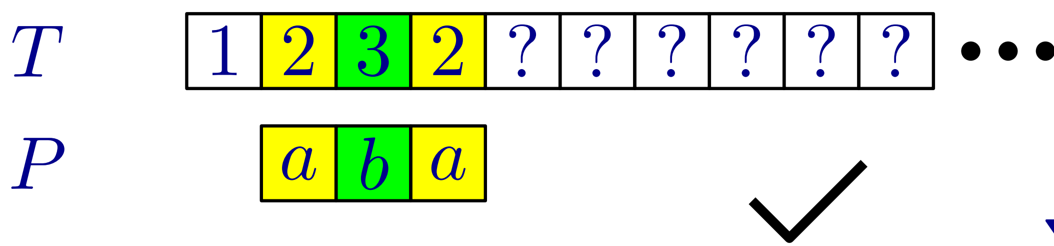
Our Results

Parameterized matching can be solved in $O(|\Sigma| \log |P|)$ space and:

- $O(1)$ time per character when $|\Sigma| = \{1, 2, 3, \ldots, |\Sigma|\}$
- $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$ time per character for general Σ

Parameterized matching in a stream

- In parameterized matching (p-matching),
  the alphabet can be relabelled



Σ is the alphabet

Our Results

Parameterized matching can be solved in $O(|\Sigma| \log |P|)$ space and:

- $O(1)$ time per character when $|\Sigma| = \{1, 2, 3, \ldots, |\Sigma|\}$
- $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$ time per character for general Σ

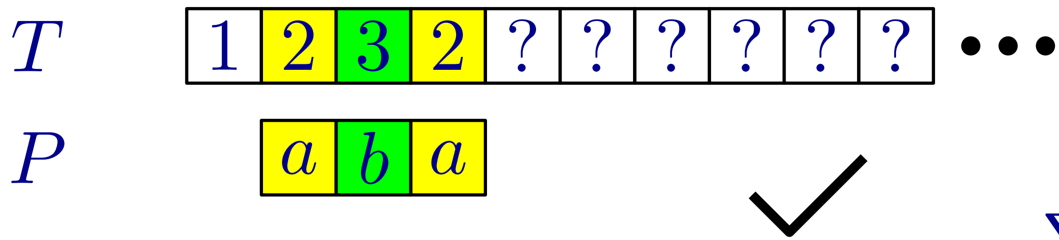
Both algorithms are randomised (Monte-Carlo)

We also give an $\Omega(|\Sigma|)$ bit randomised space lower bound

Parameterized matching in a stream

- In parameterized matching (p-matching),

the alphabet can be relabelled



Σ is the alphabet

Our Results

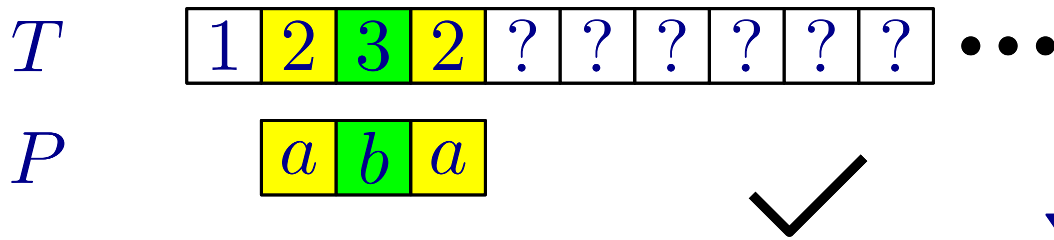
Parameterized matching can be solved in $O(|\Sigma| \log |P|)$ space and:

- $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$ time per character for general Σ

Parameterized matching in a stream

- In parameterized matching (p-matching),

the alphabet can be relabelled



Σ is the alphabet

Our Results

Parameterized matching can be solved in $O(|\Pi| \log |P|)$ space and:

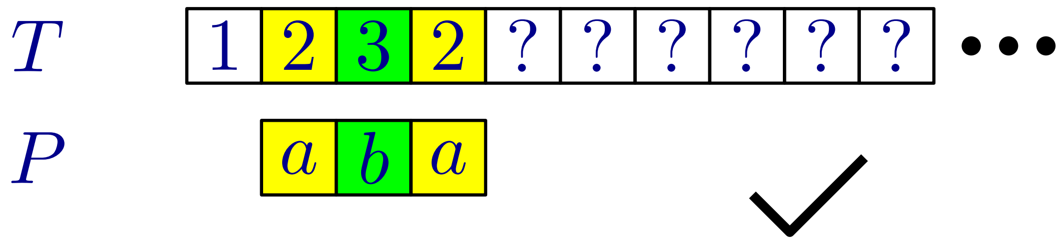
- $O(\sqrt{\log |\Pi| / \log \log |\Pi|})$ time per character for general Π

where $\Pi \subseteq \Sigma$ is the set of symbols in P
which are *allowed* to be relabelled

Parameterized matching in a stream

- In parameterized matching (p-matching),

  the alphabet can be relabelled



Parameterized matching can be solved in $O(|\Sigma| \log |P|)$ space and:

- $O(1)$ time per character when $|\Sigma| = \{1, 2, 3, \ldots, |\Sigma|\}$

The remainder of the talk will focus on this result
(the others are simple generalisations)

Predecessor Strings

S_1	a	b	a	c	c	b	a	b	c	b
S_2	b	c	b	a	a	c	b	c	a	c

Predecessor Strings

S_1	a	b	a	c	c	b	a	b	c	b
S_2	b	c	b	a	a	c	b	c	a	c

S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

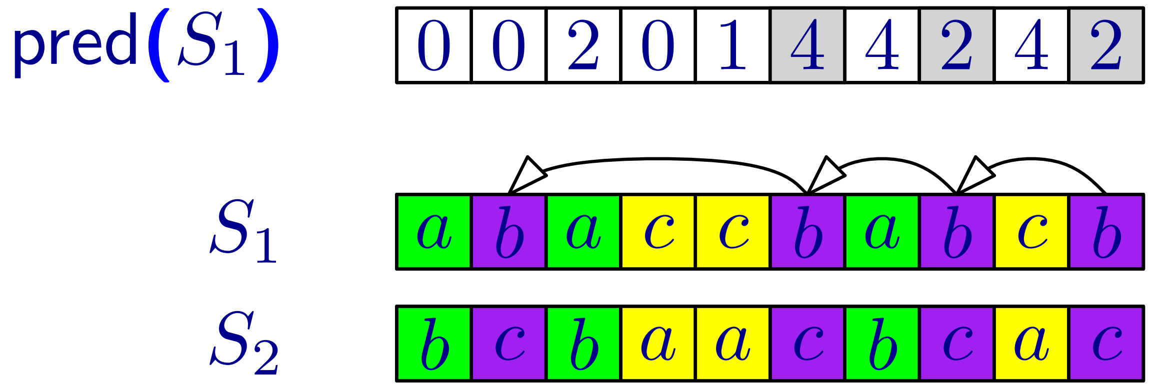
Predecessor Strings

pred(S_1)	0	0	2	0	1	4	4	2	4	2

S_1	a	b	a	c	c	b	a	b	c	b
S_2	b	c	b	a	a	c	b	c	a	c

S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

Predecessor Strings



S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

Predecessor Strings

pred(S_1)

0	0	2	0	1	4	4	2	4	2

S_1

a	b	a	c	c	b	a	b	c	b

S_2

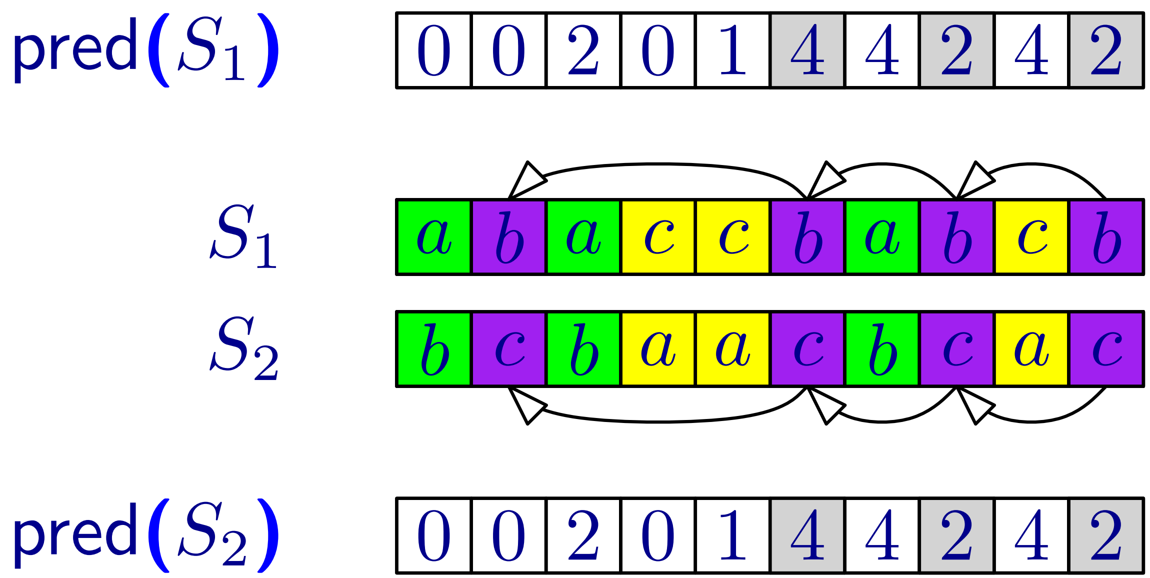
b	c	b	a	a	c	b	c	a	c

pred(S_2)

0	0	2	0	1	4	4	2	4	2

S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

Predecessor Strings



S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

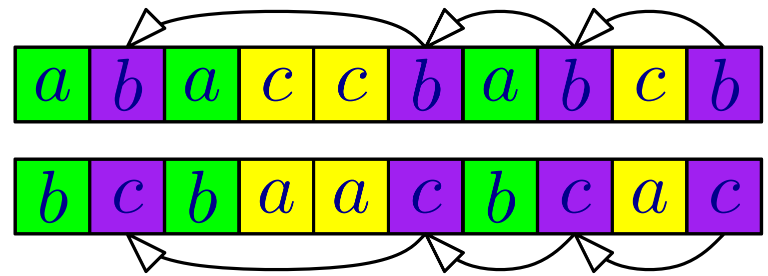
Predecessor Strings

pred(S_1)

0	0	2	0	1	4	4	2	4	2

S_1

S_2



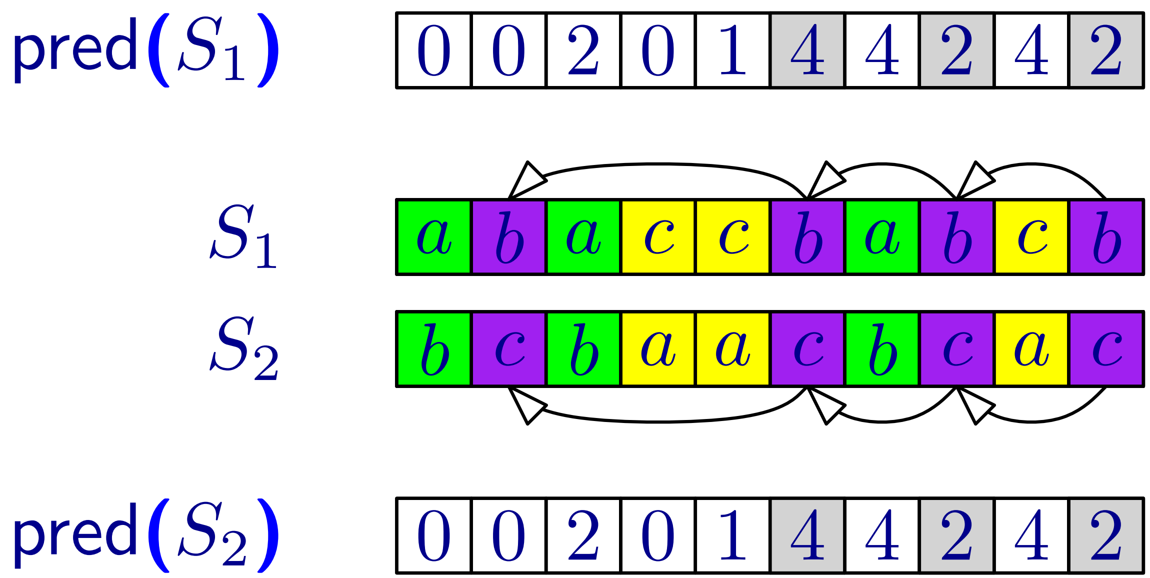
pred(S_2)

0	0	2	0	1	4	4	2	4	2

S_1 p-matches S_2 with mapping $a \to b$, $b \to c$, $c \to a$ (from S_1 to S_2)

pred(S_1)=pred(S_2)

Predecessor Strings



S_1 p-matches S_2 iff pred(S_1)=pred(S_2)

result due to Baker

Predecessor Strings

pred(T)	0	0	2	0	1	4	4	2	4	2
T	a	b	a	c	c	b	a	b	c	b
P						c	b	c	a	c

P p-matches T iff pred(P) $=$ pred$(T[i, i + |P| - 1])$

result due to Baker

Predecessor Strings

pred(T)	0	0	2	0	1	4	4	2	4	2
T	a	b	a	c	c	b	a	b	c	b
P						c	b	c	a	c
pred(P)						0	0	2	0	2

P p-matches T iff pred(P) = pred$(T[i, i + |P| - 1])$

result due to Baker

Predecessor Strings

pred(T)	0	0	2	0	1	4	4	2	4	2
T	a	b	a	c	c	b	a	b	c	b
P						c	b	c	a	c
pred(P)						0	0	2	0	2

P p-matches T iff pred(P) = pred$(T[i, i + |P| - 1])$

result due to Baker

however, pred$(T)[i, i + |P| - 1] \neq$ pred$(T[i, i + |P| - 1])$

Predecessor Strings

pred(T)

0	0	2	0	1	4	4	2	4	2

T

a	b	a	c	c	b	a	b	c	b

P

c	b	c	a	c

pred(P)

0	0	2	0	2

P p-matches T iff pred(P) = pred$(T[i, i+|P|-1])$

result due to Baker

however, pred$(T)[i, i+|P|-1] \neq$ pred$(T[i, i+|P|-1])$

some values may have to be *zeroed*

Rabin-Karp 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 $p = \Theta(|T|^4)$ is a prime and $1 \leq r < p$ is a random integer

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

Rabin-Karp fingerprints of strings

S | a | b | a | c | c | b | a | b | c | b

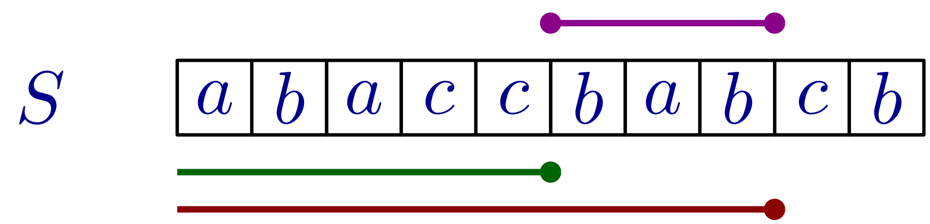
$$\phi(S) = \sum_{k=0}^{|S|-1} S[k]r^k \bmod p$$

Here $p = \Theta(|T|^4)$ 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 $O(\log |T|)$ bit word

Rabin-Karp fingerprints of strings



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

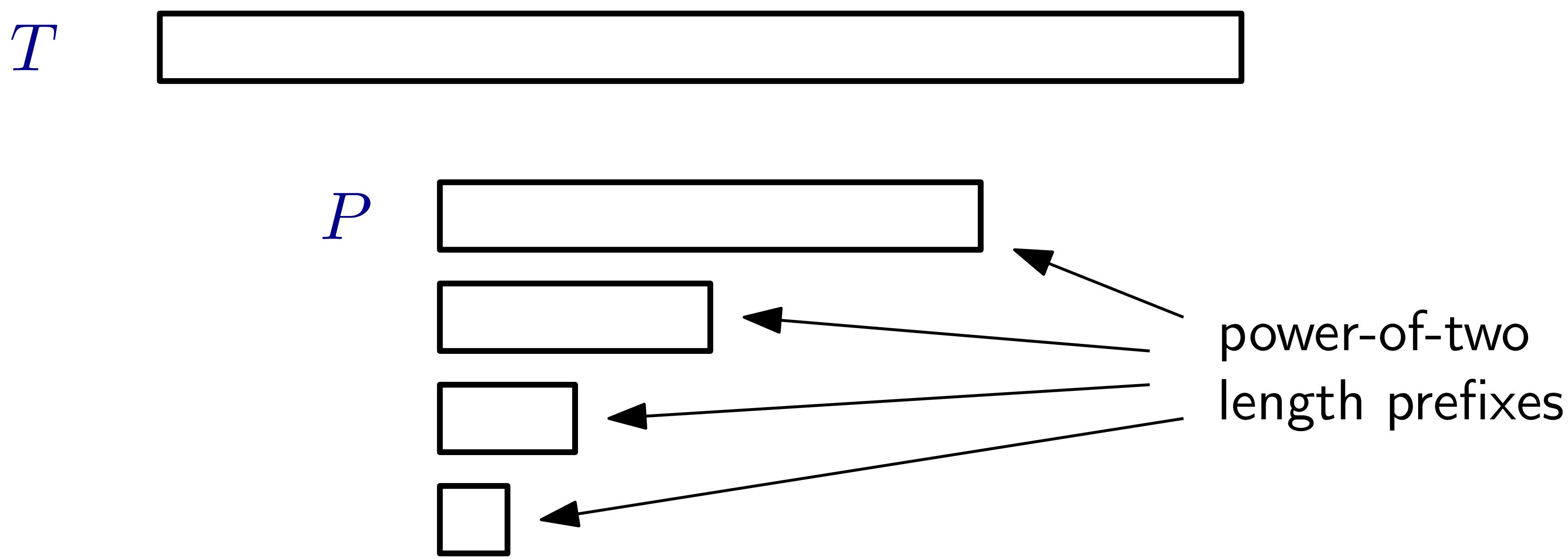
Here $p = \Theta(|T|^4)$ is a prime and $1 \leq r < p$ is a random integer

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

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

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

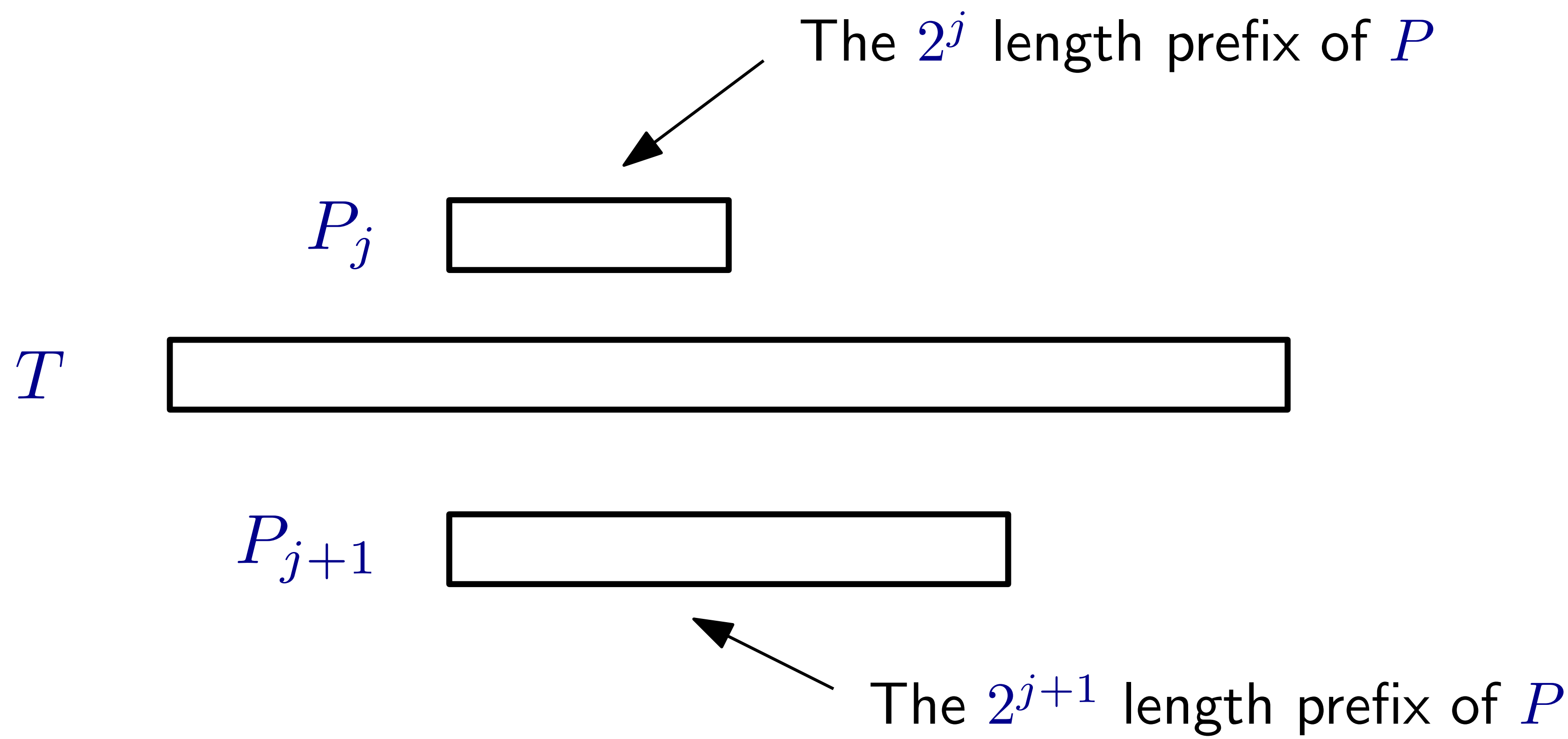
Exact matching using fingerprints (Porat and Porat)



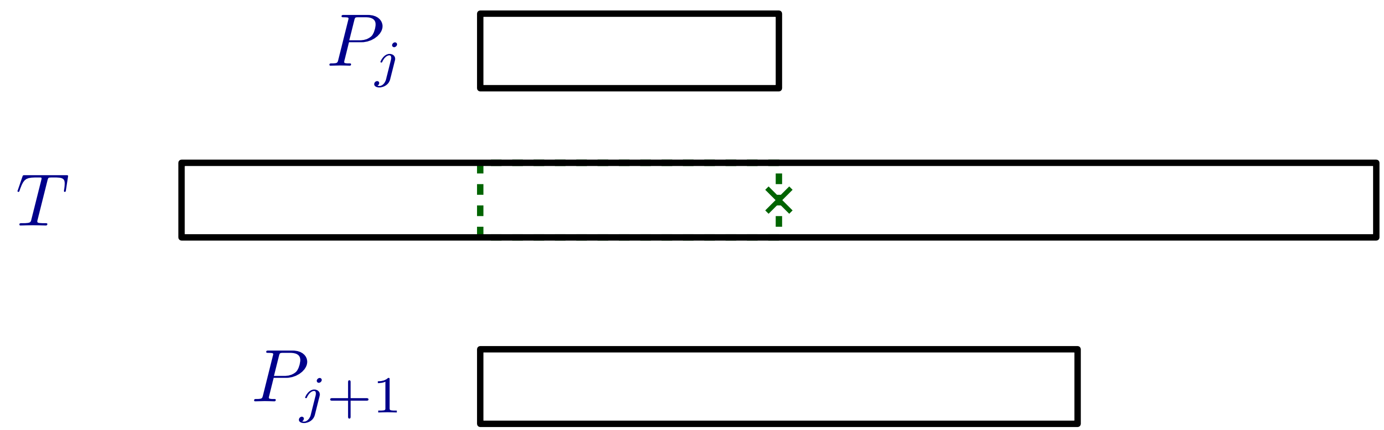
Find matches with each power-of-two length prefix

Exact matching using fingerprints (Porat and Porat)

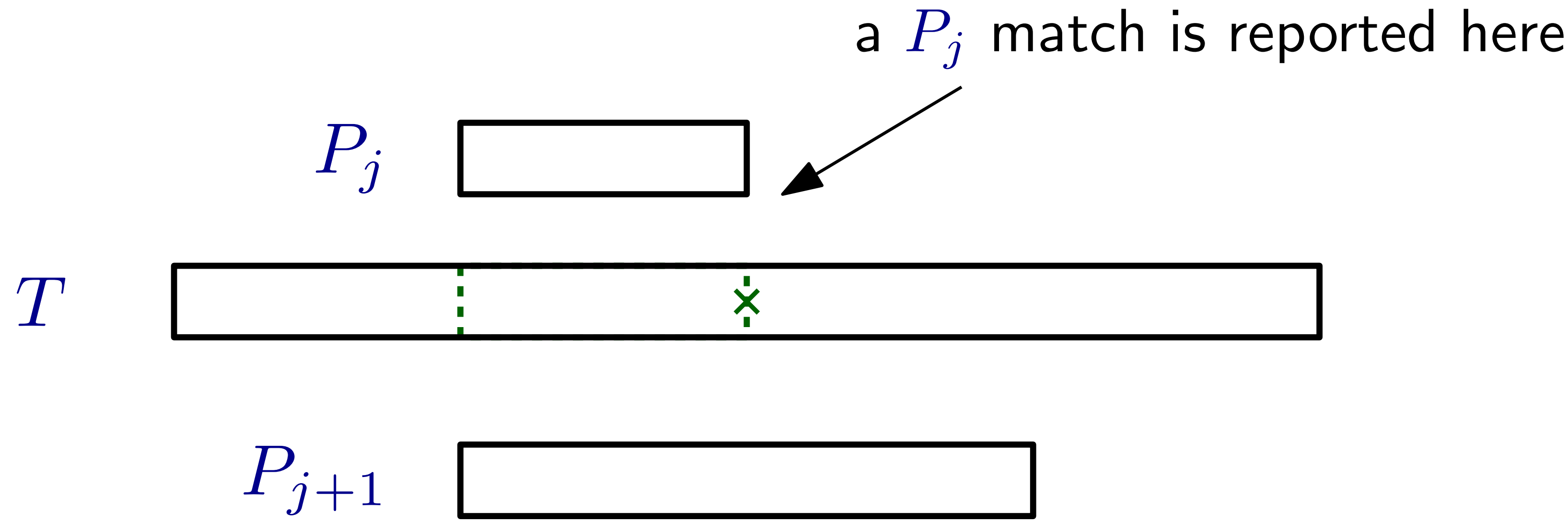
Exact matching using fingerprints (Porat and Porat)



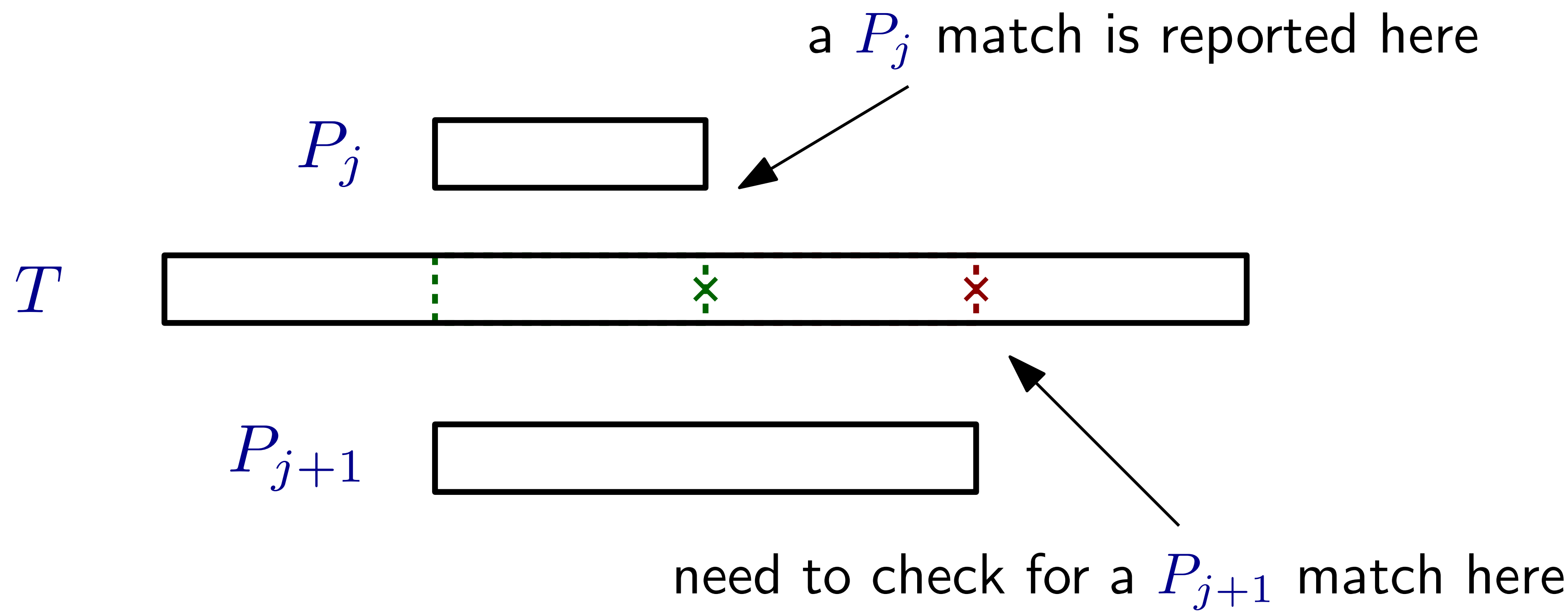
Exact matching using fingerprints (Porat and Porat)



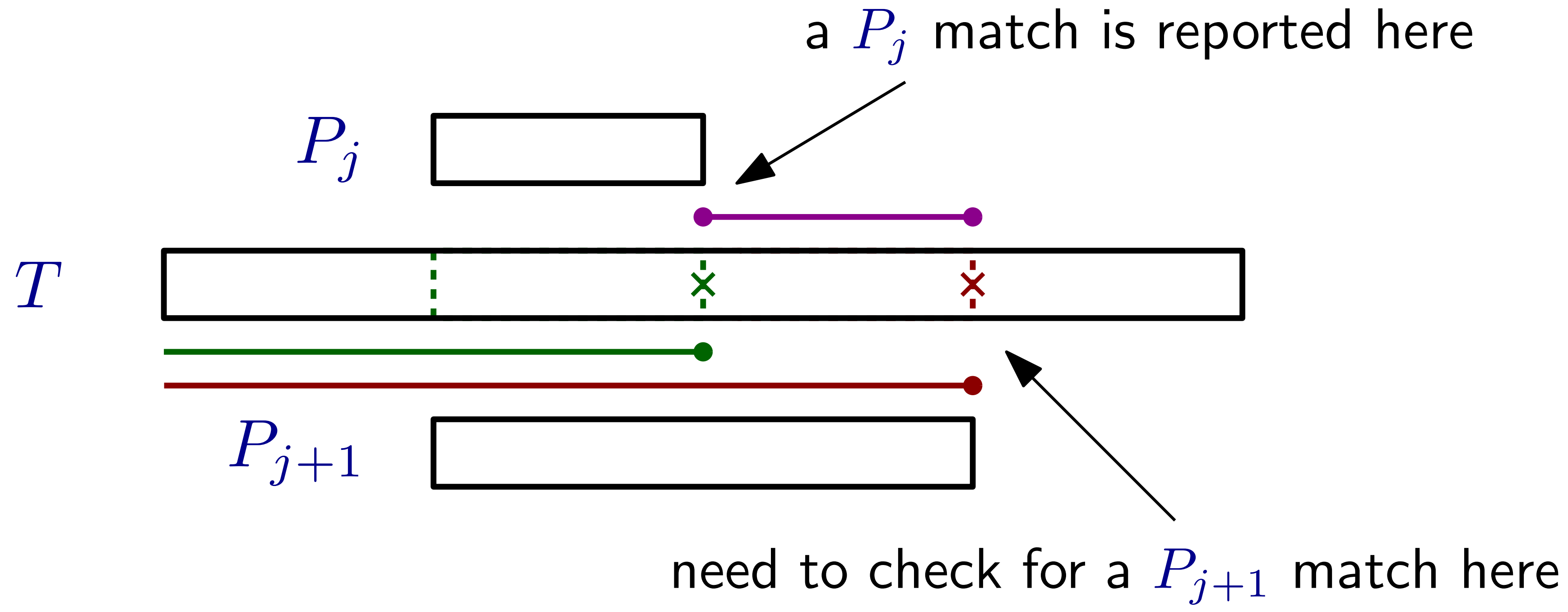
Exact matching using fingerprints (Porat and Porat)



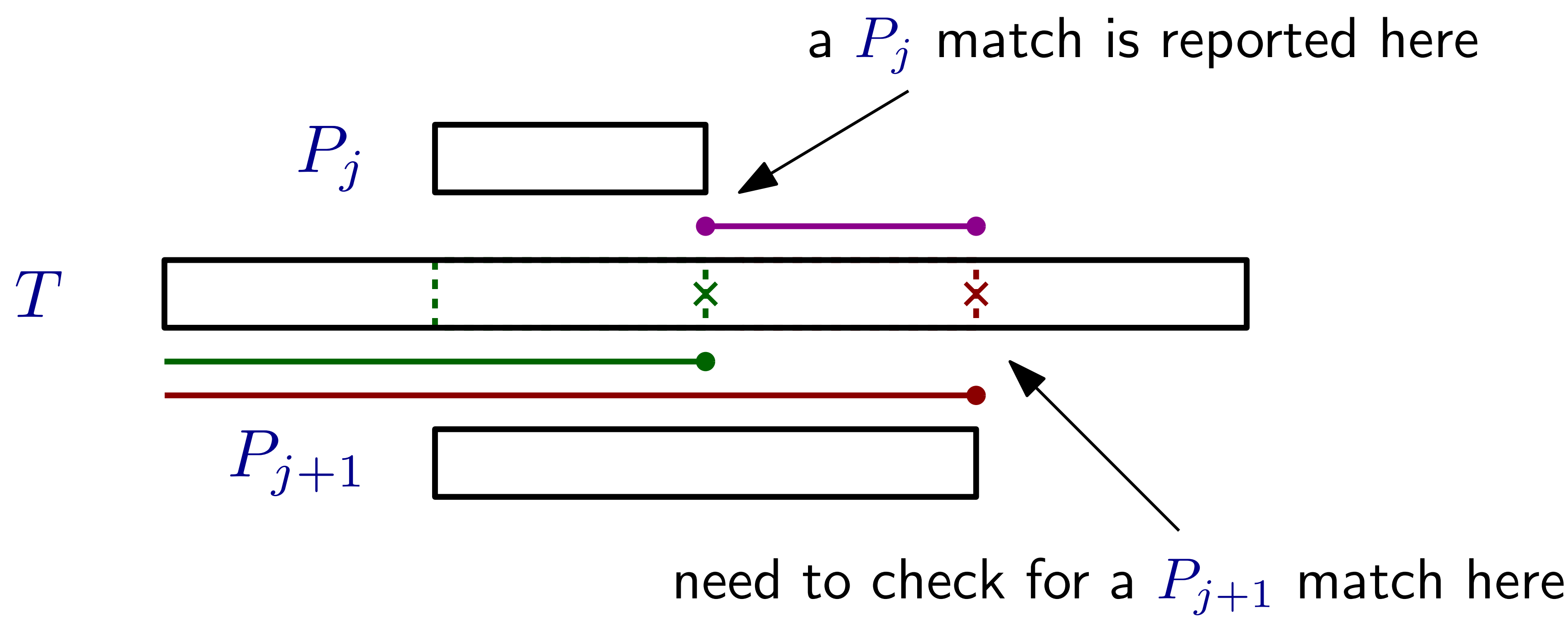
Exact matching using fingerprints (Porat and Porat)



Exact matching using fingerprints (Porat and Porat)

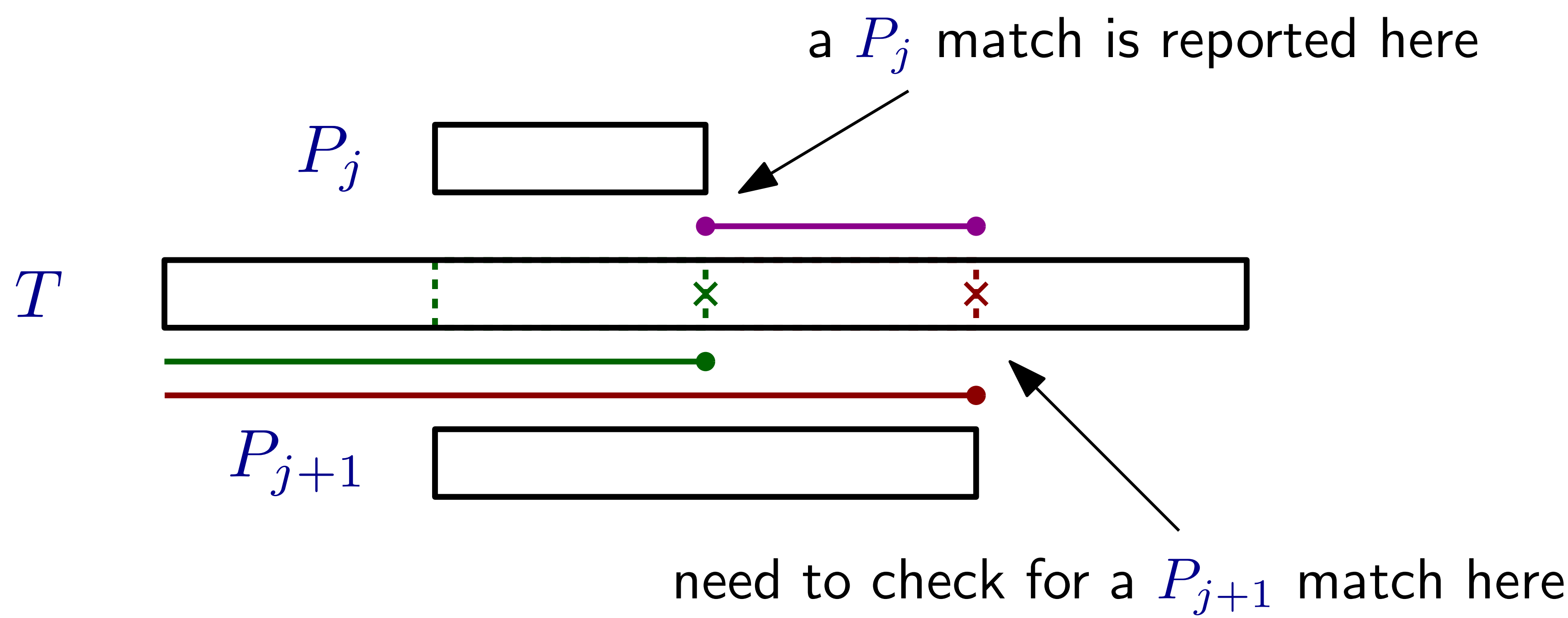


Exact matching using fingerprints (Porat and Porat)



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

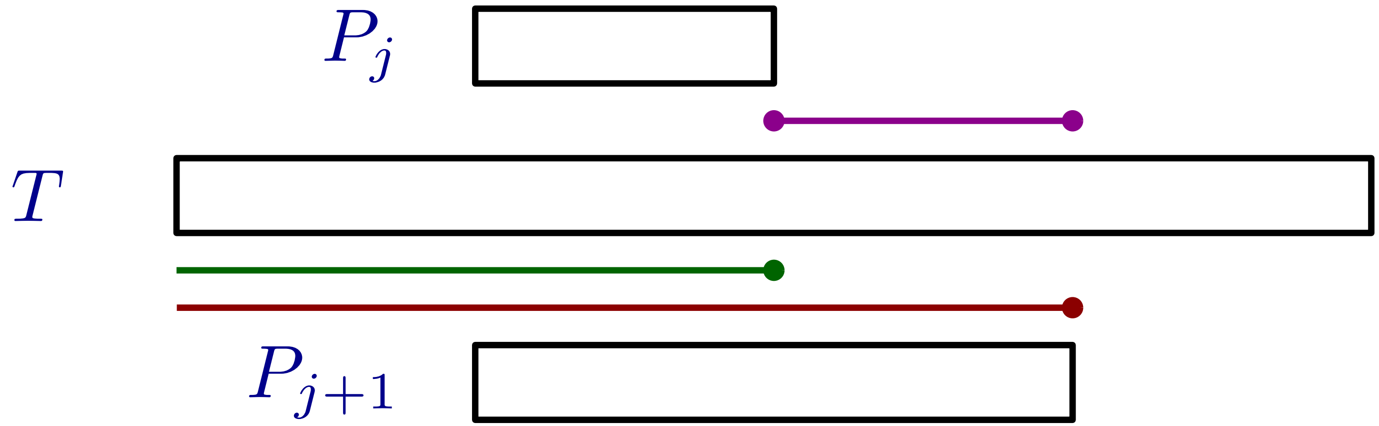
Exact matching using fingerprints (Porat and Porat)



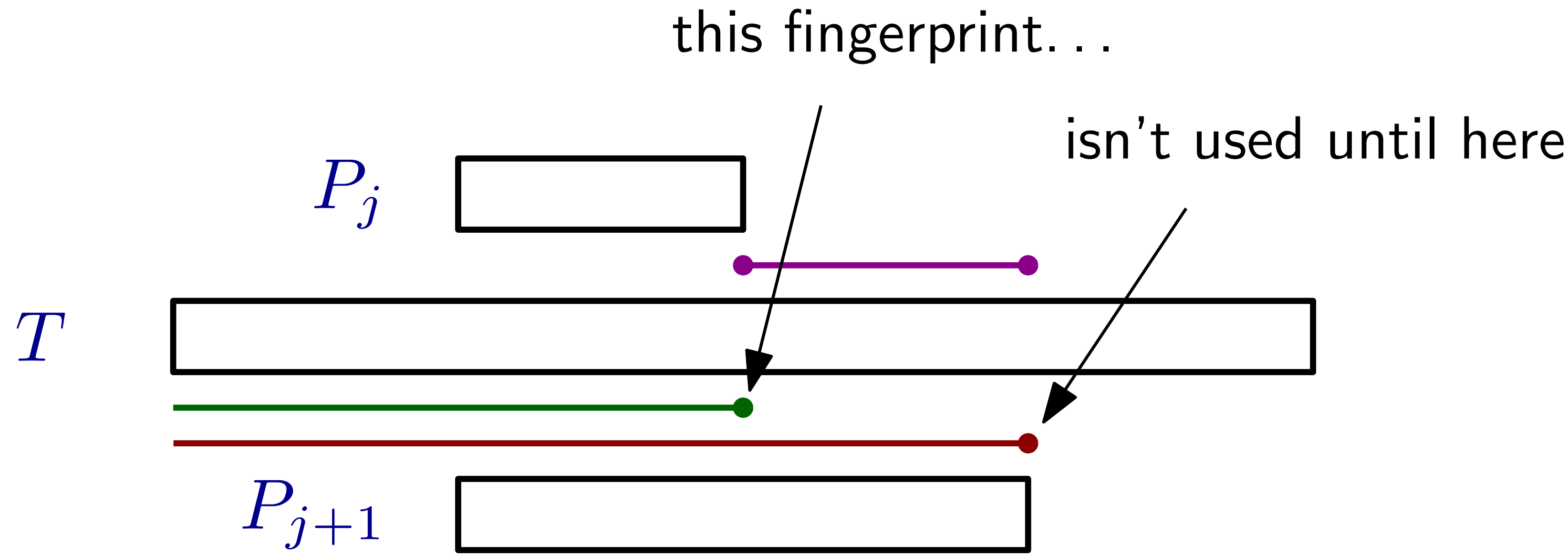
Given $\phi(T[0,\ell])$ and $\phi(T[0,r])$ we can compute $\phi(T[\ell+1,r])$ in $O(1)$ time

To decide whether P_{j+1} matches,
compare $\phi(T[\ell+1,r])$ to $\phi(P[2^j + \ldots 2^j - 1])$

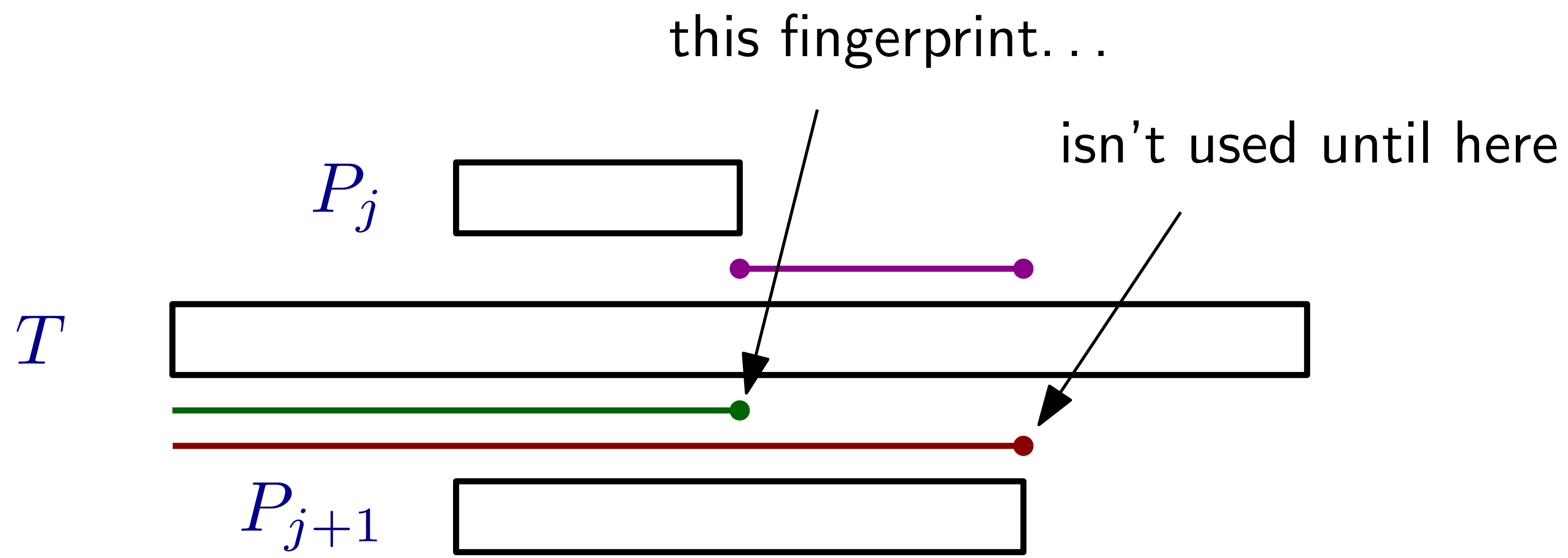
Exact matching using fingerprints (Porat and Porat)



Exact matching using fingerprints (Porat and Porat)

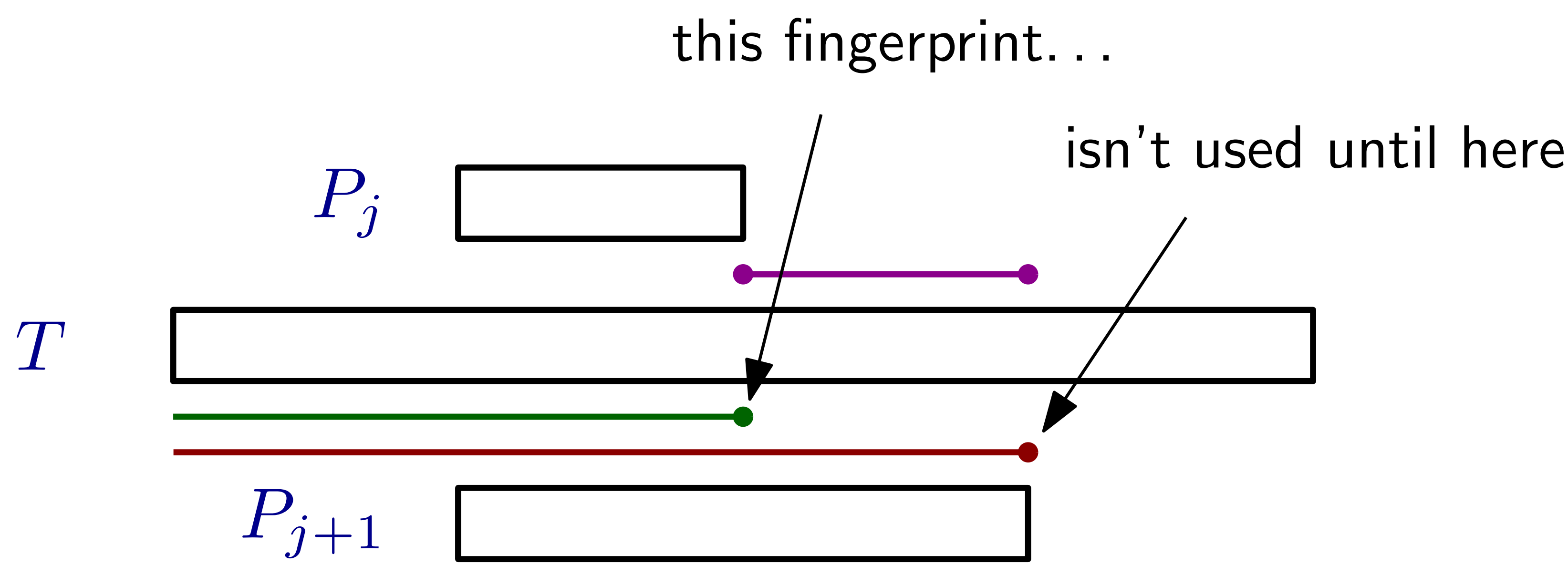


Exact matching using fingerprints (Porat and Porat)



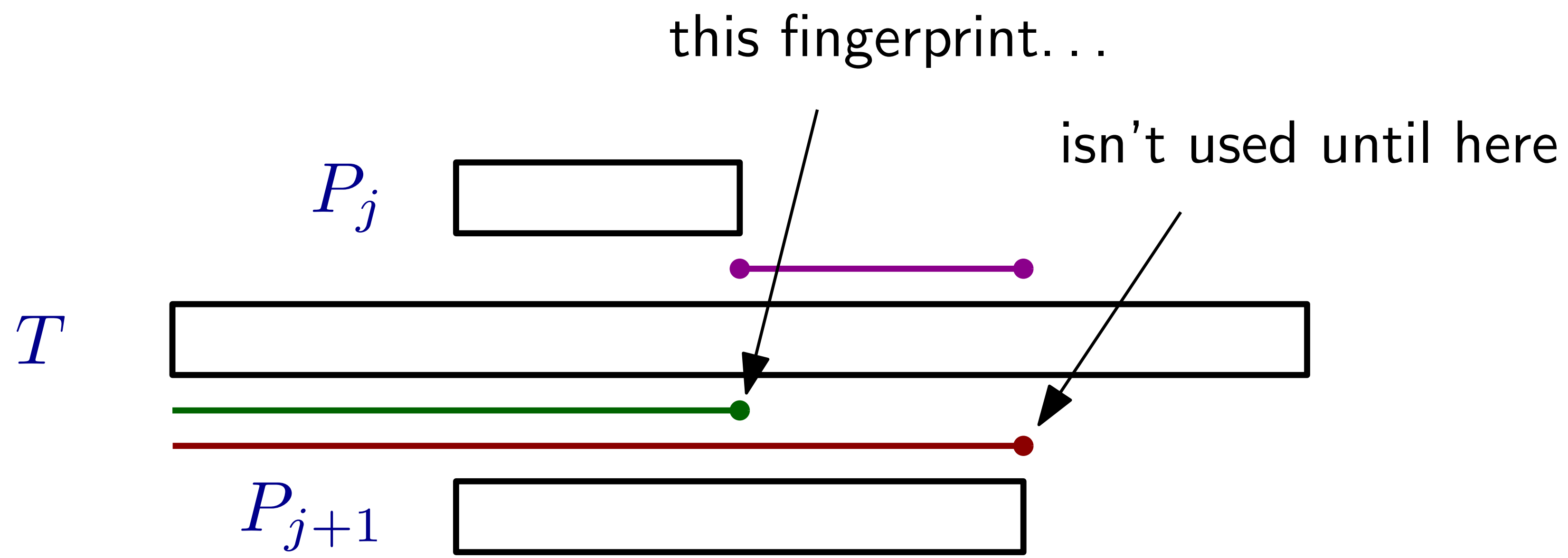
Each ϕ fits in a word but we need to store each one for a long time...

Exact matching using fingerprints (Porat and Porat)



Each ϕ fits in a word but we need to store each one for a long time...
we may have to store many ($\Omega(|P|)$) at a time.

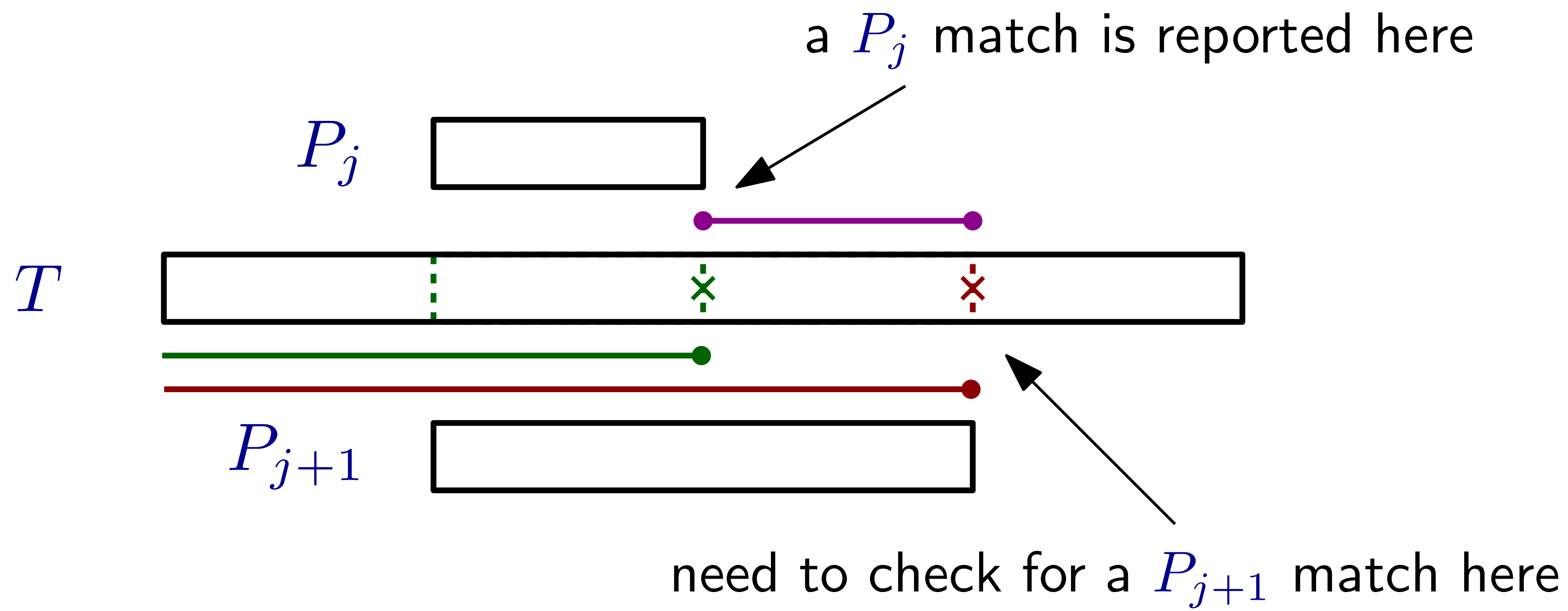
Exact matching using fingerprints (Porat and Porat)



Each ϕ fits in a word but we need to store each one for a long time...
we may have to store many ($\Omega(|P|)$) at a time.

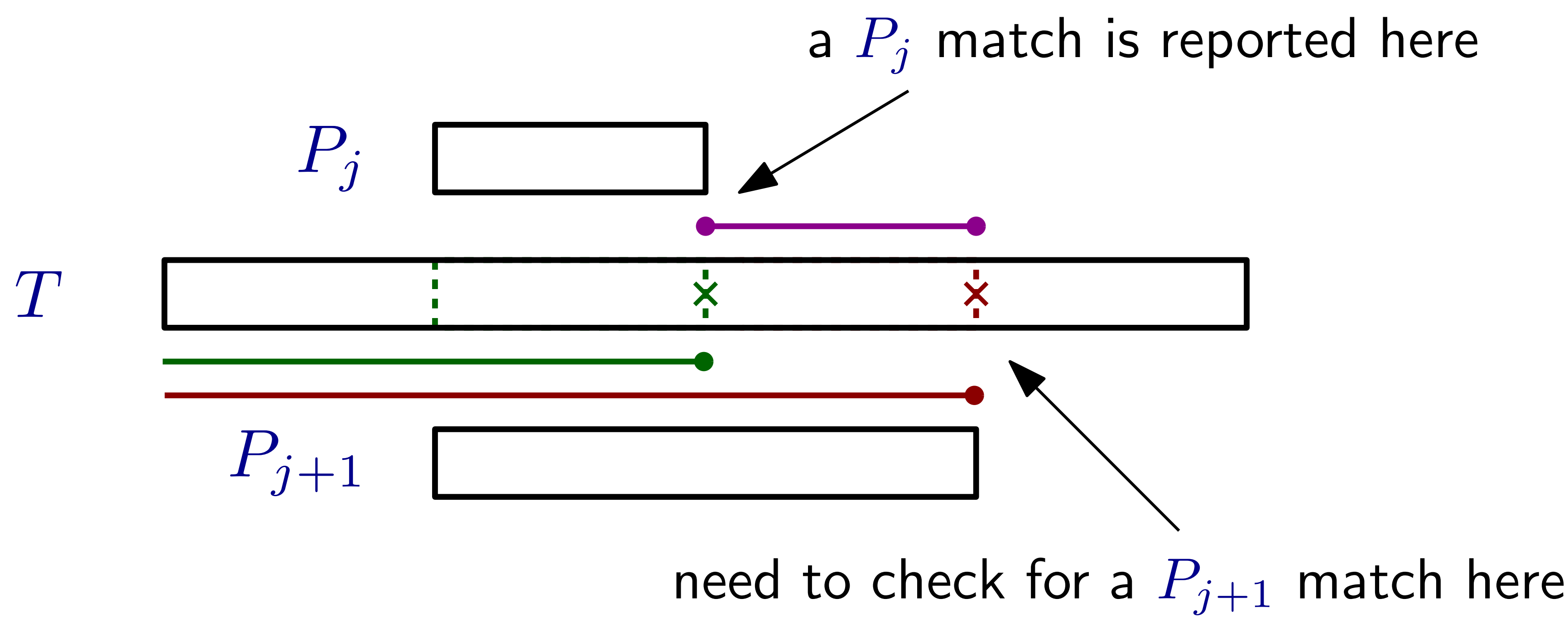
the fingerprints themselves have to be stored in a compressed form

Para. matching using fingerprints (this paper)



Overall approach: Find matches using fingerprints of predecessor strings

Para. matching using fingerprints (this paper)



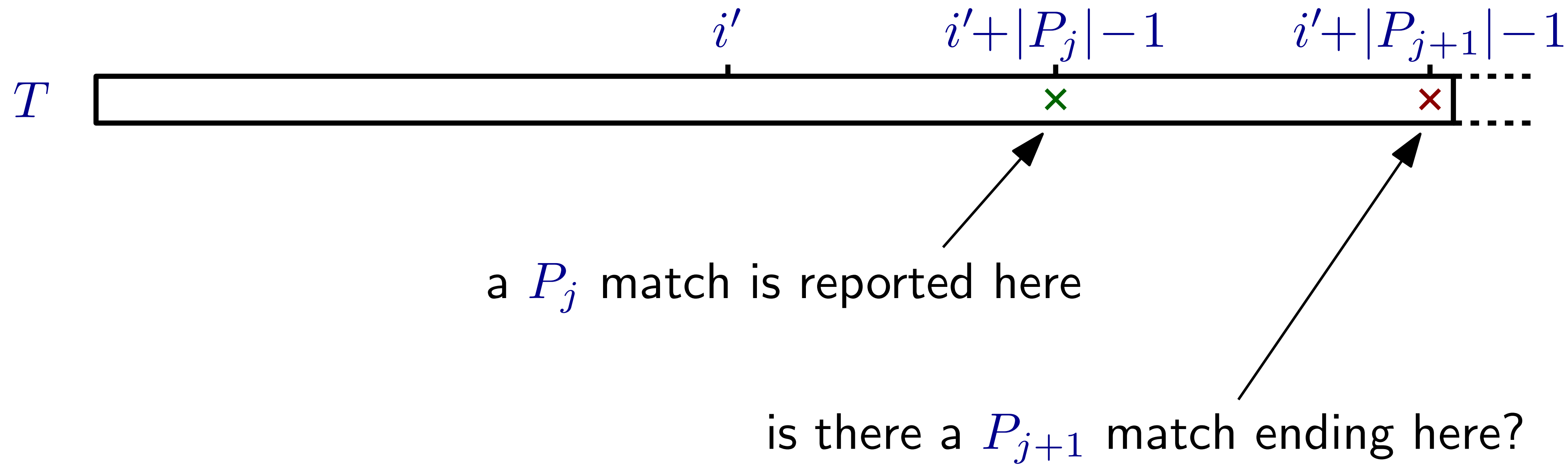
Overall approach: Find matches using fingerprints of predecessor strings

Key Problem 1: $\mathrm{pred}(T)[\ell+1, r] \neq \mathrm{pred}(T[\ell+1, r])$

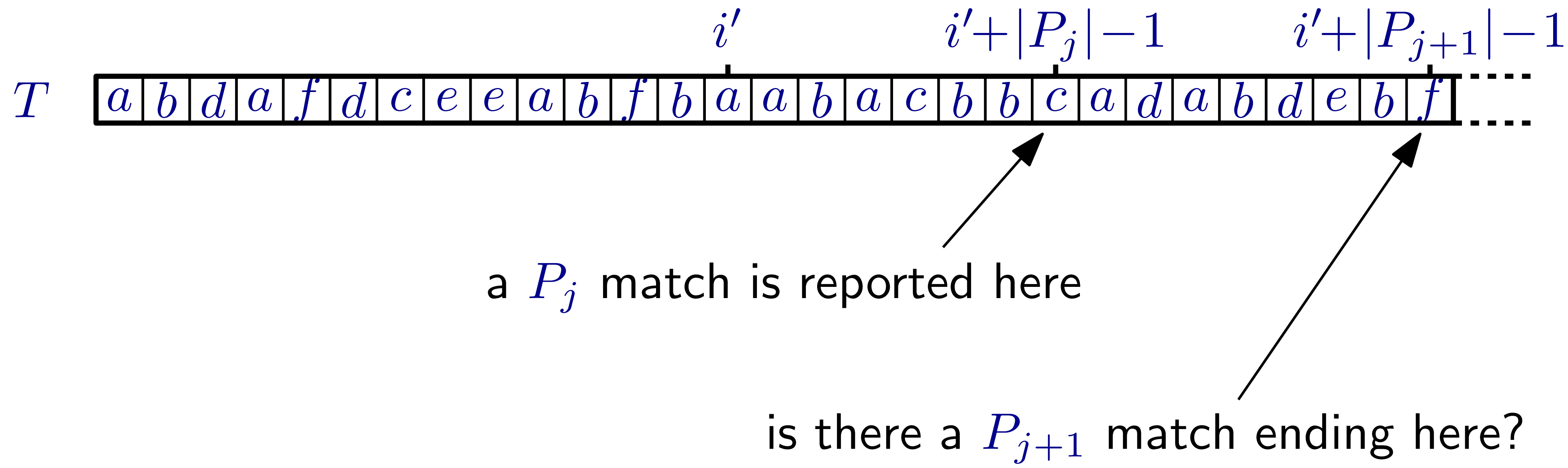
Key Problem 2: How do we store all the fingerprints?

Key Problem 3: How do we deamortise the algorithm?

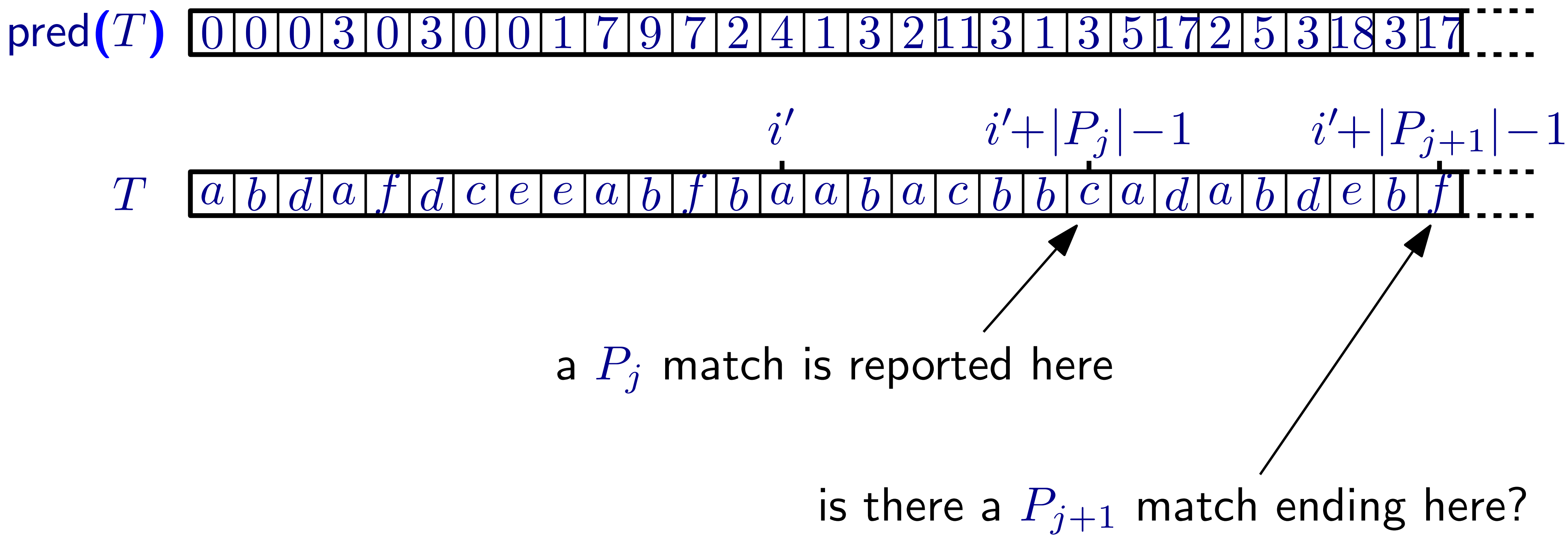
Key Problem 1: Correcting predecessor fingerprints



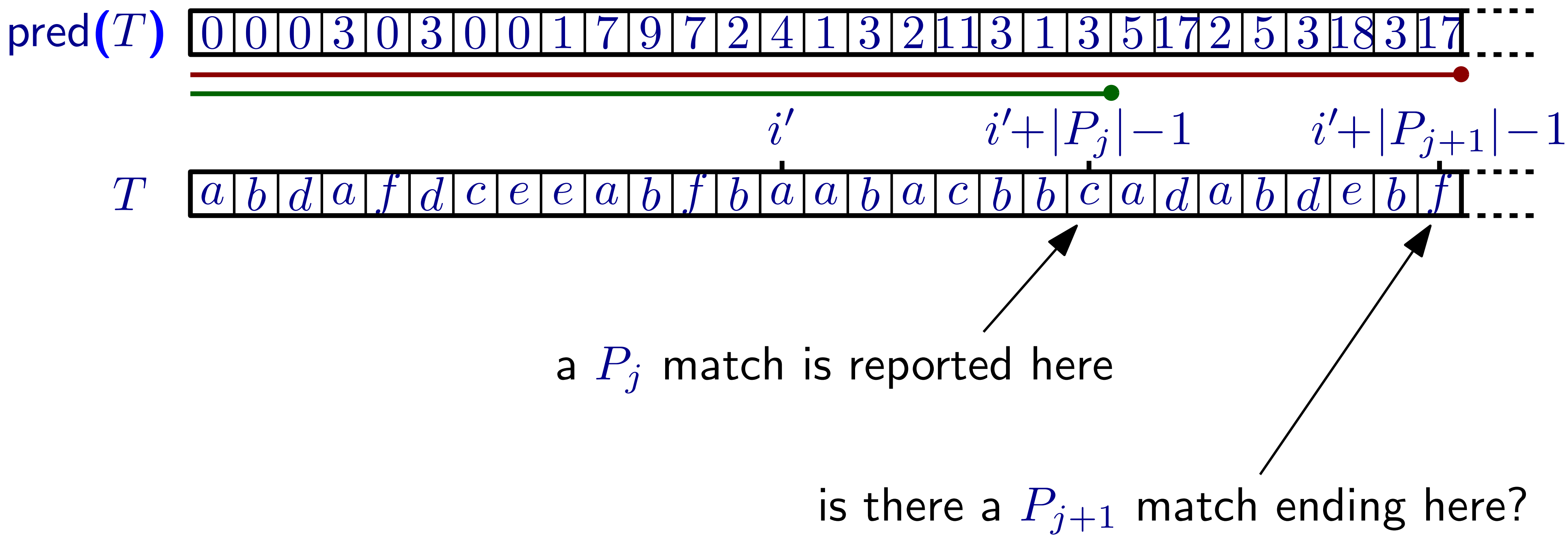
Key Problem 1: Correcting predecessor fingerprints



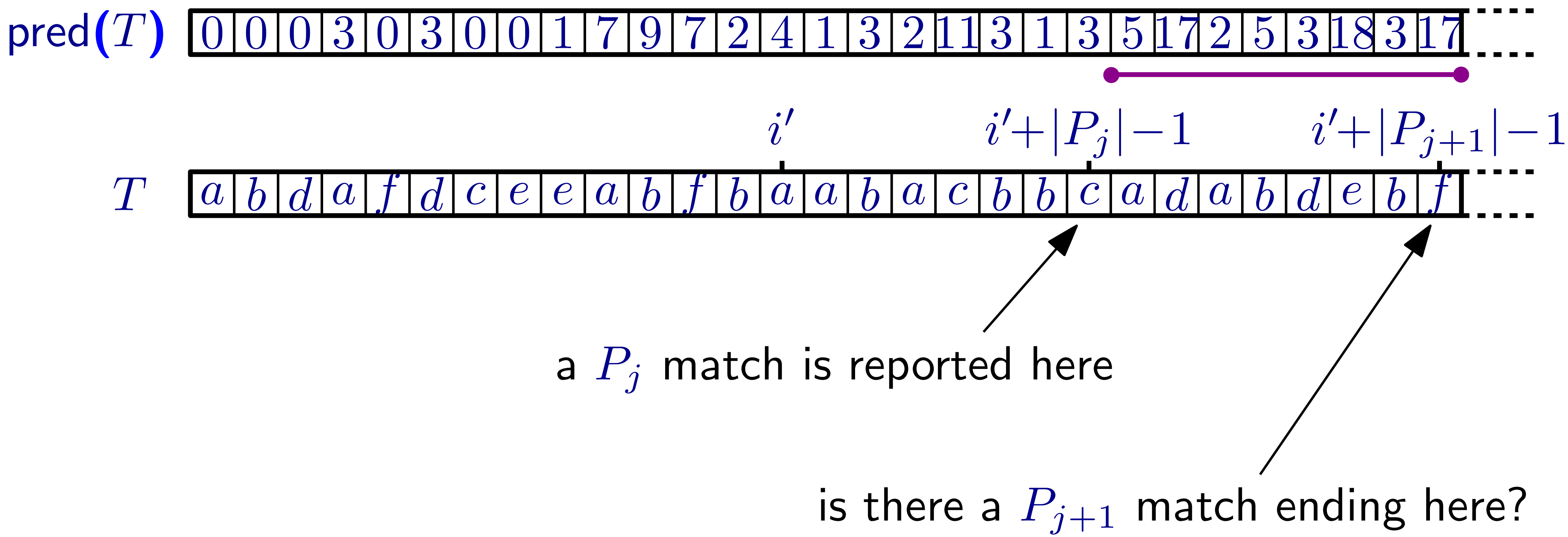
Key Problem 1: Correcting predecessor fingerprints



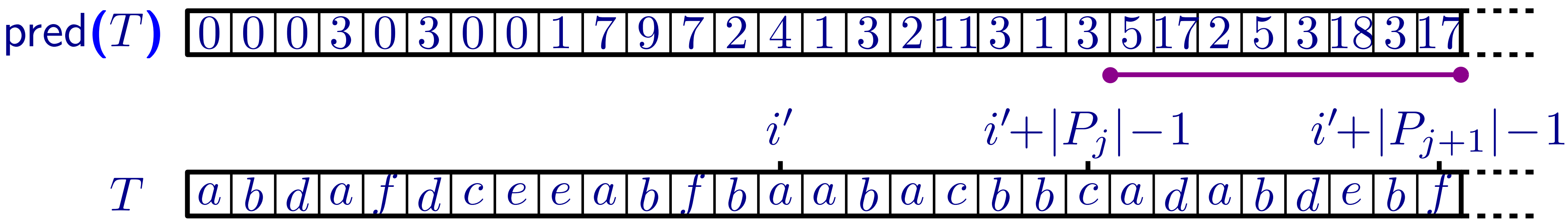
Key Problem 1: Correcting predecessor fingerprints



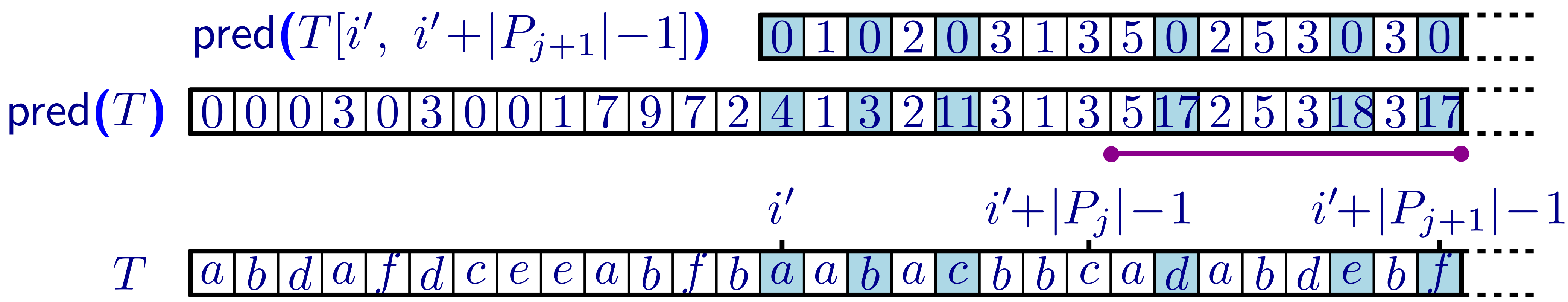
Key Problem 1: Correcting predecessor fingerprints



Key Problem 1: Correcting predecessor fingerprints



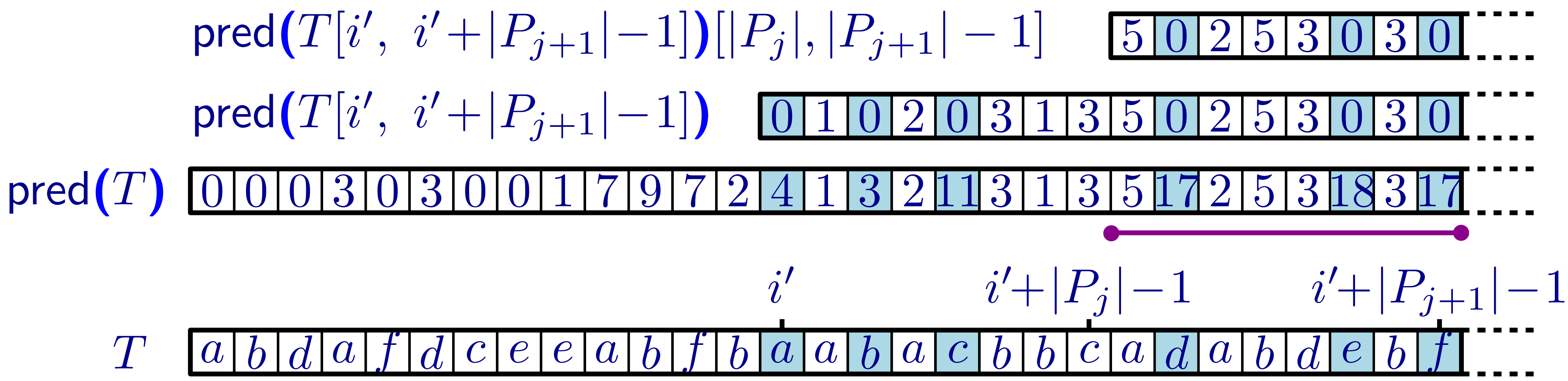
Key Problem 1: Correcting predecessor fingerprints



P_{j+1} p-matches iff

pred($T[i', i'+|P_{j+1}|-1]$) = pred(P_{j+1})

Key Problem 1: Correcting predecessor fingerprints



As P_j p-matches,

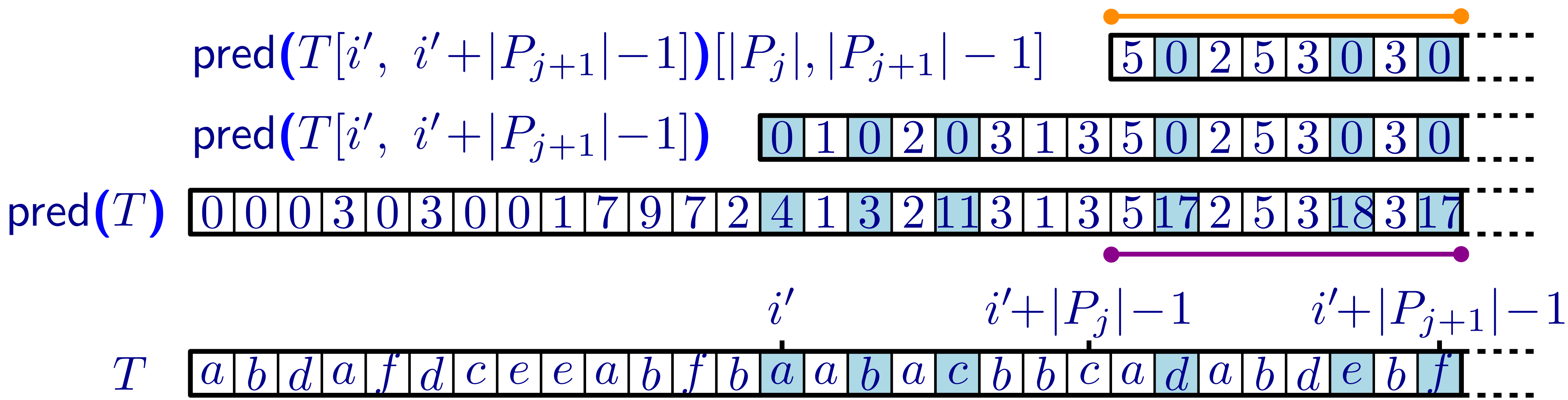
P_{j+1} p-matches iff

$$\text{pred}(T[i', i'+|P_{j+1}|-1])[|P_j|, |P_{j+1}|-1] = \text{pred}(P_{j+1})[|P_j|, |P_{j+1}|-1]$$

Compute this online

Precompute this

Key Problem 1: Correcting predecessor fingerprints



As P_j p-matches,

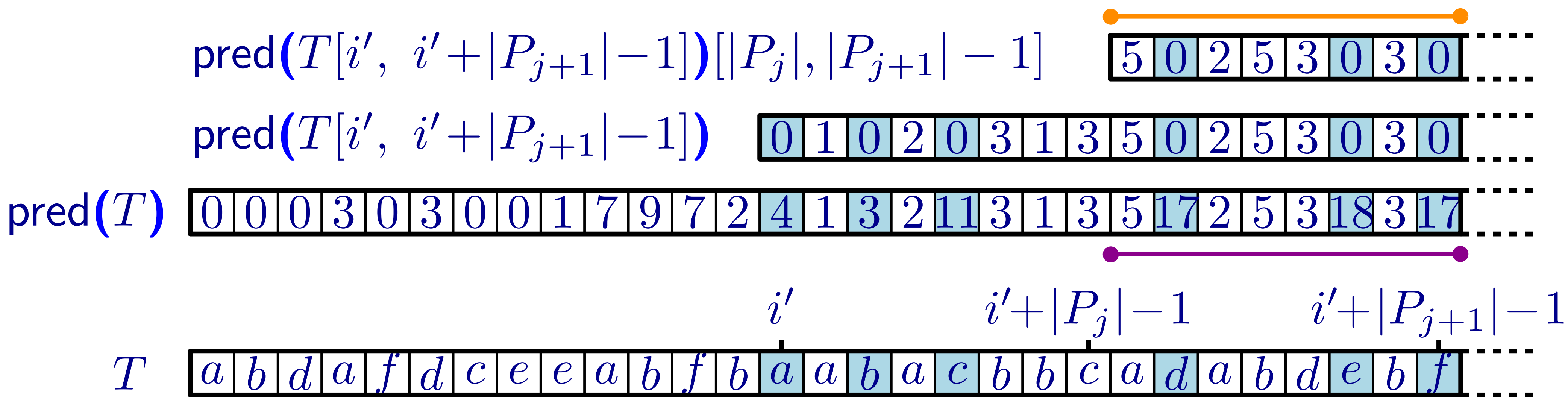
P_{j+1} p-matches iff

$$\text{pred}(T[i',\ i'+|P_{j+1}|-1])[|P_j|, |P_{j+1}|-1] = \text{pred}(P_{j+1})[|P_j|, |P_{j+1}|-1]$$

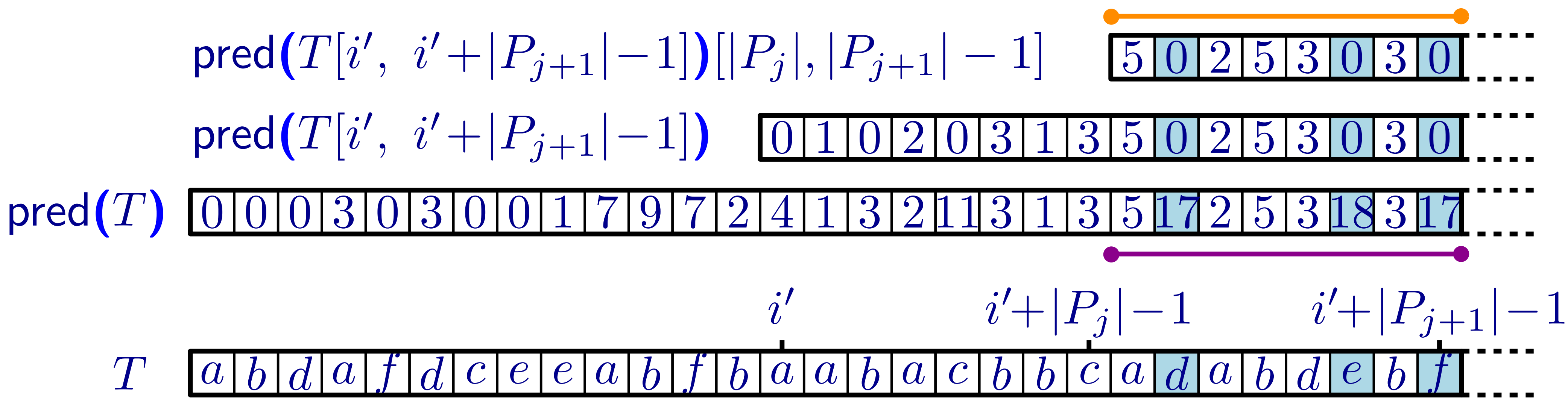
Compute this online

Precompute this

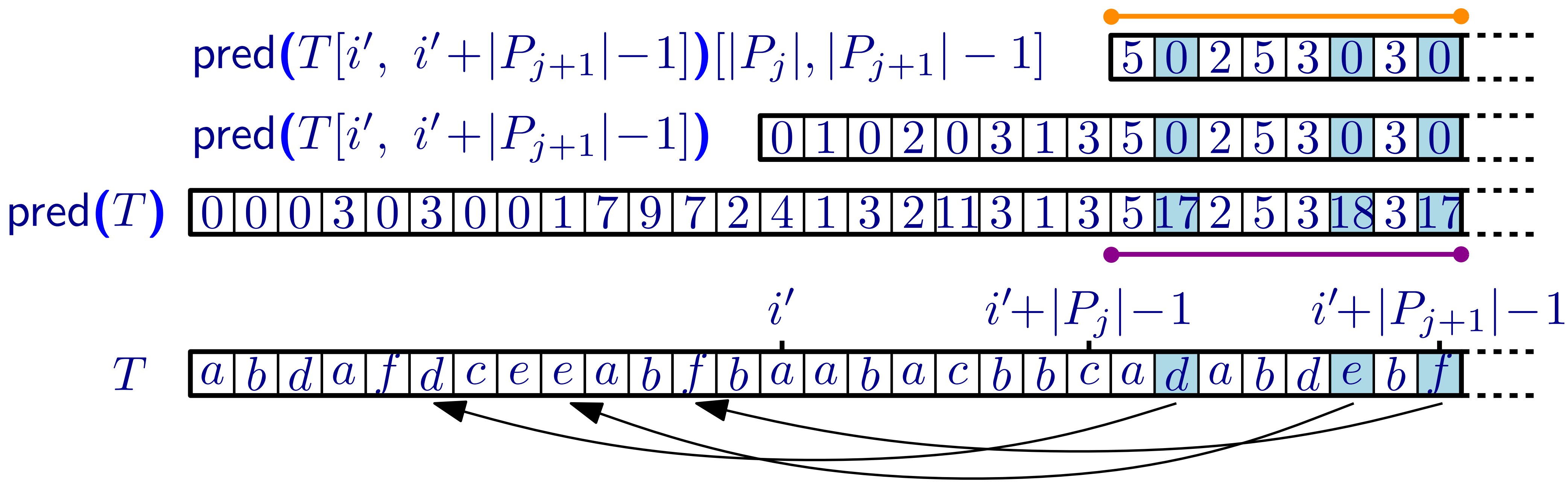
Key Problem 1: Correcting predecessor fingerprints



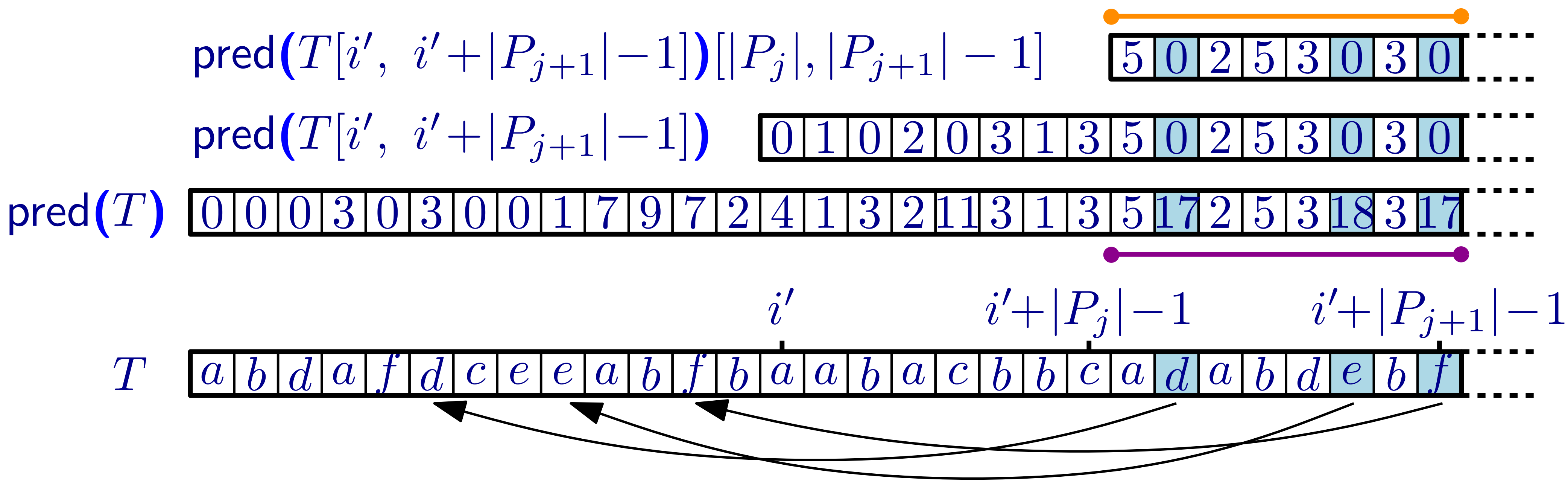
Key Problem 1: Correcting predecessor fingerprints



Key Problem 1: Correcting predecessor fingerprints

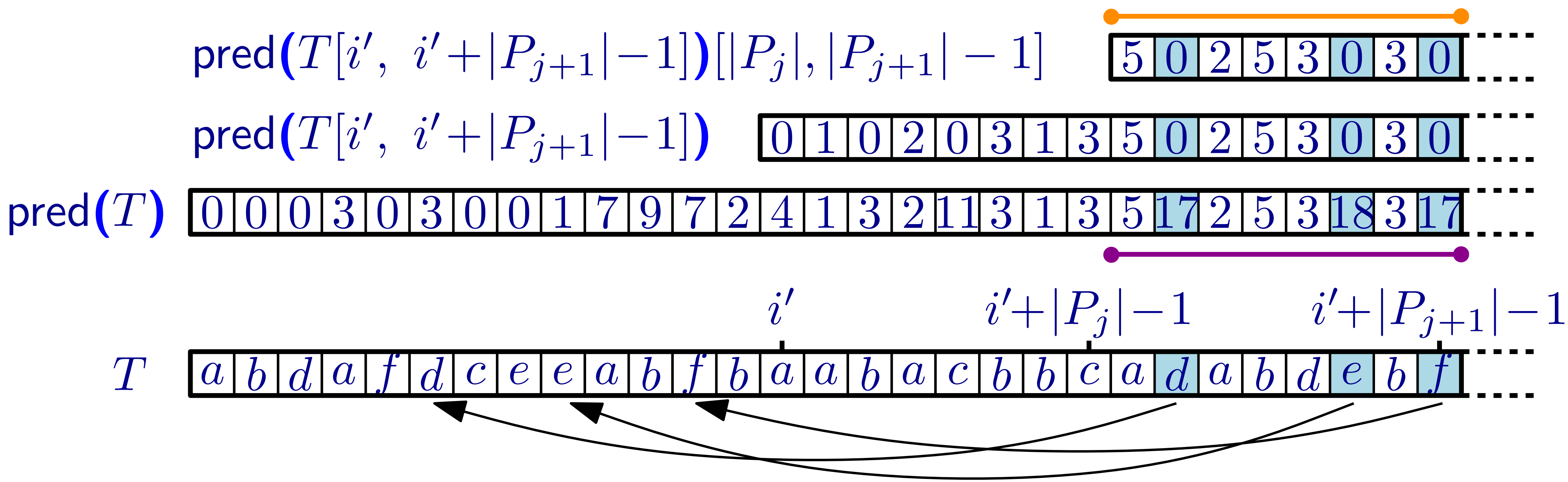


Key Problem 1: Correcting predecessor fingerprints



All these characters have large predecessor values. . . at least $|P_j|$

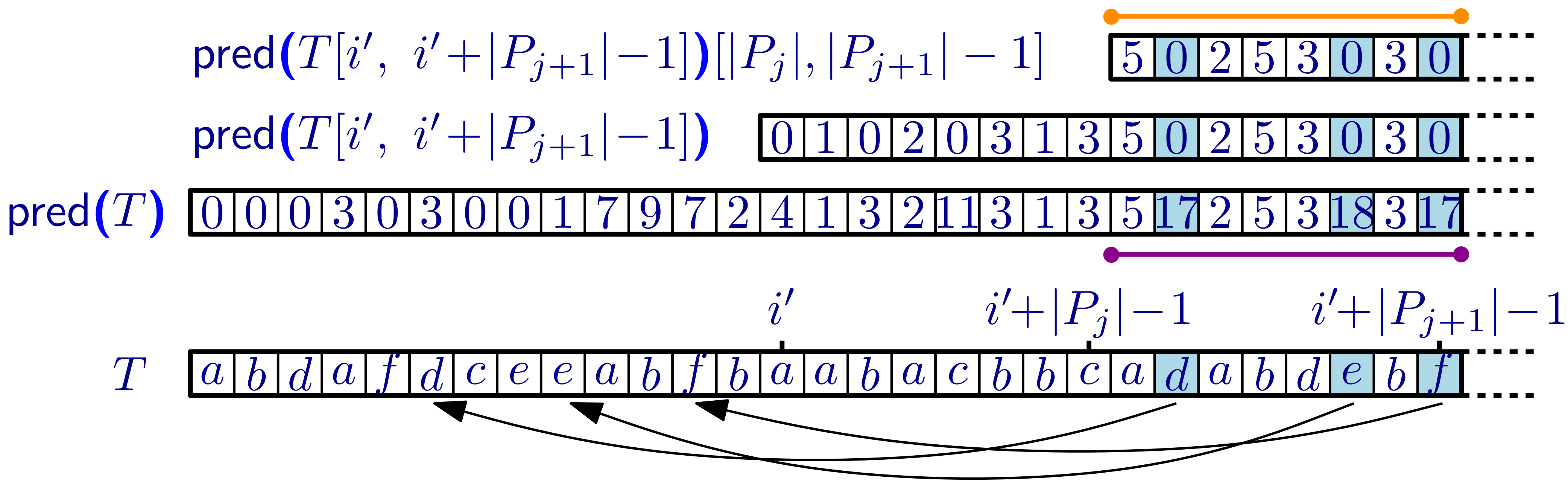
Key Problem 1: Correcting predecessor fingerprints



All these characters have large predecessor values. . . at least $|P_j|$

There are $O(|\Sigma|)$ such characters in an $O(|P_{j+1}|)$ length text window. . .
so we can store them all

Key Problem 1: Correcting predecessor fingerprints

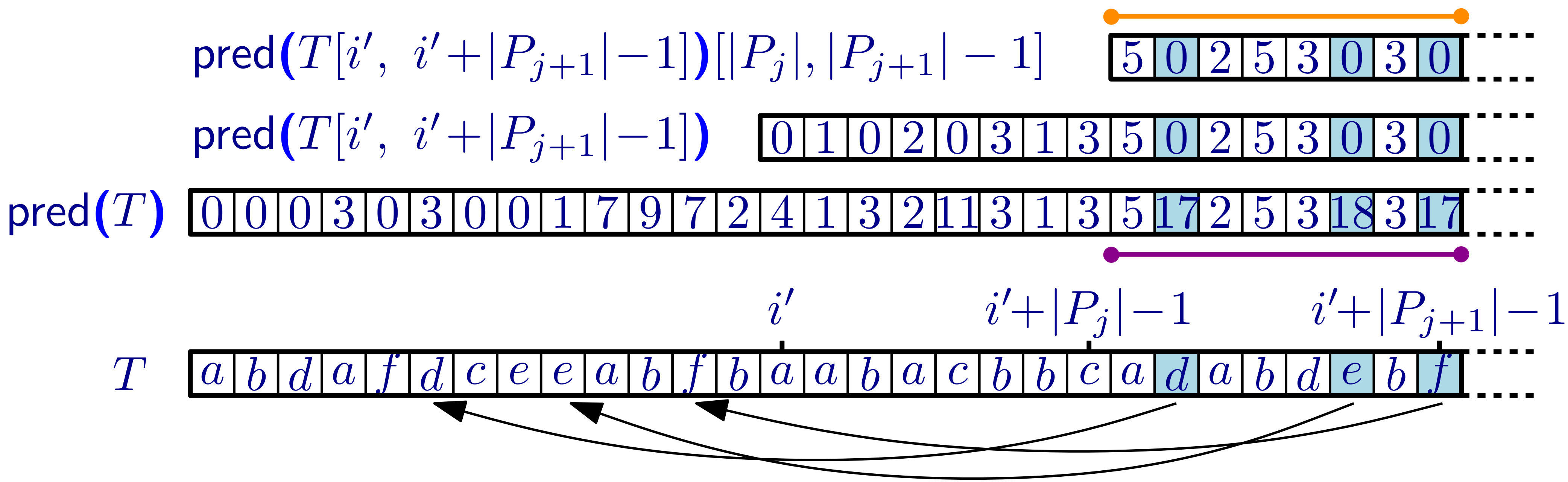


All these characters have large predecessor values. . . at least $|P_j|$

There are $O(|\Sigma|)$ such characters in an $O(|P_{j+1}|)$ length text window. . .
so we can store them all

Modifying the fingerprint in $O(|\Sigma|)$ time is simple arithmetic. . .

Key Problem 1: Correcting predecessor fingerprints

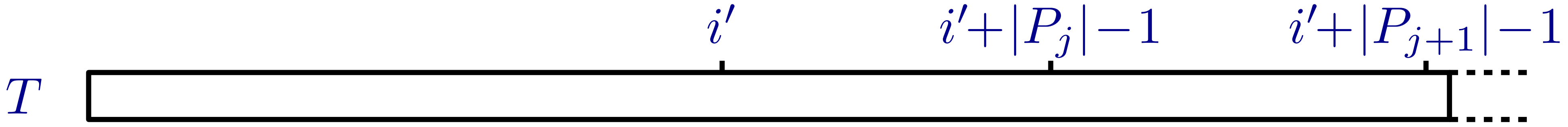


All these characters have large predecessor values... at least $|P_j|$

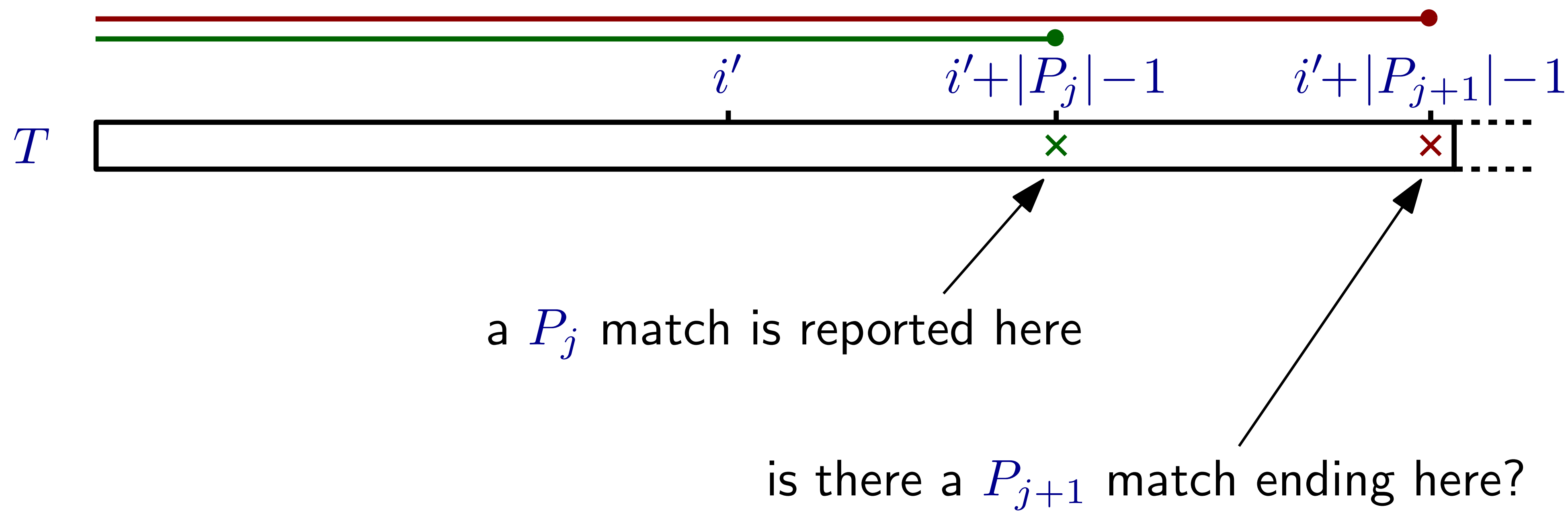
There are $O(|\Sigma|)$ such characters in an $O(|P_{j+1}|)$ length text window...
so we can store them all

Modifying the fingerprint in $O(|\Sigma|)$ time is simple arithmetic...
don't panic about the time complexity - we'll fix that later

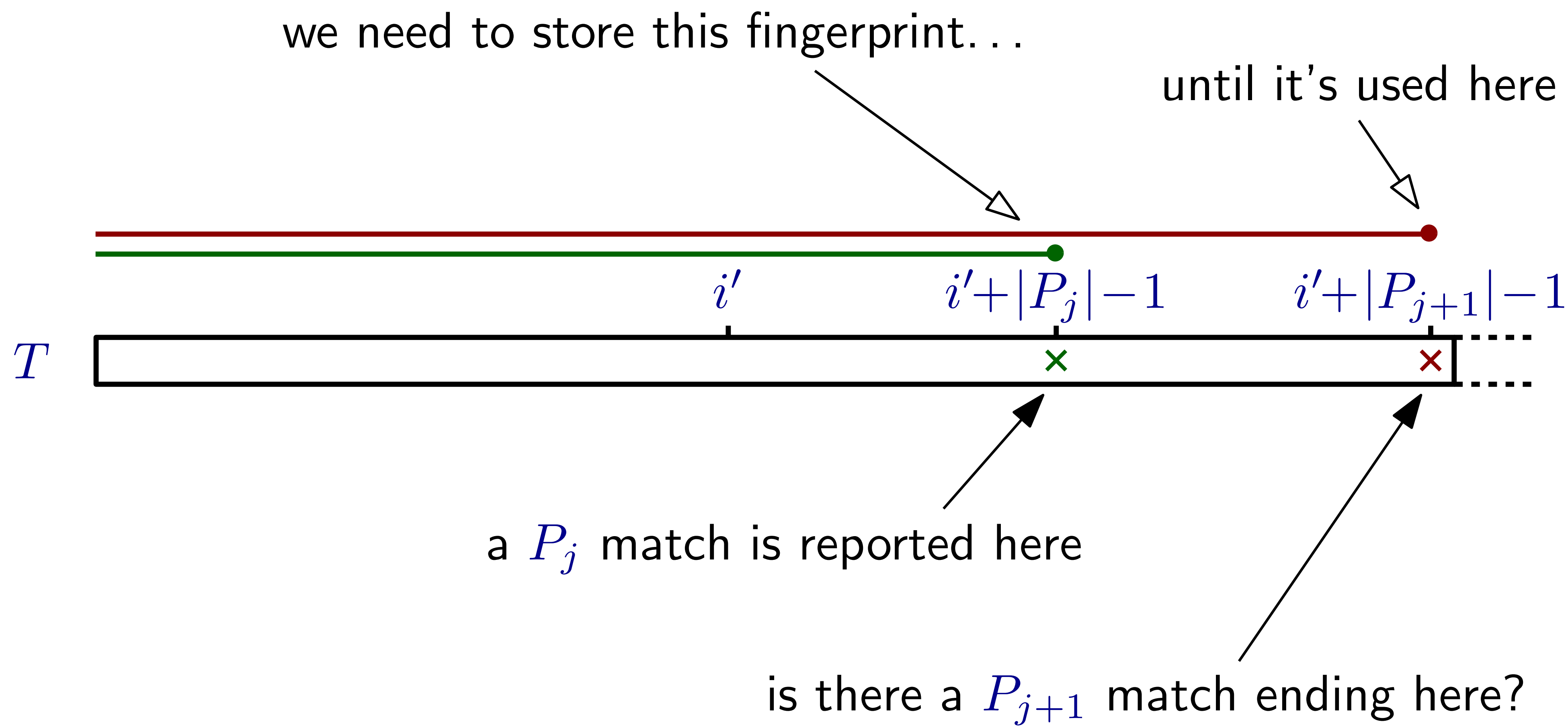
Key Problem 1: Correcting predecessor fingerprints



Key Problem 1: Correcting predecessor fingerprints



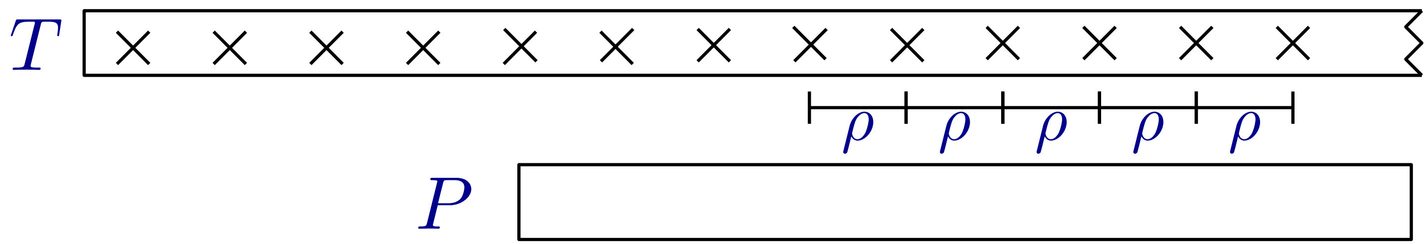
Key Problem 1: Correcting predecessor fingerprints



Key Problem 2: Storing the fingerprints

The structure of exact matches

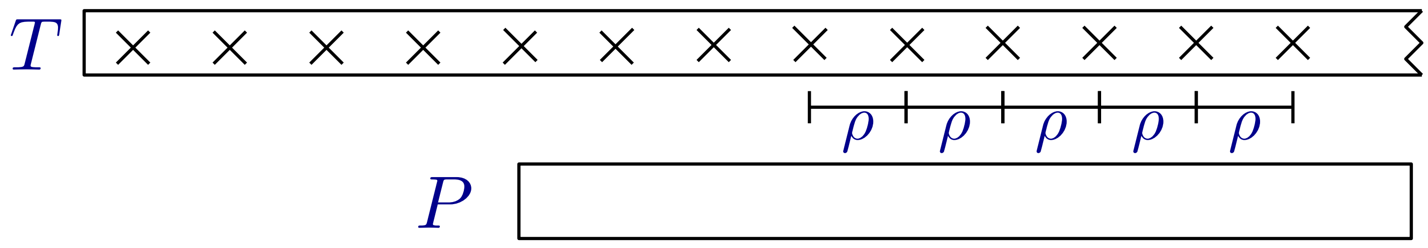
either every ρ or far apart



Key Problem 2: Storing the fingerprints

The structure of exact matches

either every ρ or far apart

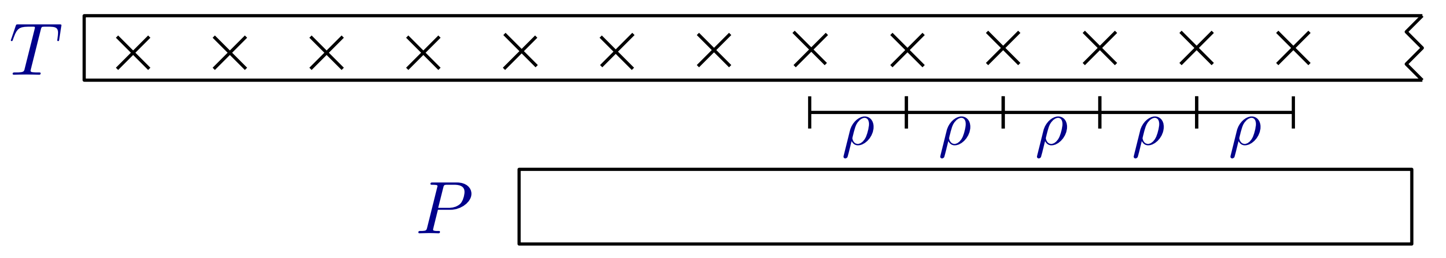


this allowed the partial **exact** *matches with each* P_j *to be encoded as an arithmetic progression in constant space*

Key Problem 2: Storing the fingerprints

The structure of exact matches

either every ρ or far apart

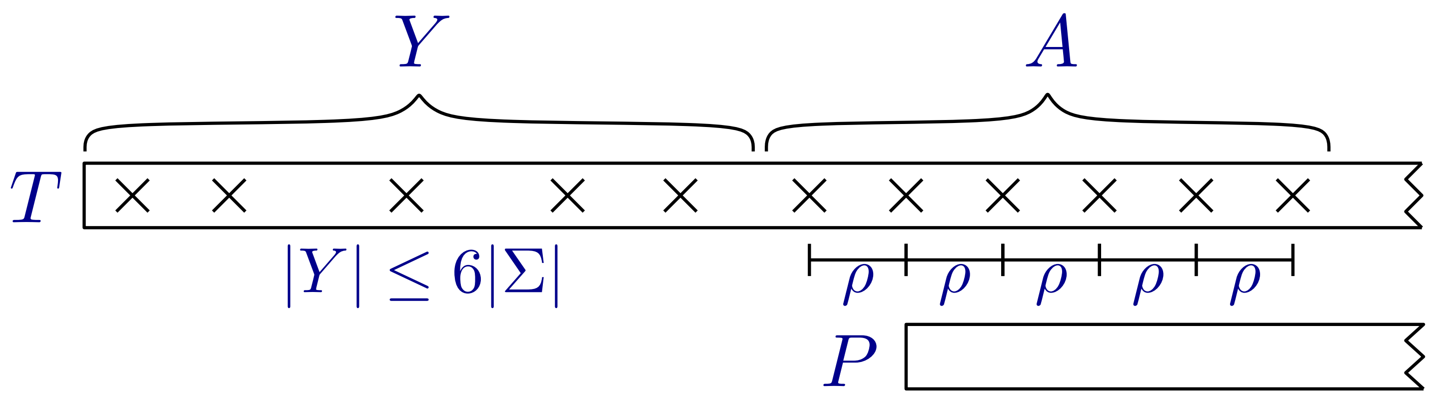


this allowed the partial **exact** *matches with each P_j to be encoded as an arithmetic progression in constant space*

the fingerprints can also be encoded in an analagous manner

Key Problem 2: Storing the fingerprints

The structure of parameterized matches



this allows the partial parameterized matches with each P_j to be encoded in $O(|\Sigma|)$ space

the fingerprints can also be encoded in an analagous manner

Key Problem 3: Deamortising the algorithm

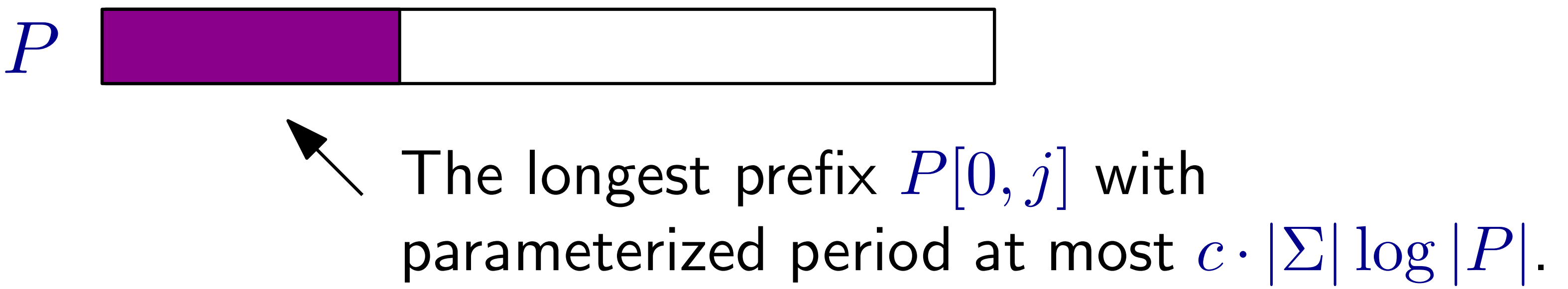
As described, the algorithm takes $O(|\Sigma|)$ time per prefix match found.

Using an idea of Breslauer and Galil, this is reduced to $O(1)$ per char.

Key Problem 3: Deamortising the algorithm

As described, the algorithm takes $O(|\Sigma|)$ time per prefix match found.

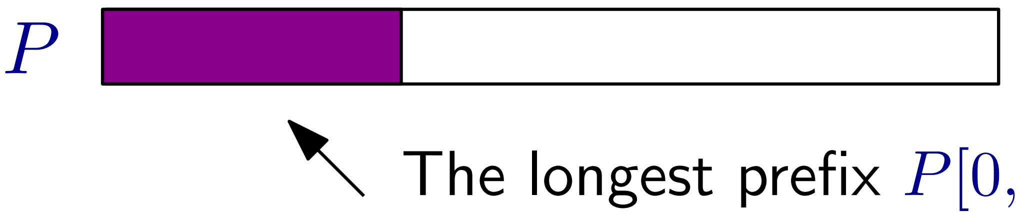
Using an idea of Breslauer and Galil, this is reduced to $O(1)$ per char.



Key Problem 3: Deamortising the algorithm

As described, the algorithm takes $O(|\Sigma|)$ time per prefix match found.

Using an idea of Breslauer and Galil, this is reduced to $O(1)$ per char.



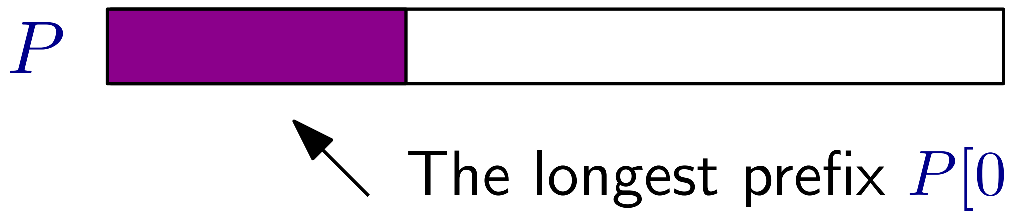
The longest prefix $P[0, j]$ with parameterized period at most $c \cdot |\Sigma| \log |P|$.

Matches with $P[0, j+1]$ are $\Omega(|\Sigma| \log |P|)$ alignments apart.

Key Problem 3: Deamortising the algorithm

As described, the algorithm takes $O(|\Sigma|)$ time per prefix match found.

Using an idea of Breslauer and Galil, this is reduced to $O(1)$ per char.



The longest prefix $P[0, j]$ with parameterized period at most $c \cdot |\Sigma| \log |P|$.

Matches with $P[0, j+1]$ are $\Omega(|\Sigma| \log |P|)$ alignments apart.

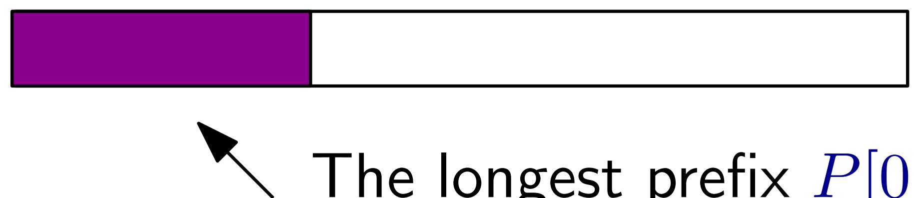
We also give a deterministic algorithm which outputs all $P[0, j]$ matches in $O(|\Sigma| \log |P|)$ space and $O(1)$ time per character.

Key Problem 3: Deamortising the algorithm

As described, the algorithm takes $O(|\Sigma|)$ time per prefix match found.

Using an idea of Breslauer and Galil, this is reduced to $O(1)$ per char.

P



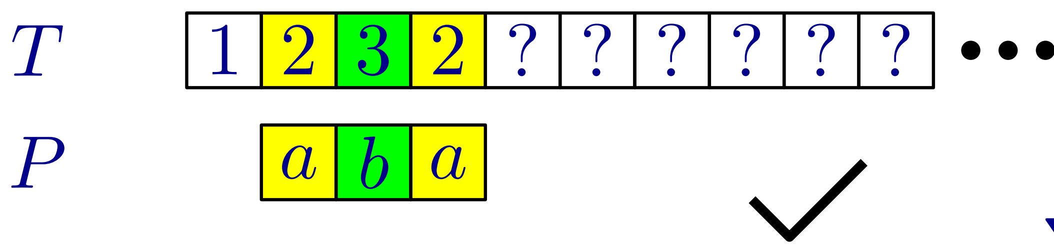
The longest prefix $P[0, j]$ with parameterized period at most $c \cdot |\Sigma| \log |P|$.

Matches with $P[0, j+1]$ are $\Omega(|\Sigma| \log |P|)$ alignments apart.

We also give a deterministic algorithm which outputs all $P[0, j]$ matches in $O(|\Sigma| \log |P|)$ space and $O(1)$ time per character.

(in fact it works for any pattern with small parameterized period)

Conclusions



Σ is the alphabet

Our Main Results

Parameterized matching can be solved in $O(|\Sigma| \log |P|)$ space and:

- $O(1)$ time per character when $|\Sigma| = \{1, 2, 3, \ldots, |\Sigma|\}$
- $O(\sqrt{\log |\Sigma| / \log \log |\Sigma|})$ time per character for general Σ

Both algorithms are randomised (Monte-Carlo)

Thank you for listening