String Range Matching

Juha Kärkkäinen Dominik Kempa Simon J. Puglisi

University of Helsinki, Finland

CPM 2014
Moscow, June 2014

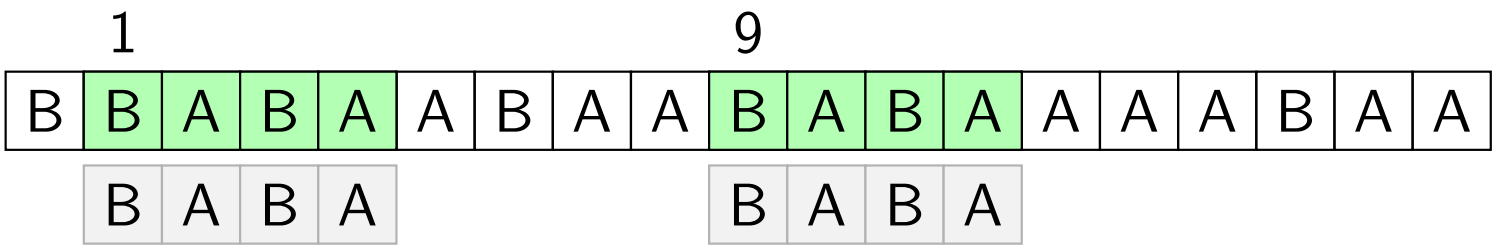
Outline

Outline

Exact String Matching

Exact String Matching

Given strings X[0..n) and Y[0..m)
compute positions i such that
X[i..$i+m$) = Y.



Exact String Matching

Exact String Matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$X[i..i+m) = Y$.

But

- No text preprocessing
- No space for text index

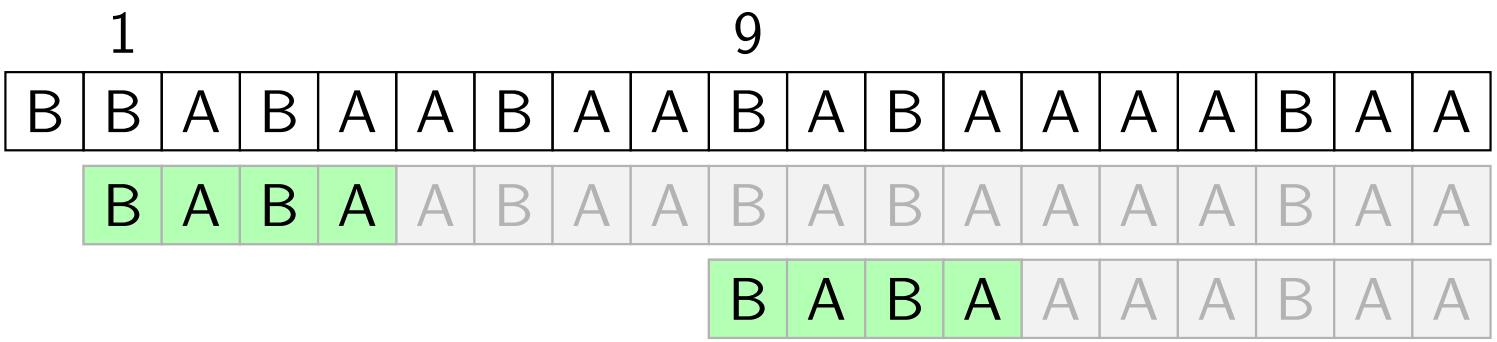
Exact String Matching

Notation

$X_i = X[i..n)$ suffix starting at position i

Exact String Matching (alternative formulation)

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
Y is a prefix of X_i.



String Range Matching

String Range Matching

Given strings $X[0..n)$, $Y[0..m_Y)$ and $Z[0..m_Z)$
compute positions i such that
$Y \leq X_i < Z$.

Notation

$$m = m_Y + m_Z$$

String Range Matching

String Range Matching

Given strings $X[0..n)$, $Y[0..m_Y)$ and $Z[0..m_Z)$
compute positions i such that
$Y \leq X_i < Z$.

Notation

$$m = m_Y + m_Z$$

Exact string matching is a special case

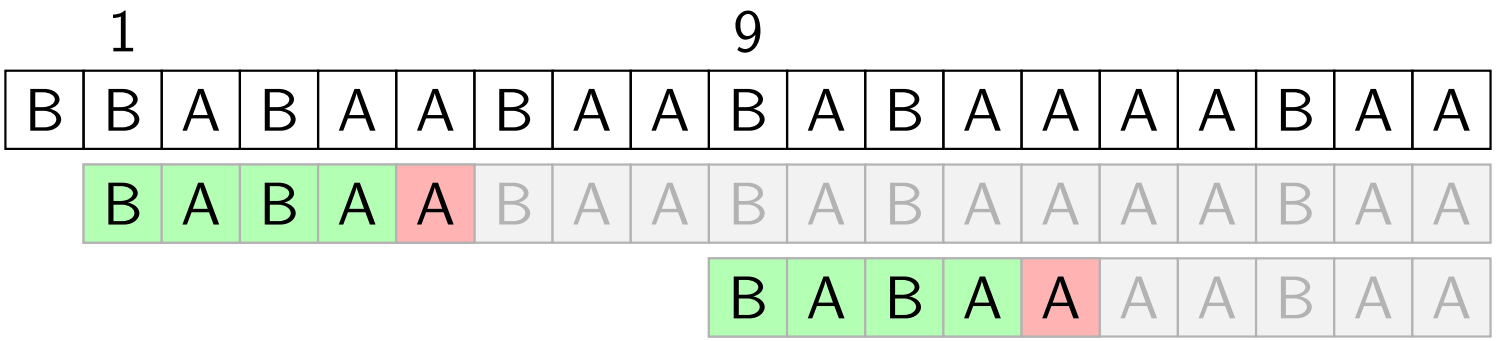
- $Z = Y\#$
- $\#$ is a special symbol larger than other symbols

String Range Matching

String Range Matching

Given strings $X[0..n)$, $Y[0..m_Y)$ and $Z[0..m_Z)$
compute positions i such that
$Y \leq X_i < Z$.

$Y = B\ A\ B\ A$ $\quad$ $Z = B\ A\ B\ A\ \#$



One-Sided String Range Matching

One-Sided String Range Matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$X_i < Y$.

One-Sided String Range Matching

One-Sided String Range Matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$X_i < Y$.

One-sided vs two-sided

- $[Y, Z) = [\varepsilon, Z) \setminus [\varepsilon, Y)$
- More similar to exact string matching (no Z)
- Simpler algorithms
- Same complexity?

Applications

Suffix array construction

- String range matching is a subproblem in some algorithms for constructing the suffix array or the Burrows–Wheeler transform
- At least three implementations
  - Kärkkäinen [2007]
  - Ferragina, Gagie & Manzini [2012]
  - Kärkkäinen & Kempa [2014]

Outline

Time–Space Complexity

Exact String Matching

- Dozens (hundreds?) of algorithms
- Many with linear time
  - Knuth, Morris & Pratt [1977]
- Some with constant extra space too
  - Extra space excludes input and output
  - Galil & Seiferas [1981]
- This is clearly time-space optimal

Selected Exact String Matching Algorithms

Algorithm	Time	Extra space	Abbrev.
Knuth, Morris & Pratt [1977]	n	m	KMP
Galil & Seiferas [1980]	n	$\log m$	GS
Crochemore [1992]	n	1	C

Selected Exact String Matching Algorithms

Algorithm	Time	Extra space	Abbrev.
Knuth, Morris & Pratt [1977]	n	m	KMP
Galil & Seiferas [1980]	n	$\log m$	GS
Crochemore [1992]	n	1	C

Basis for string range matching algorithms

- Key feature is left-to-right matching of the pattern
  - Algorithms that start matching in the middle or at the end of the pattern are not suitable for string range matching
- Other common features
  - pure left-to-right scanning of text
  - comparison-based, alphabet-independent

Algorithms for String Range Matching

Based on	Time	Extra space	Comments
KMP	n	m	Kärkkäinen [2007]
GS	n	$\log m$	counting only
C	$n \log m$	1	$\log m$ passes over the text

Algorithms for String Range Matching

Based on	Time	Extra space	Comments
KMP	n	m	Kärkkäinen [2007]
GS	n	$\log m$	counting only
C	$n \log m$	1	$\log m$ passes over the text

"Cheating" algorithms

Based on	Time	Extra space	Comments
C	n	1	overwrites Y and Z
C	n	1	reads output, reporting only

Outline

Problems

Exact string matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$X[i..i+m) = Y$.

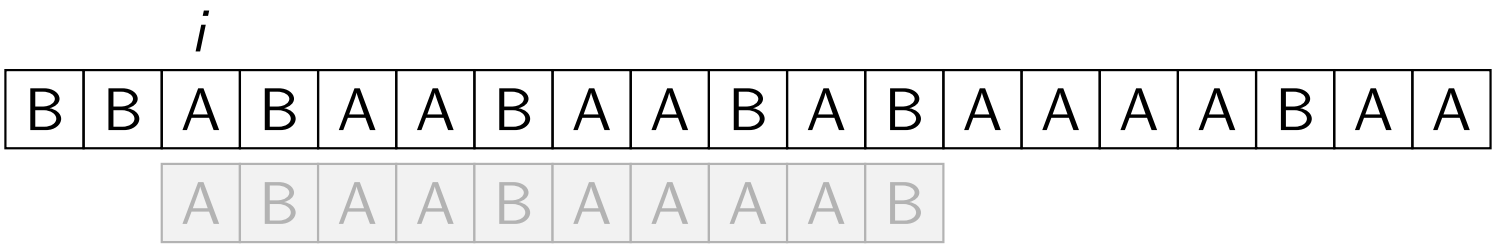
One-sided string range matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$X_i < Y$.

Basic Approach

At position i

Compute

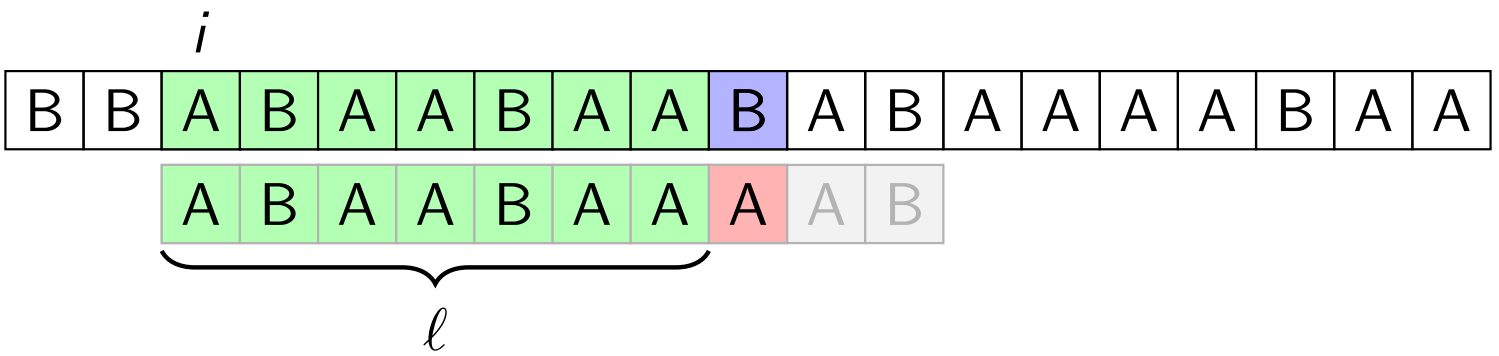


Basic Approach

At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ lcp = length of longest common prefix

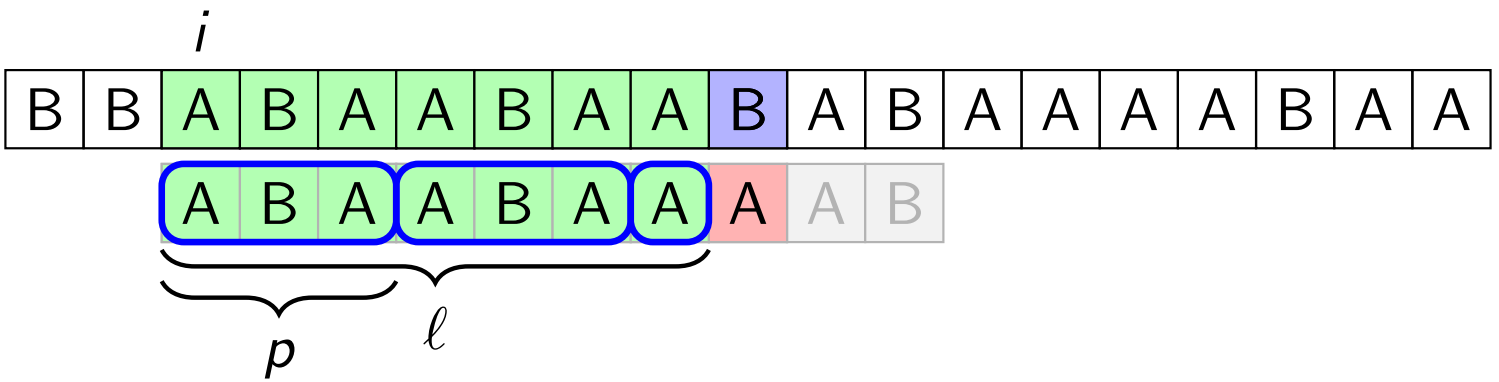


Basic Approach

At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ — lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ — per = smallest period



Basic Approach

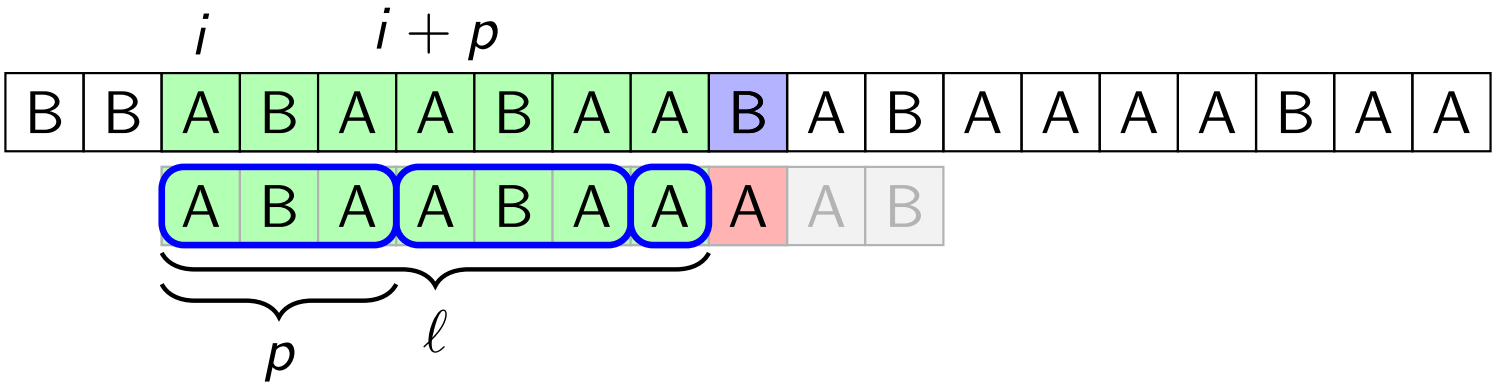
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ — lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ — per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

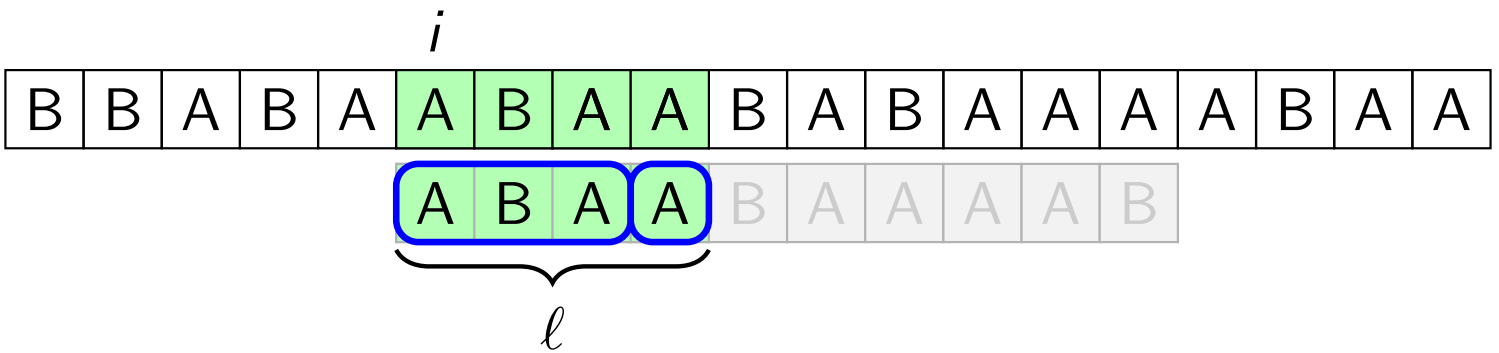
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ — lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ — per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

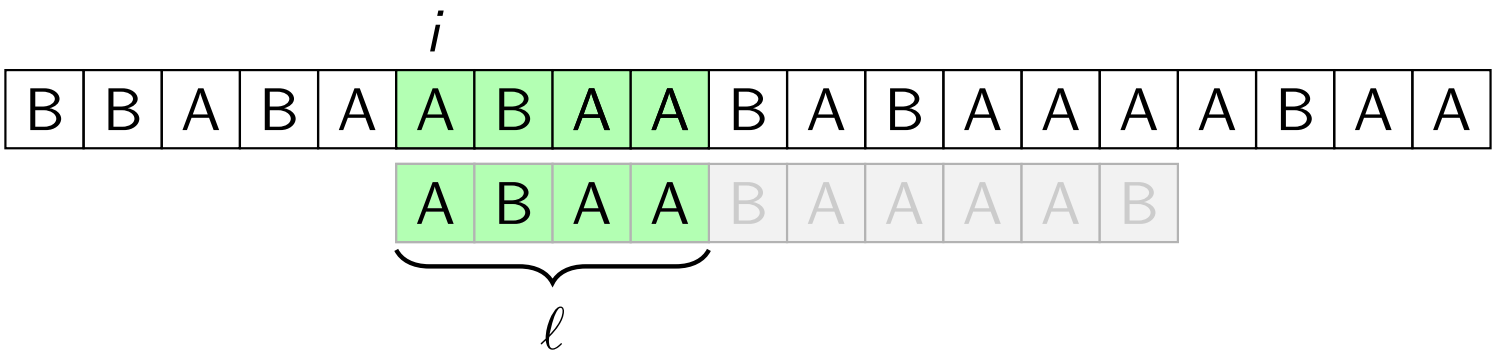
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ — lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ — per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

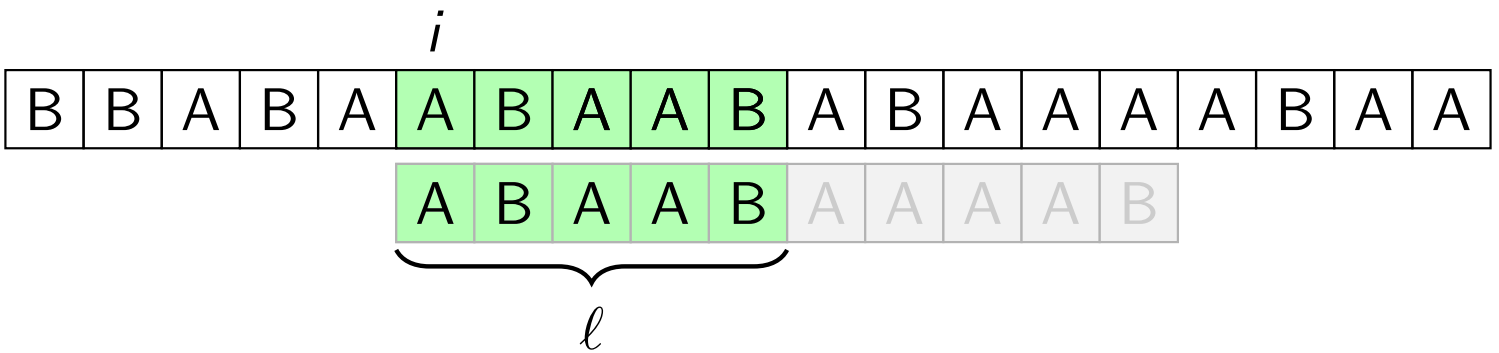
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ — lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ — per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

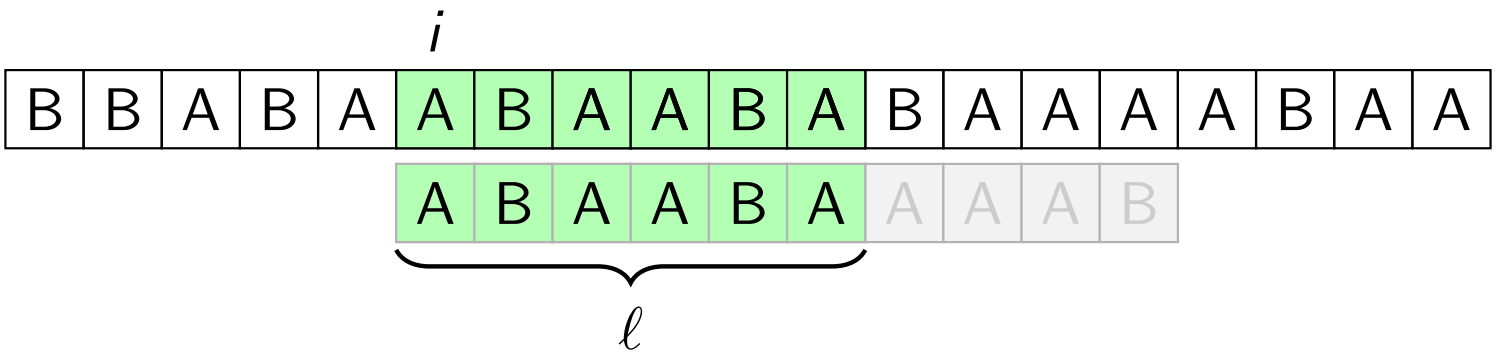
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

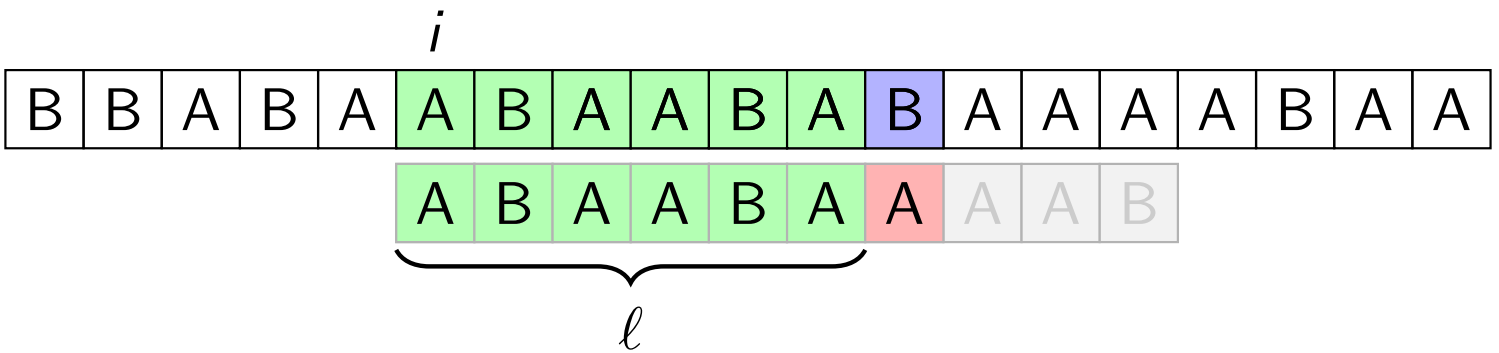
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ $\quad$ lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ $\quad$ per = smallest period

Shift by p

- $i = i + p$
- $\ell = \ell - p$



Basic Approach

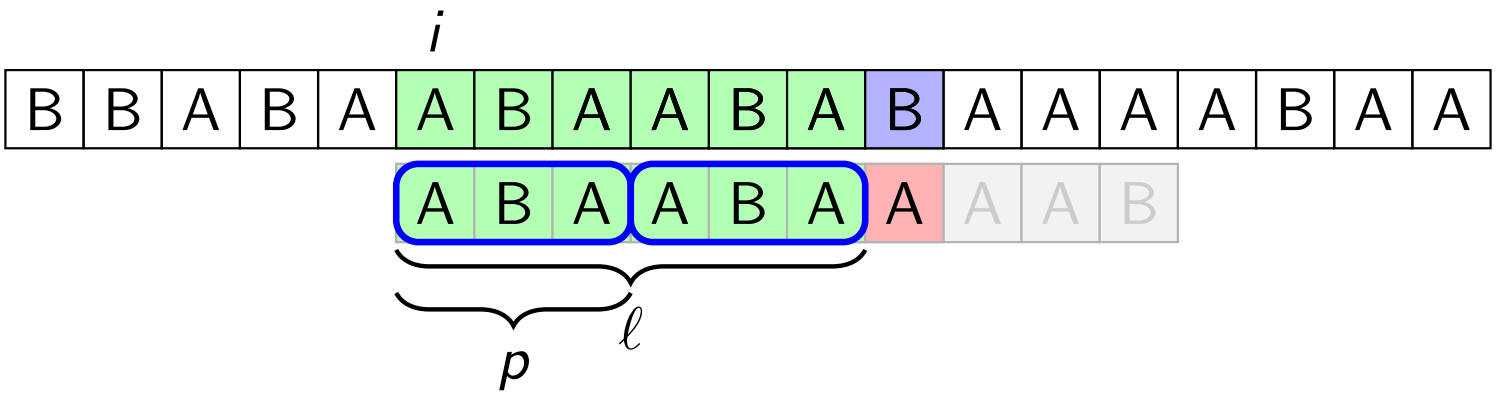
At position i

Compute

- $\ell = \mathrm{lcp}(X_i, Y)$ $\quad$ lcp = length of longest common prefix
- $p = \mathrm{per}(Y[0..\ell))$ $\quad$ per = smallest period

Shift by p

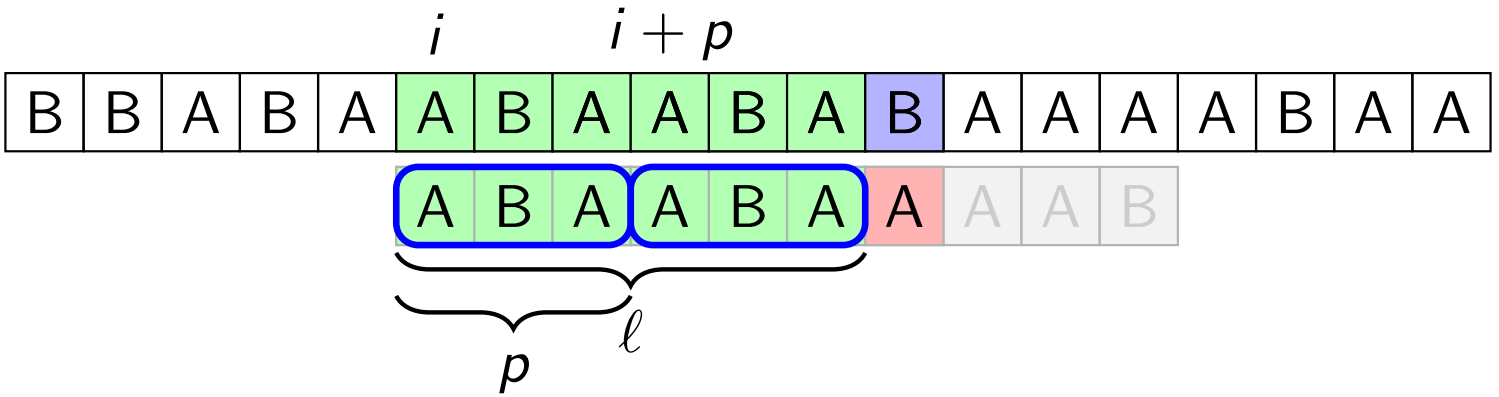
- $i = i + p$
- $\ell = \ell - p$



Basic Approach

Remaining questions

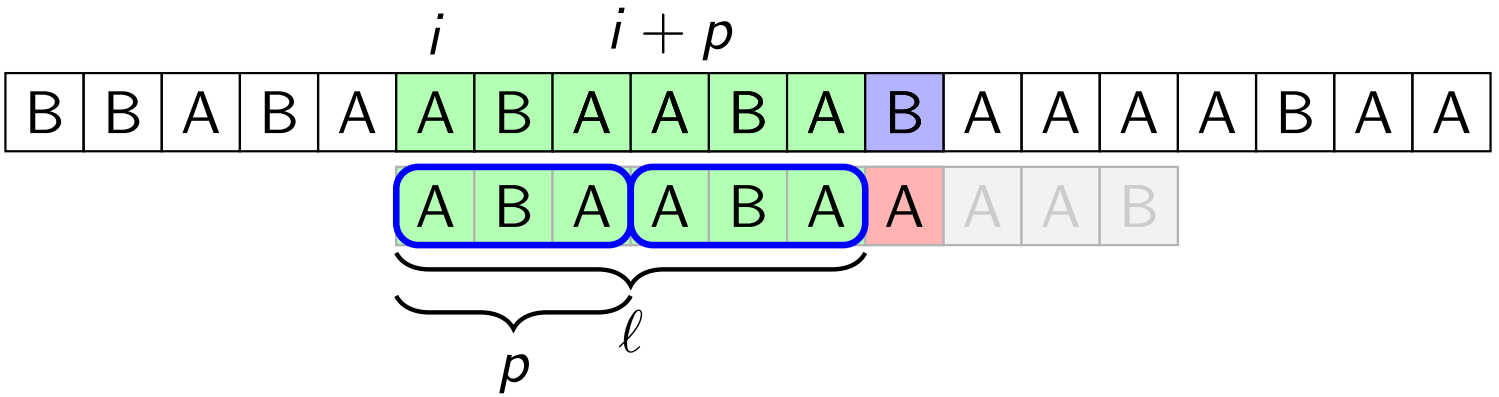
- How to compute $p = \mathrm{per}(Y[0..\ell))$ given ℓ?



Basic Approach

Remaining questions

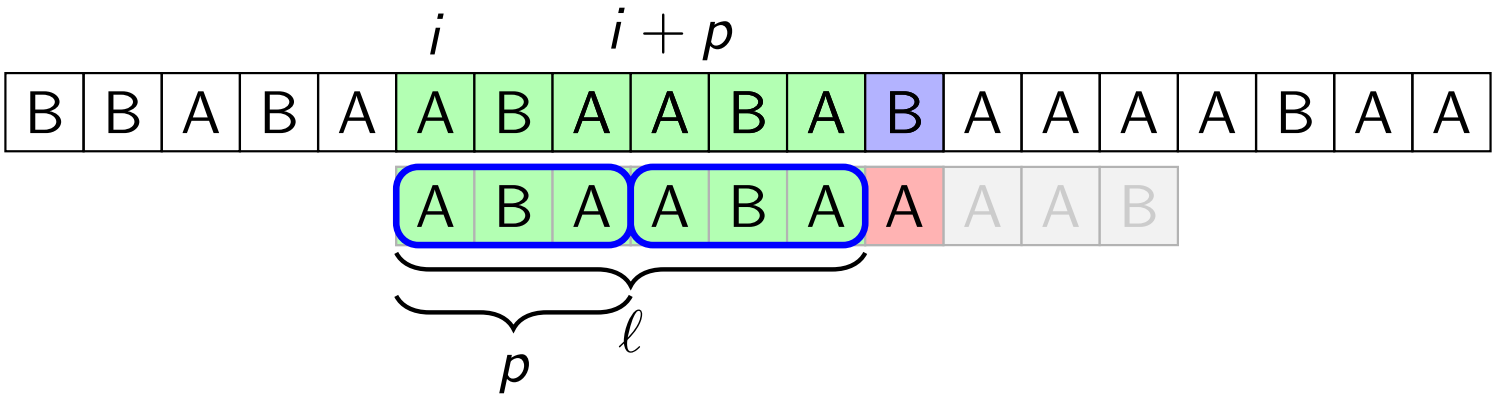
- How to compute $p = \mathrm{per}(Y[0..\ell))$ given ℓ?
  - Solved by the underlying String Matching algorithm



Basic Approach

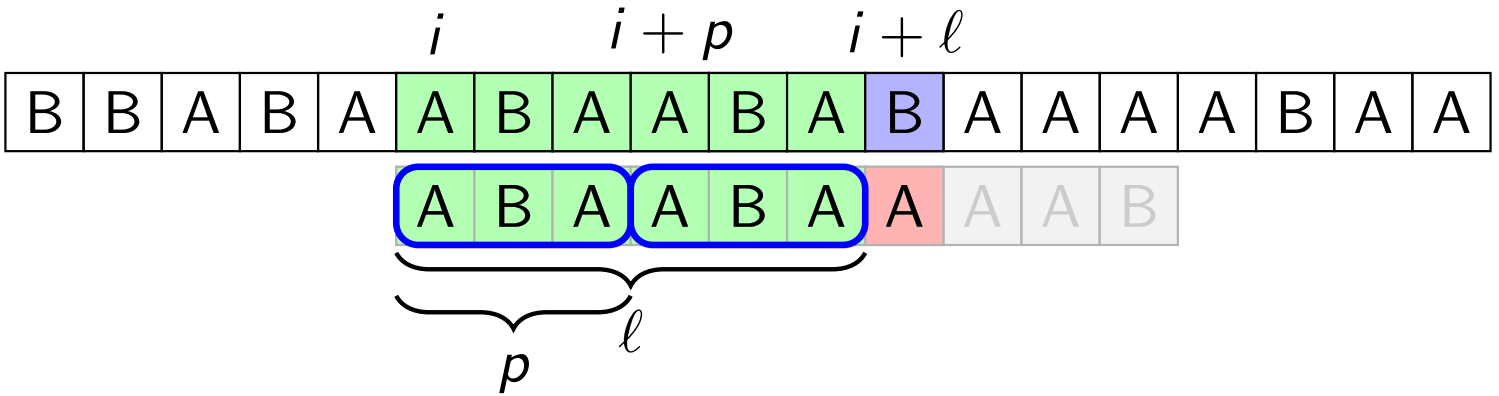
Remaining questions

- How to compute $p = \mathrm{per}(Y[0..\ell))$ given ℓ?
  - Solved by the underlying String Matching algorithm
- What about the skipped positions j, $i < j < i + p$?



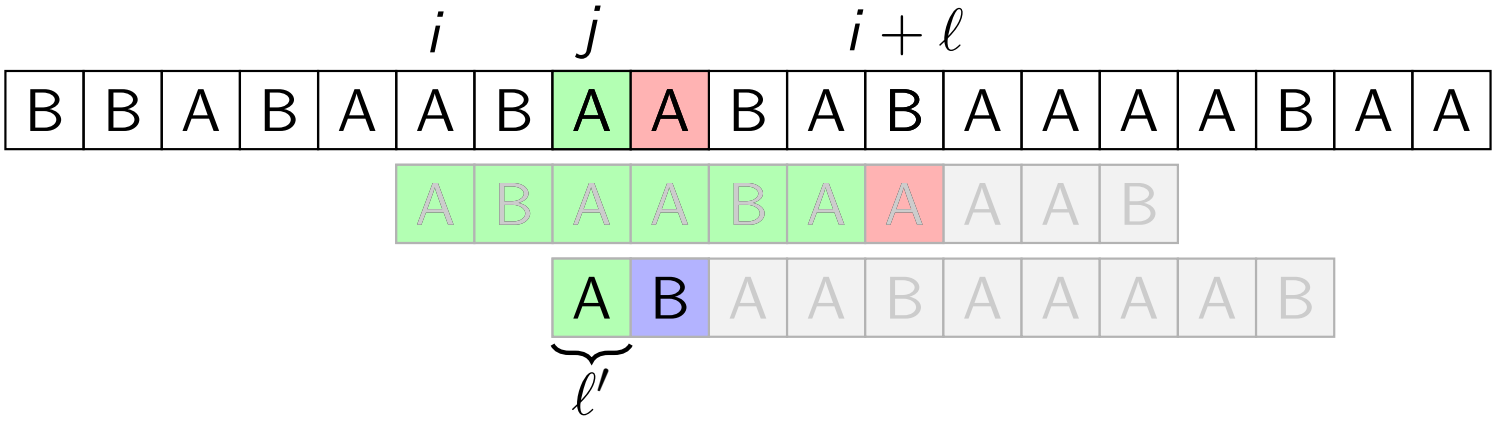
Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$



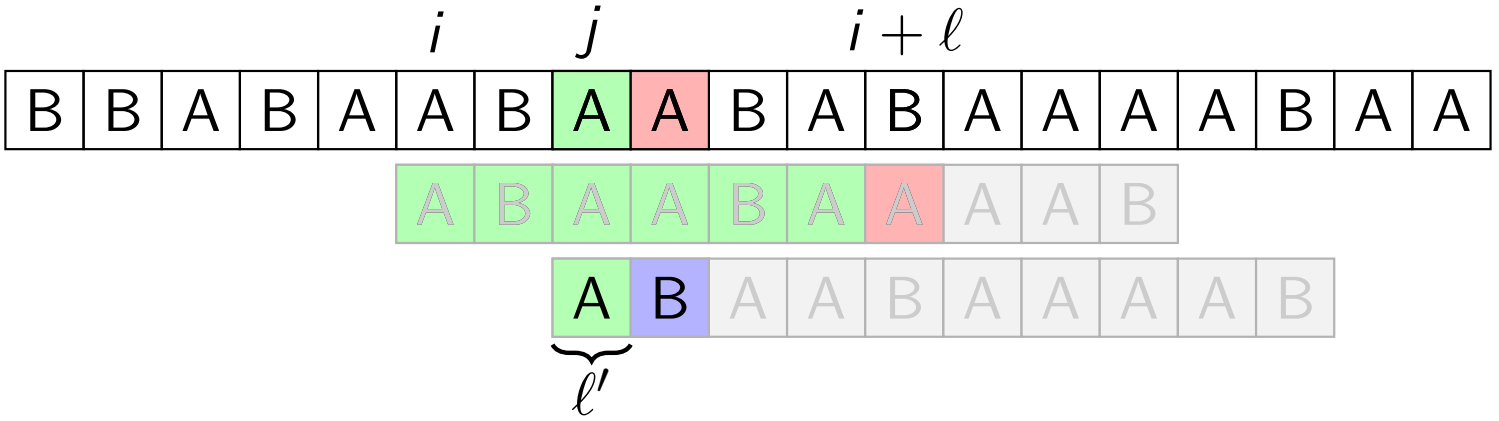
Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$
- $i < j < i + p, \quad \ell' = \mathrm{lcp}(X_j, Y)$ skipped position j



Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$
- $i < j < i + p, \quad \ell' = \mathrm{lcp}(X_j, Y)$ skipped position j

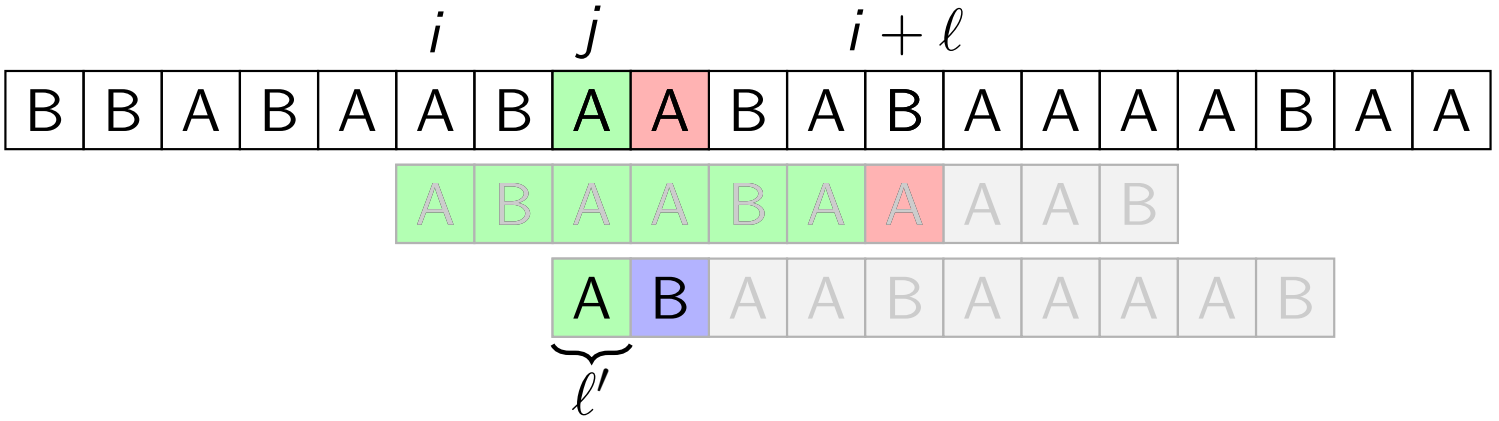


Lemma

$j + \ell' < i + \ell.$

Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$
- $i < j < i + p, \quad \ell' = \mathrm{lcp}(X_j, Y)$ skipped position j



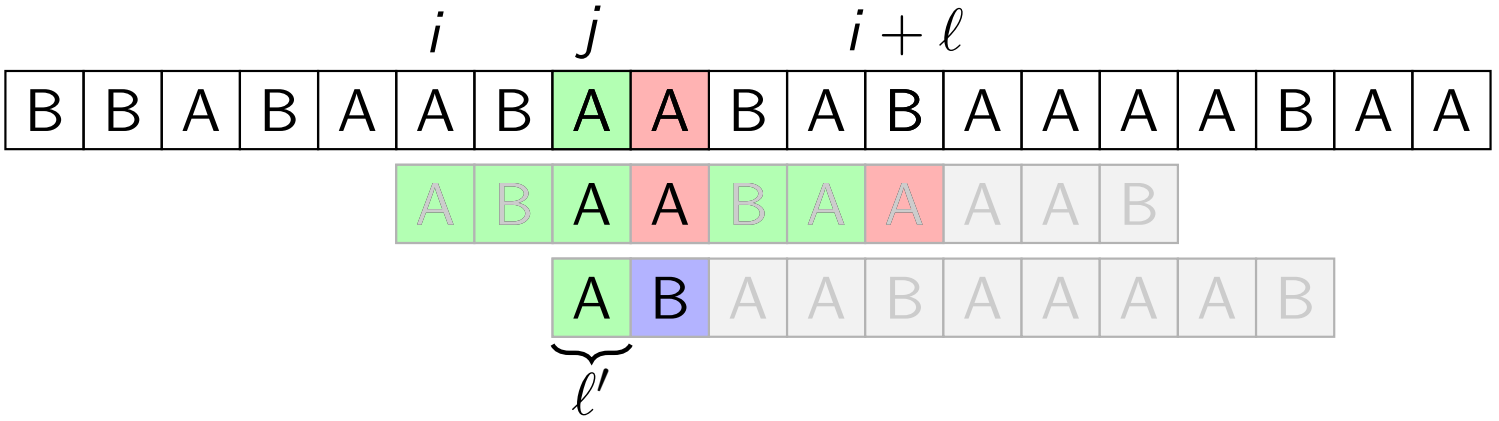
Lemma

$j + \ell' < i + \ell$. *Thus*

- $X[j..j + m) \neq Y$ *Exact string matching*

Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$
- $i < j < i + p, \quad \ell' = \mathrm{lcp}(X_j, Y)$ skipped position j



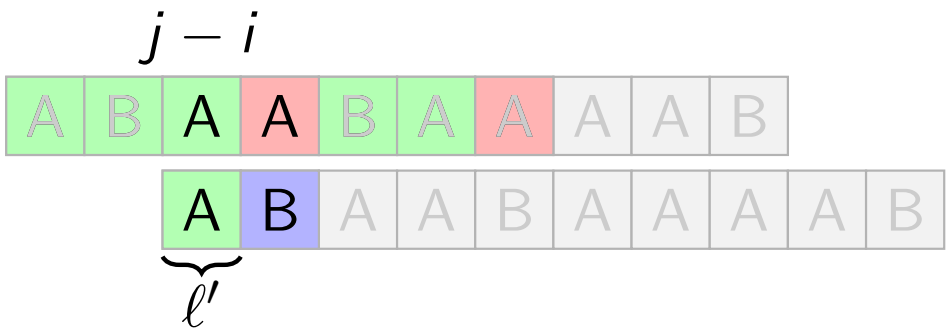
Lemma

$j + \ell' < i + \ell$. *Thus*

- $X[j..j + m) \neq Y$ *Exact string matching*
- $X_j < Y$ *iff* $Y_{j-i} < Y$ *String range matching*

Skipped Positions

- $\ell = \mathrm{lcp}(X_i, Y), \quad p = \mathrm{per}(Y[0..\ell))$
- $i < j < i + p, \quad \ell' = \mathrm{lcp}(X_j, Y)$ skipped position j



Lemma

$j + \ell' < i + \ell$. *Thus*

- $X[j..j + m) \neq Y$ *Exact string matching*
- $X_j < Y$ *iff* $Y_{j-i} < Y$ *String range matching*

Skipped Positions

The main difference between exact string matching and string range matching is how they deal with skipped positions j.

Exact string matching

- $X[j..j+m) \neq Y \implies$ ignore j

String range matching

- $X_j < Y$ iff $Y_{j-i} < Y$
- When shifting from i to $i+h$, find if $Y_k < Y$ for all $k \in [1..h)$
- Preprocess Y to answer this
- Easy when we can use $\mathcal{O}(m)$ space (KMP)
- m bits is enough
- Difficult in $o(m)$ bits of space

Skipped Positions

$\mathcal{O}(n)$ time, $\mathcal{O}(\log m)$ space

- Allow only $\mathcal{O}(\log m)$ different shift lengths
- For each shift length, store the number of skipped suffixes that are smaller than Y
- Always use the largest precomputed shift not exceeding the optimal shift
  - If the precomputed shifts are chosen correctly, the algorithm still runs in linear time
- Works only for counting version of string range matching

Skipped Positions

Restricted one-sided string range matching

Given strings $X[0..n)$ and $Y[0..m)$
compute positions i such that
$Y[0..(2/3)m) \leq X_i < Y$.

$\mathcal{O}(n \log m)$ time, $\mathcal{O}(1)$ space

- The restricted problem can be solved in linear time and constant extra space using Crochemore's algorithm
- The general one-sided problem can be reduced to $\mathcal{O}(\log m)$ restricted problems:
  - Solve the restricted problem for $Y[0..(2/3)^i m)$ for $i \in [0.. \log_{3/2} m)$ and return the (disjoint) union of results

Skipped Positions

"Cheating" algorithms

- $< m$ bits is enough to store the skipped position information
- The necessary bits can be obtained from input or output

Overwrite input

- Find the longest prefix of Y that occurs elsewhere
- Use that prefix as storage area

Copy output

- Keep track of the longest matching prefix seen so far
- Copy bits from the corresponding position in the output

Summary of New Algorithms

Based on	Time	Extra space	Comments
GS	n	$\log m$	counting only
C	$n \log m$	1	$\log m$ passes over the text

"Cheating" algorithms

Based on	Time	Extra space	Comments
C	n	1	reads output, reporting only
C	n	1	overwrites Y and Z

Outline

Better Algorithms

What can be done in linear time (without cheating)?

- reporting in $o(m)$ space?
- counting in $o(\log m)$ space?

Better Algorithms

What can be done in linear time (without cheating)?

- reporting in $o(m)$ space?
- counting in $o(\log m)$ space?

What can be done in constant extra space (without cheating)?

- $o(n \log m)$ time?
- $o(\log m)$ passes over the text?

Better Algorithms

What can be done in linear time (without cheating)?

- reporting in $o(m)$ space?
- counting in $o(\log m)$ space?

What can be done in constant extra space (without cheating)?

- $o(n \log m)$ time?
- $o(\log m)$ passes over the text?

Tradeoffs?

Lower Bounds

Separating problems

- Is string range matching harder than exact string matching?
  - Conjecture yes
- Is reporting harder than counting?
- Is one-sided problem easier than two-sided?

Matching upper and lower bounds

- Prove time-space optimality of algorithms

Thank you!