Fast q-gram Mining on SLP Compressed Strings

Kyushu University

○Keisuke Goto, Hideo Bannai,

Shunsuke Inenaga, Masayuki Takeda

Sponsored by Hara Research Foundation

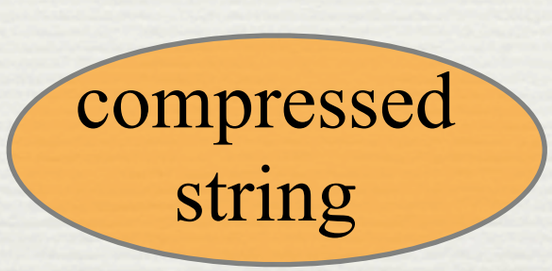
Background

- Large scale string data is usually stored in compressed form

Large Scale String

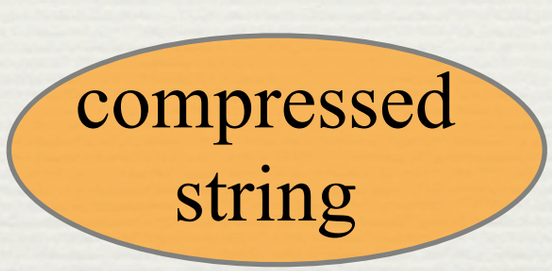
Background

- Large scale string data is usually stored in compressed form



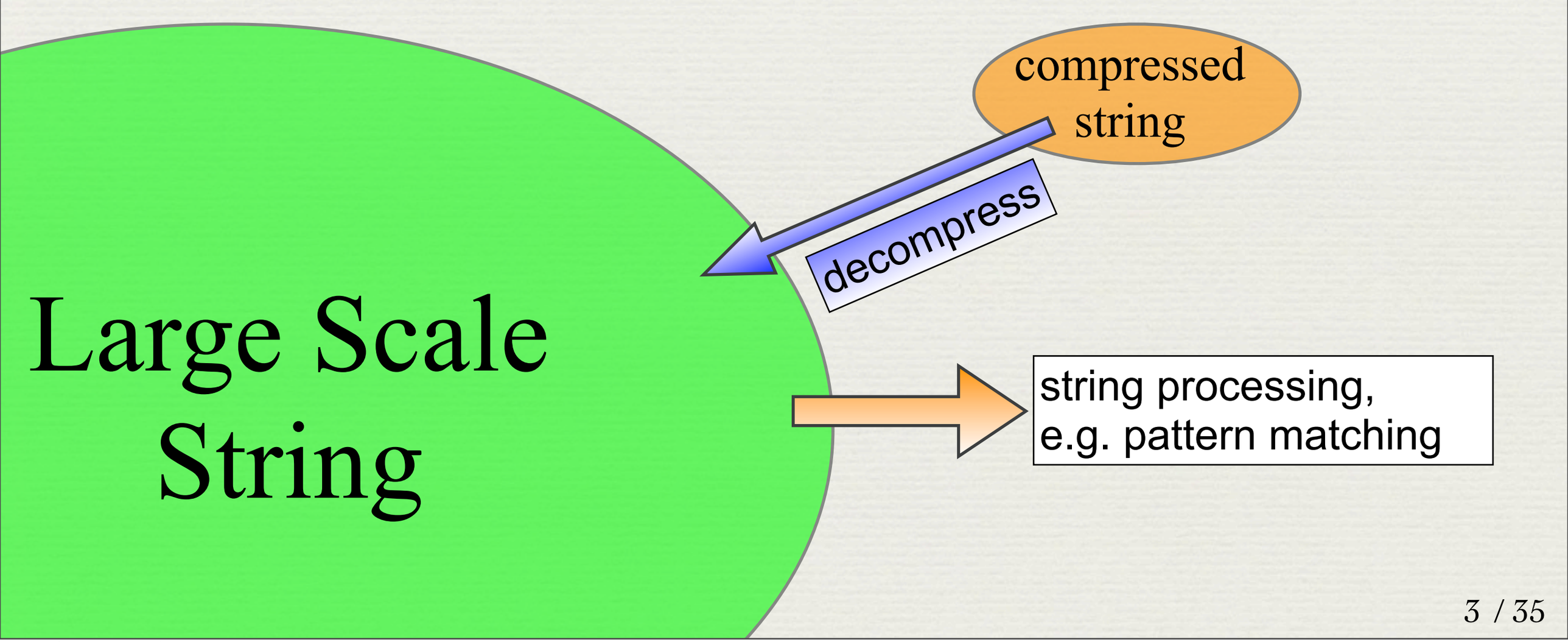
Background

- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time



Background

- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time



Background

- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time

Large Scale String

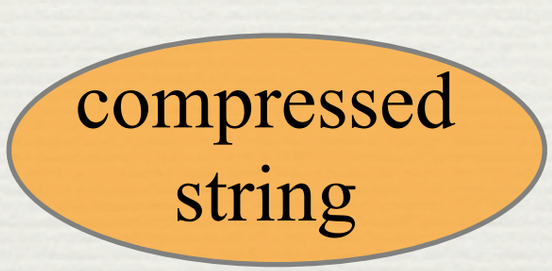
compressed string

decompress

string processing,
e.g. pattern matching

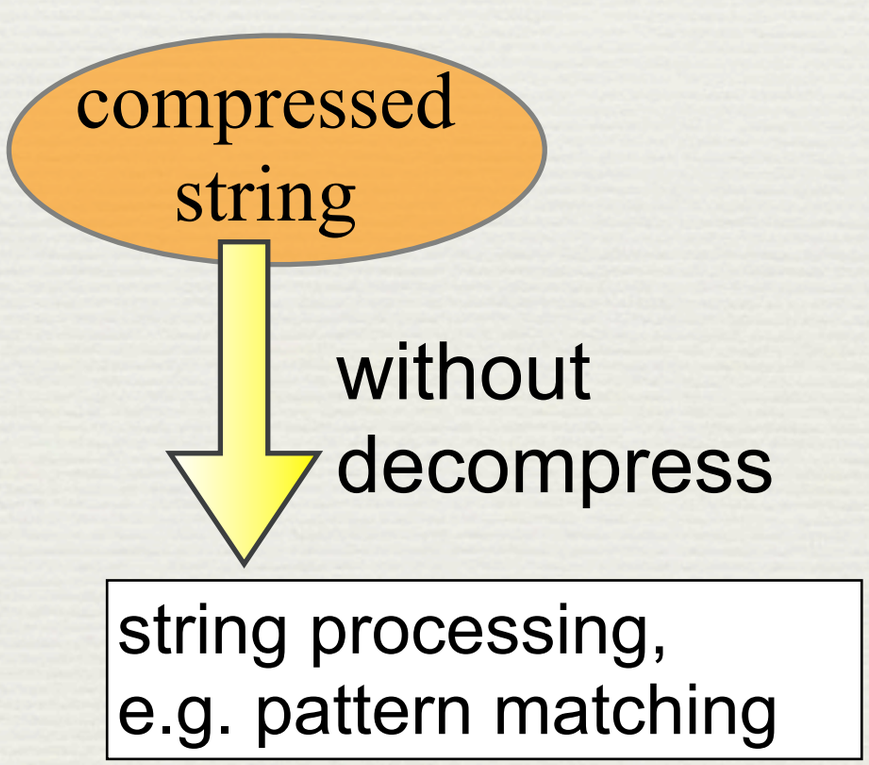
Background

- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time
- One solution is to process compressed strings **without explicit decompression**



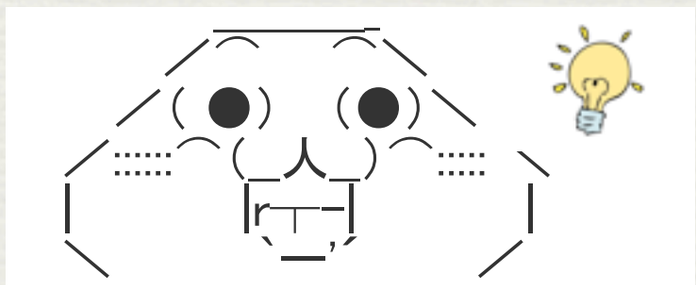
Background

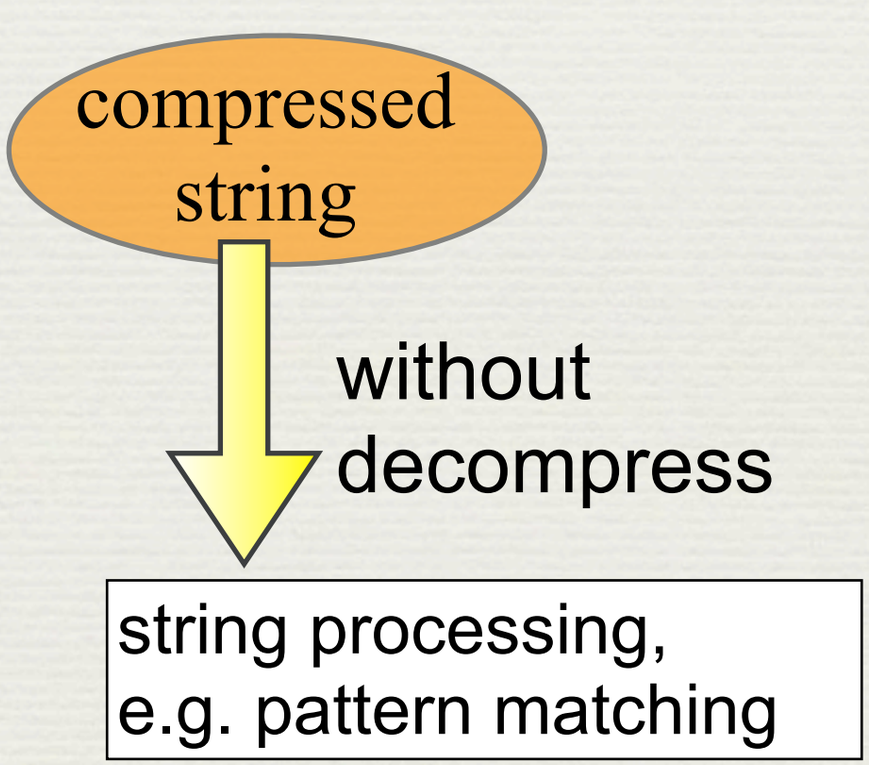
- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time
- One solution is to process compressed strings **without explicit decompression**



Background

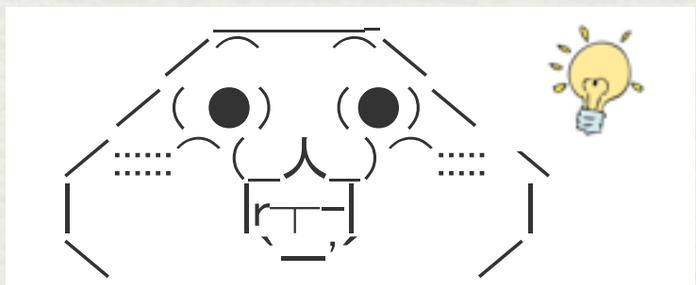
- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time
- One solution is to process compressed strings **without explicit decompression**

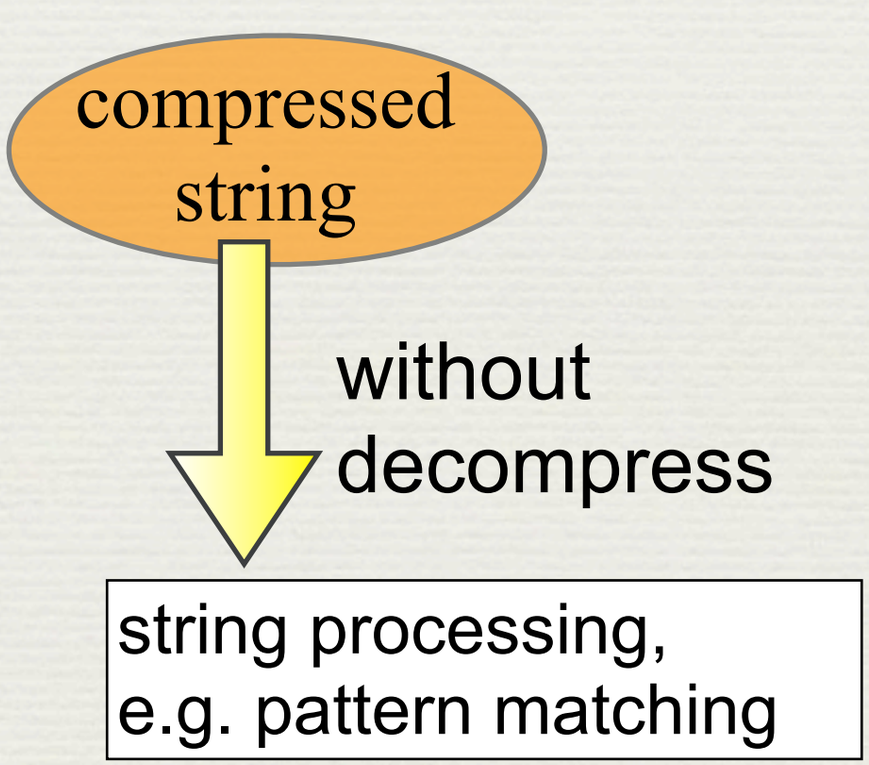




Background

- Large scale string data is usually stored in compressed form
- In order to process such data, we usually need to decompress, which requires a lot of space and time
- One solution is to process compressed strings **without explicit decompression**





this work: algorithm for computing ***q*-gram frequencies** on compressed strings

q-gram Frequencies Problem

q-gram Frequencies Problem

Input：string T, integer q

Output：$|Occ(T, x)|$ for $\forall\, x \in \Sigma^q$ s.t. $|Occ(T, x)| > 0$

where $Occ(T, P)=\{\, i \mid T[i..(i+|P|-1)] = P,\ 0 \leqq i < |T| - |P| \,\}$

i.e. frequencies of all length-q strings (q-grams) occuring in T

Example $q = 3$

T = abaababaab

aba bab
baa aba
aab baa
aba aab

q-gram	Frequency
aab	2
aba	3
baa	2
bab	1

- q-gram frequencies are important features of strings, widely used in data mining and machine learning

Straight Line Program (SLP)

Definition (Straight Line Program)

Straight Line Program is a context free grammar in the Chomsky normal form that derives a single string

$X_1 = \mathrm{expr}_1, X_2 = \mathrm{expr}_2, ..., X_n = \mathrm{expr}_n$

$\mathrm{expr}_i \in \Sigma$ or

$\mathrm{expr}_i = X_l \cdot X_r \ (l, r < i)$

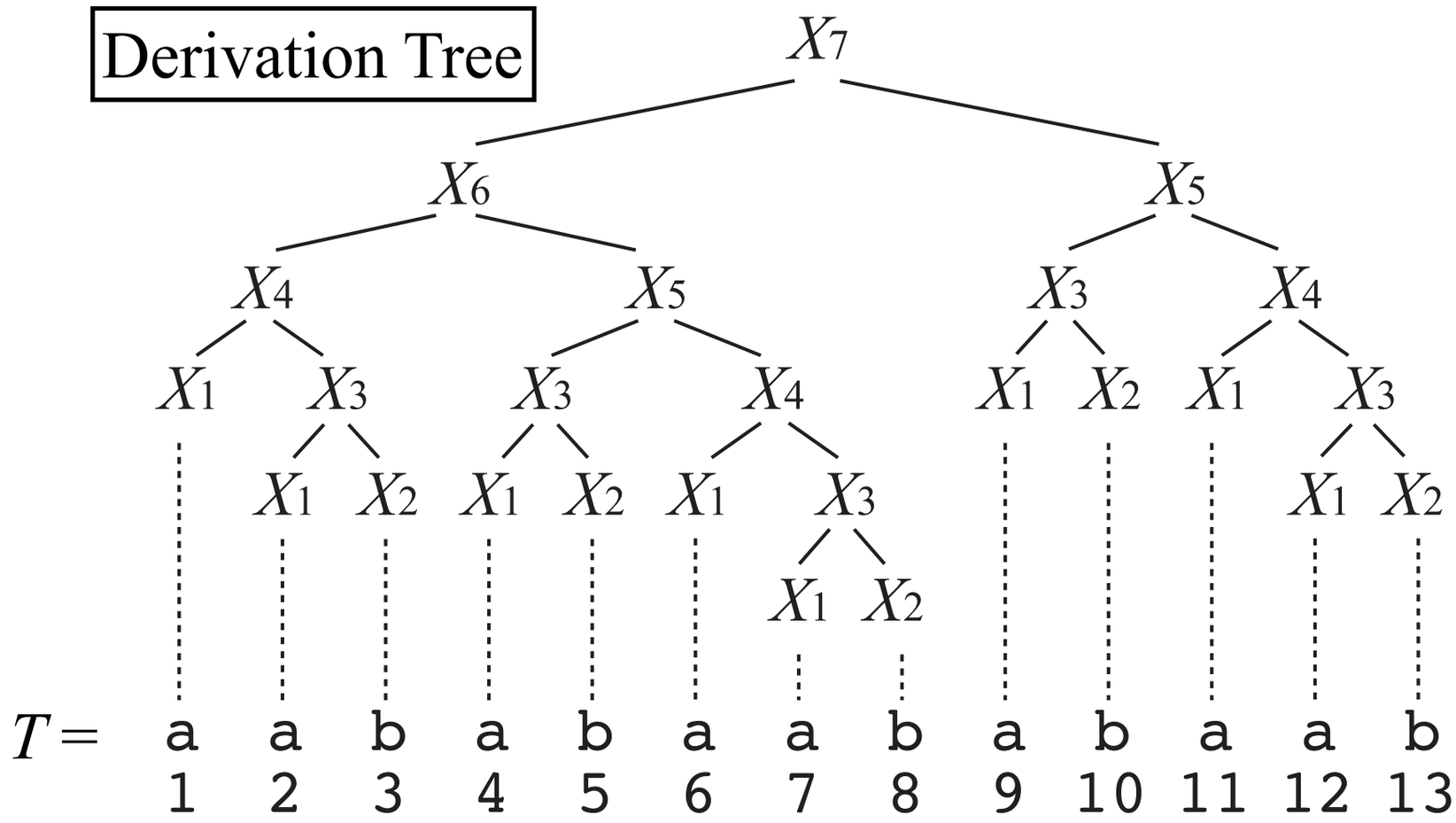
- SLP can represent the output of well-known compression schemes
  - e.g. RE-PAIR SEQUITUR, LZ78, LZW
- LZ77, LZSS can be quickly transformed to SLP **[Rytter, 2003]**

Example SLP

$$\mathcal{D} = \begin{aligned} X_1 &= \mathrm{a} \\ X_2 &= \mathrm{b} \\ X_3 &= X_1 X_2 \\ X_4 &= X_1 X_3 \\ X_5 &= X_3 X_4 \\ X_6 &= X_4 X_5 \\ X_7 &= X_6 X_5 \end{aligned}$$

$$n = |\mathcal{D}| = 7$$

Derivation Tree



Length of decompressed string can be $\Theta(2^n)$

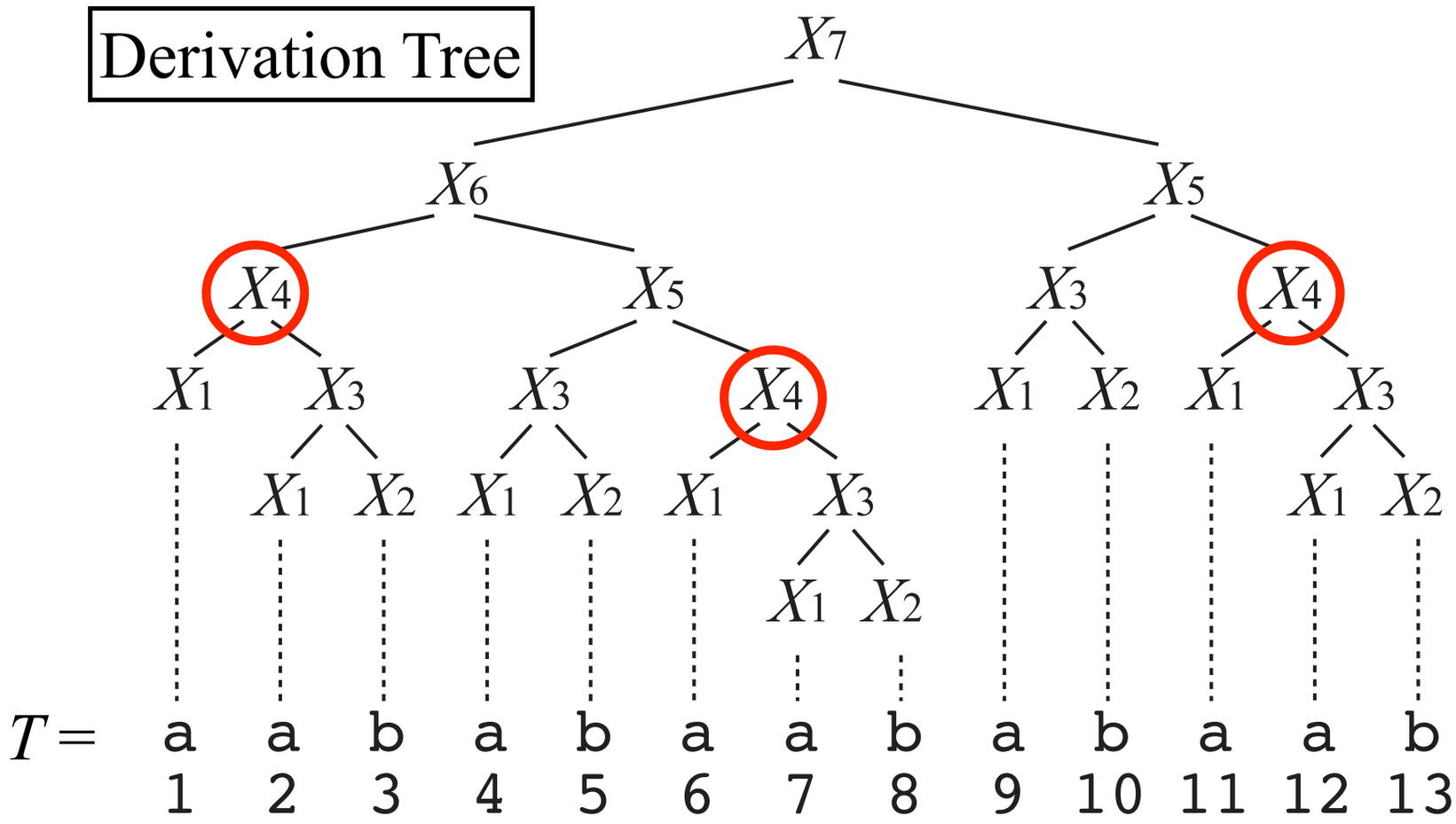
$vOcc(X_i)$: the number of occurrence X_i in the derivation tree of X_n

Example SLP

$$\mathcal{D} = \begin{aligned} X_1 &= \mathrm{a} \\ X_2 &= \mathrm{b} \\ X_3 &= X_1 X_2 \\ X_4 &= X_1 X_3 \\ X_5 &= X_3 X_4 \\ X_6 &= X_4 X_5 \\ X_7 &= X_6 X_5 \end{aligned}$$

$n = |\mathcal{D}| = 7$

Derivation Tree



Length of decompressed string can be $\Theta(2^n)$

$vOcc(X_i)$: the number of occurrence X_i in the derivation tree of X_n

Example $vOcc(X_4) = 3$

Computing q-gram frequencies on uncompressed strings

$O(q|T|\log|\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- Simply scan string T and increment frequency of each q-gram $T[i\,..\,i+q-1]$

Example

$q = 3$

$T =$ abaababaab

aba bab
baa aba
aab baa
aba aab

q-gram	Frequency
aab	2
aba	3
baa	2
bab	1

- Each access to associative array: $O(q \log|\Sigma|)$ time
- Total: $O(q|T|\log|\Sigma|)$ time, $O(q|T|)$ space

$O(|T|)$-time Algorithm

- Construct and simply scan the Suffix Array, LCP Array of T for intervals where LCP ≧ q

Example $q = 3$

$T =$ `abaababaab`

Intervals where LCP ≧ q, represent all occurrences of a specific q-gram which occur more than once

Output position i and frequency of $T[i \,..\, i+q-1]$

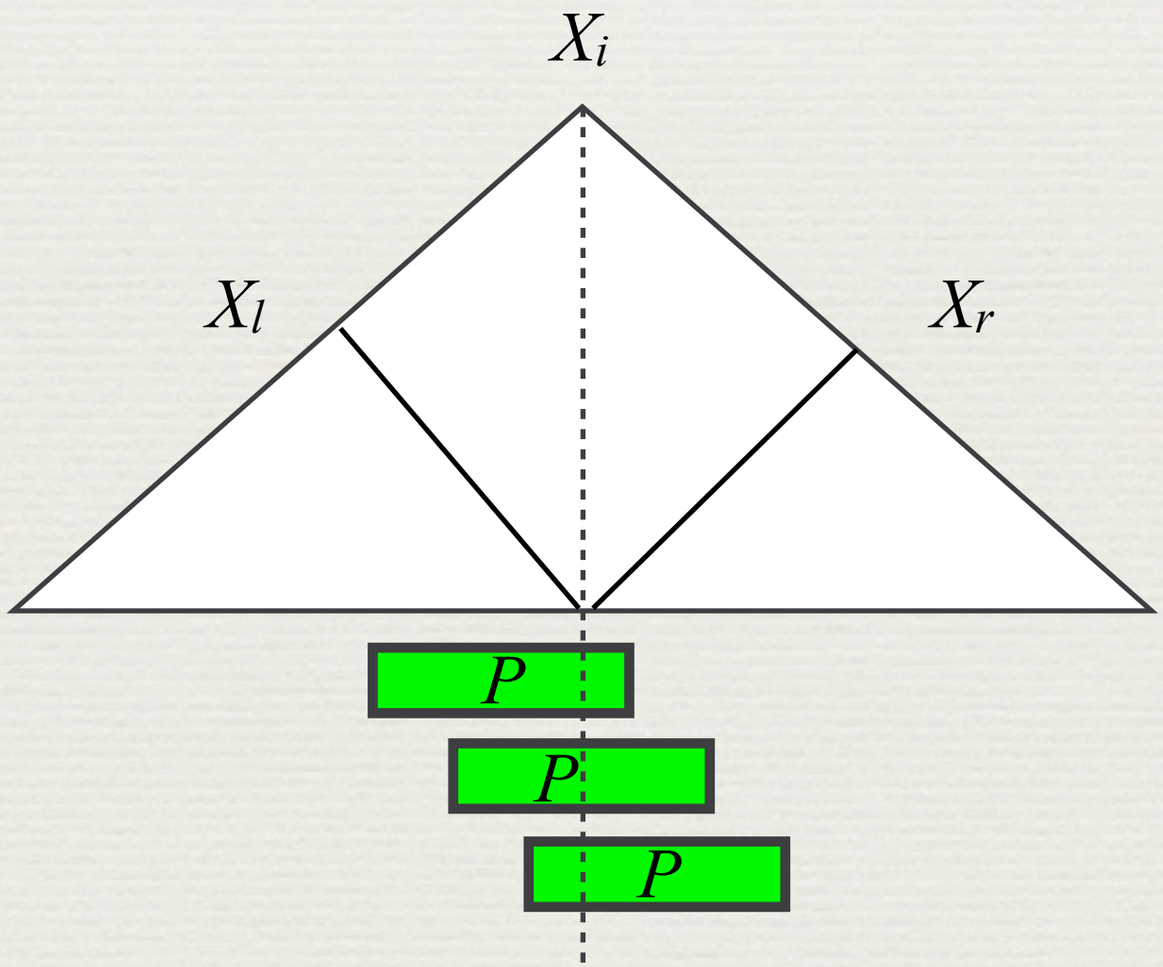
i	SA[i]	LCP[i]	T[SA[i] .. $\|T\|$]
0	7	0	aab
1	2	3	aababaab
2	8	1	ab
3	5	2	abaab
4	0	5	abaababaab
5	3	3	ababaab
6	9	0	b
7	6	1	baab
8	1	4	baababaab
9	4	2	babaab

- Suffix Array, LCP Array construction: $O(|T|)$ time & space **[Kasai et al, 2001]**
- Total: $O(|T|)$ time & space **e.g. [Kärkkäinen & Sanders, 2003]**

Computing q-gram frequencies on SLPs

*Occ**: Crossing Occurrences

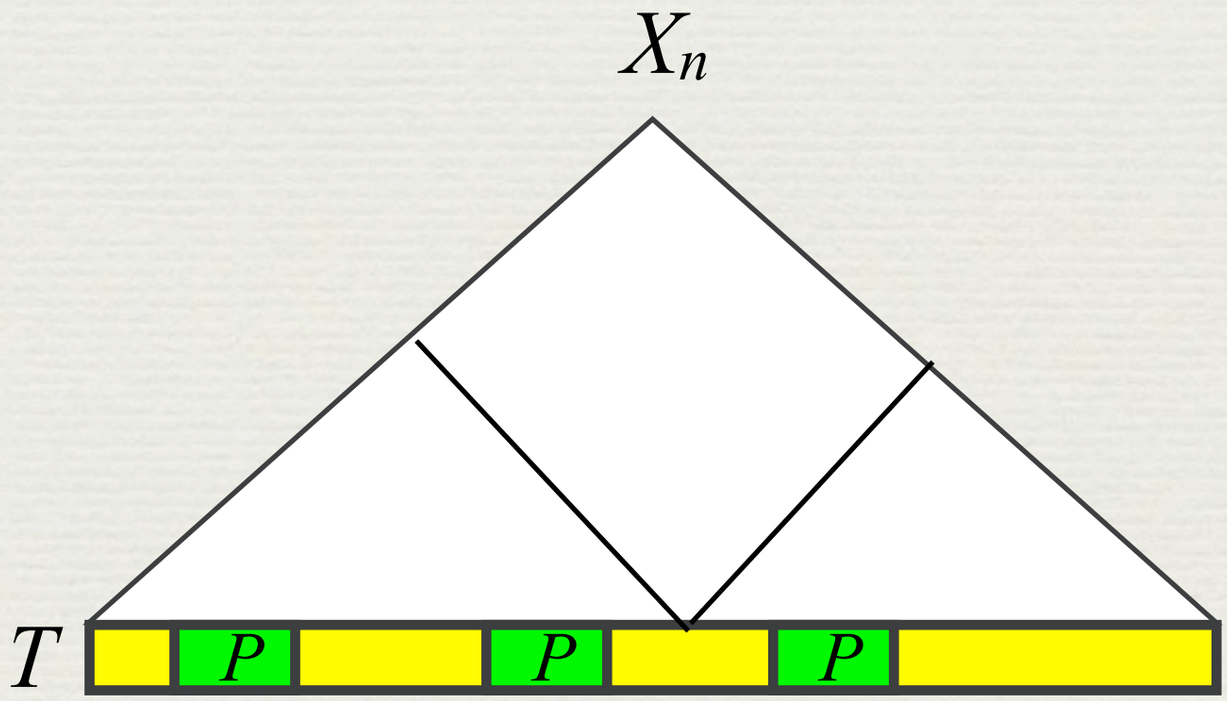
- For $X_i=X_l\,X_r$, Occurrences of *P is crossing occurrences* of *Xi* if occurrences of *P* start in X_l and end in X_r
- We denote such occurrence by Occ*(Xi, P)



Lemma

Lemma

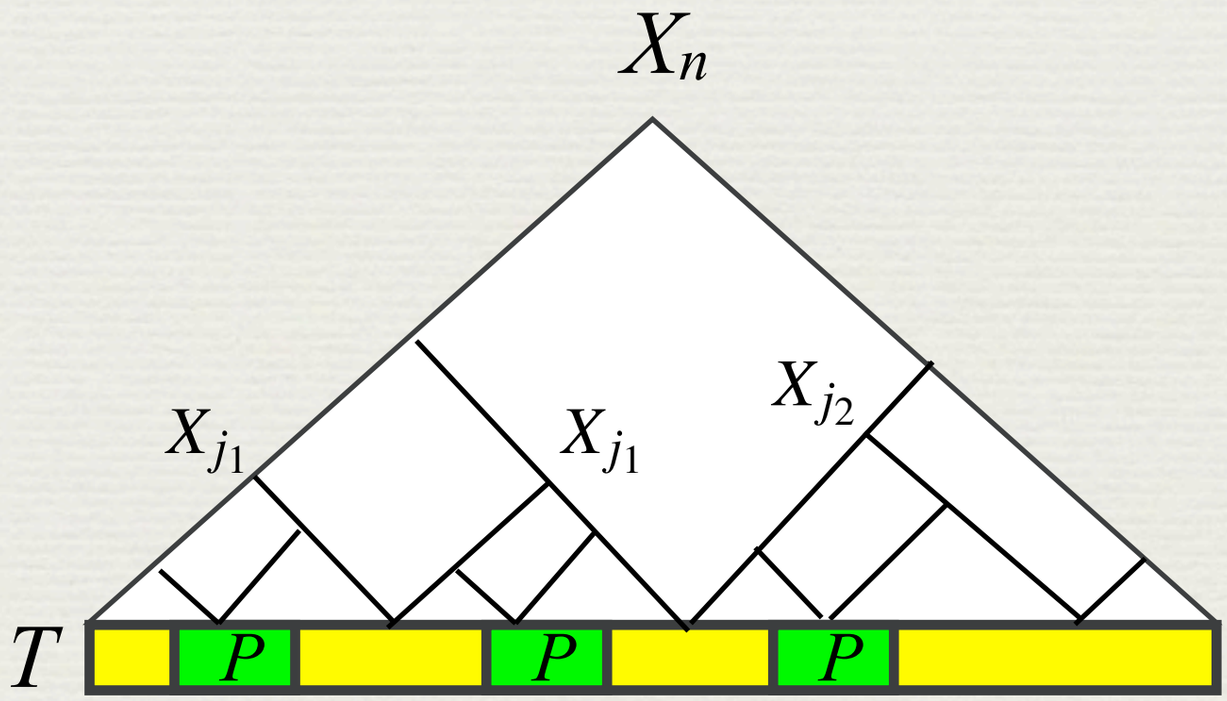
$$|Occ(T,P)| = \sum_{i}^{n} |Occ^{*}(X_i,P)| \cdot vOcc(X_i)$$



Lemma

Lemma

$$|Occ(T,P)| = \sum_{i}^{n} |Occ^{*}(X_i,P)| \cdot vOcc(X_i)$$

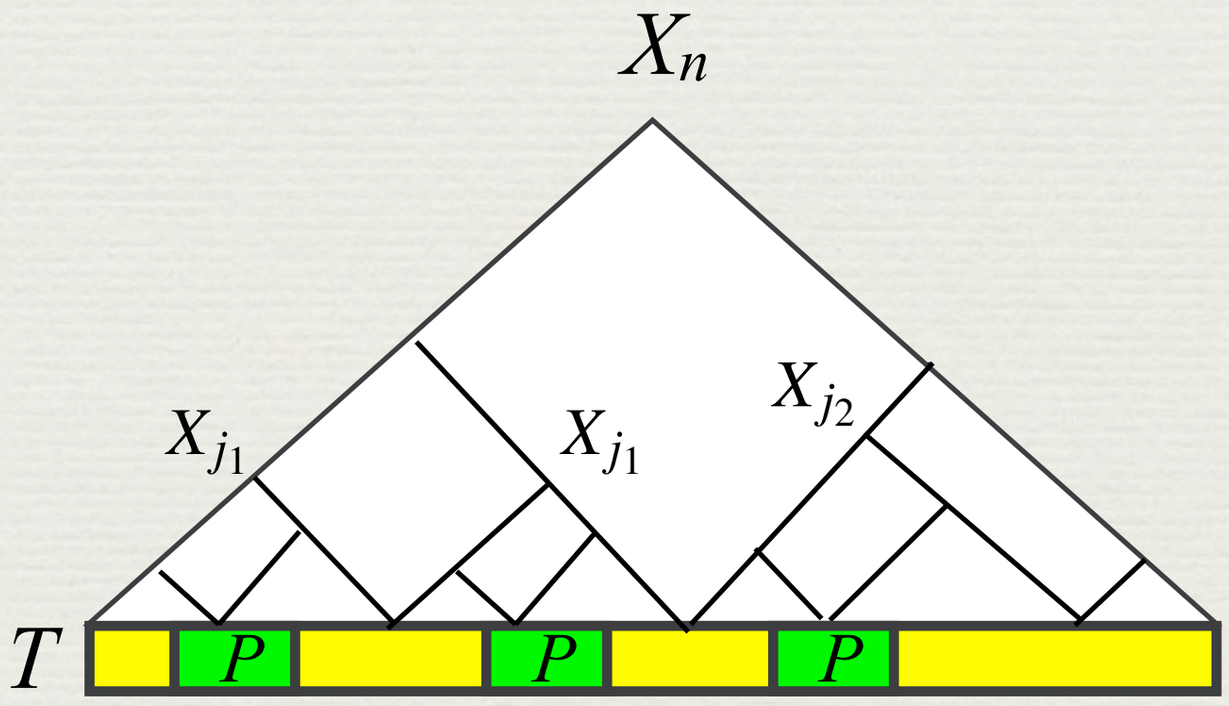


Lemma

Lemma

$$|Occ(T,P)| = \sum_{i}^{n} |Occ^{*}(X_i,P)| \cdot vOcc(X_i)$$

For any substring P of T,
there exists $X_i = X_l X_r$ such that P occurs
crossing the boundary of X_l and X_r



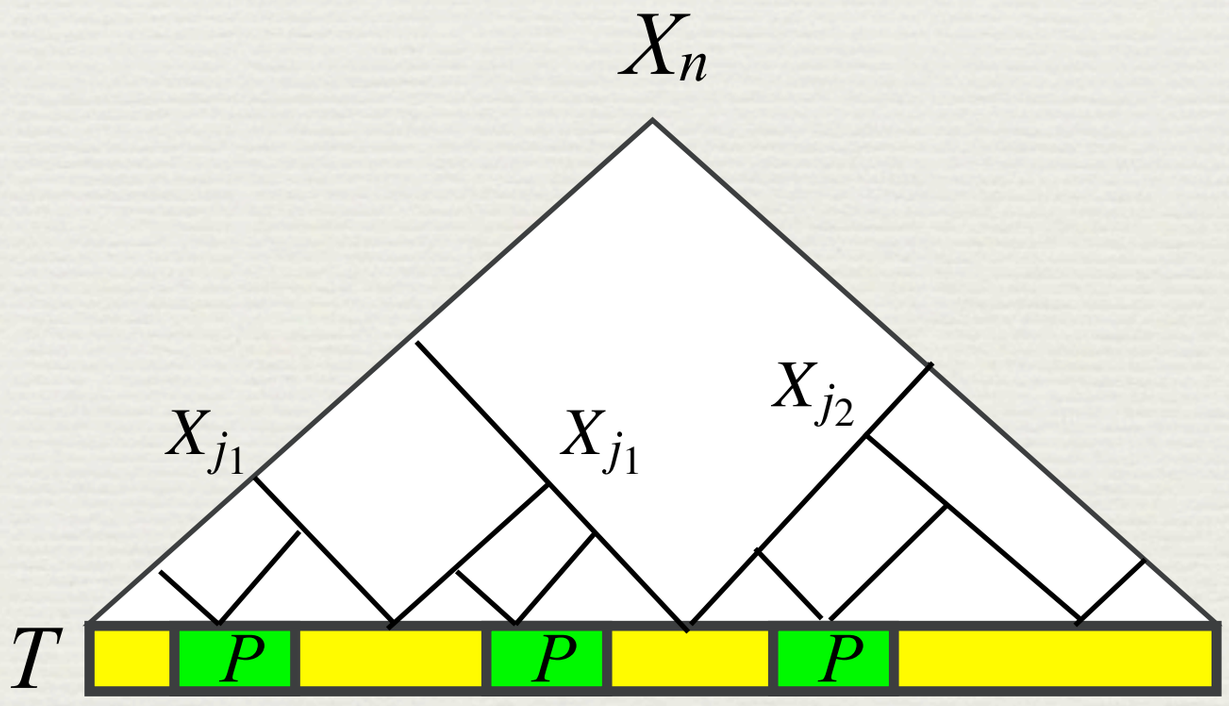
Lemma

Lemma

$$|Occ(T,P)| = \sum_{i}^{n} |Occ^{*}(X_i,P)| \cdot vOcc(X_i)$$

For any substring P of T,
there exists $X_i = X_l X_r$ such that P occurs crossing the boundary of X_l and X_r

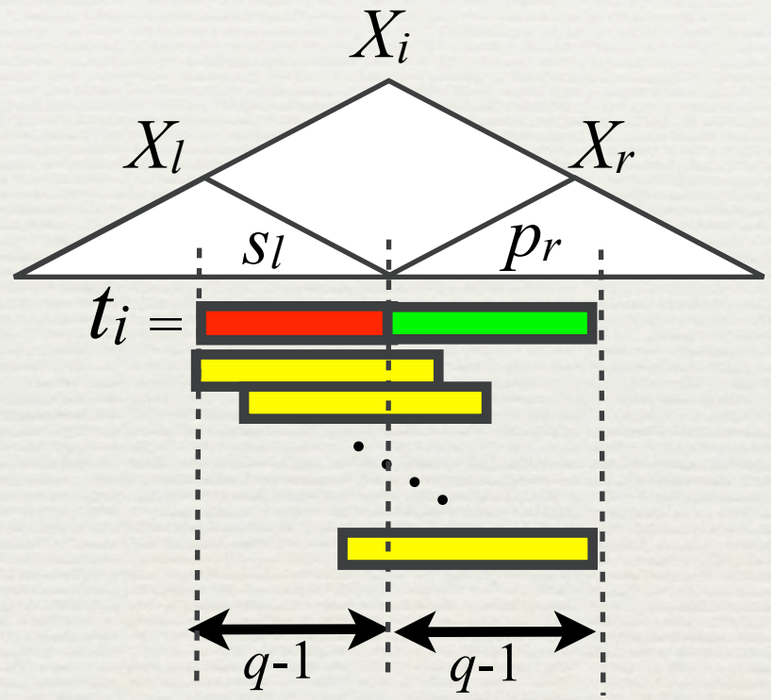
The number of P crossing the boundary of X_l and X_r is $vOcc(X_i)$, thus lemma holds.



multiple weighted q-gram frequencies problem

for all $X_i = X_l, X_r$, let $t_i = s_l p_r$ where s_l: length-(q-1) suffix of X_l
p_r: length-(q-1) suffix of X_r

Occurrences of q-gram in t_i correspond to crossing occurrences of X_i



We reduction q-gram frequencies problem to *multiple weighted q-gram frequencies problem*

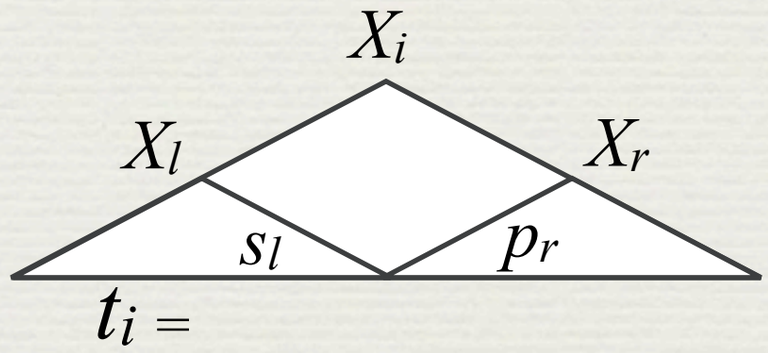
multiple weighted q-gram frequencies problem

Input： string set $S = \{t_1, t_2, \ldots, t_n\}$, integer set $w = \{w_1, w_2, \ldots, w_n\}$

Output： $\sum_{i=1}^{n} |Occ(t_i, x)| \cdot w_i$ for $\forall\, x \in \Sigma^q$

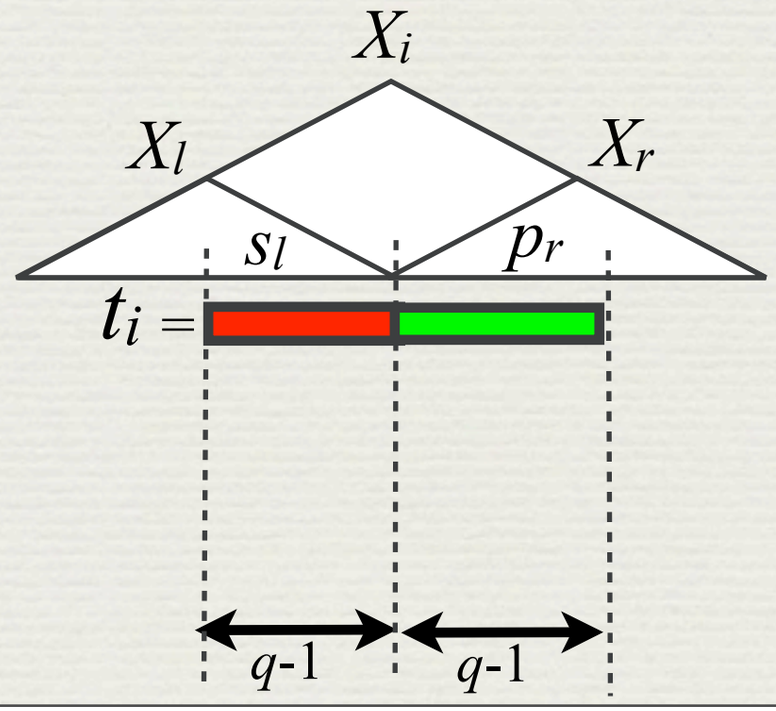
Our Strategy

- STEP 1. for all $X_i = X_l\, X_r$, compute $vOcc(X_i)$



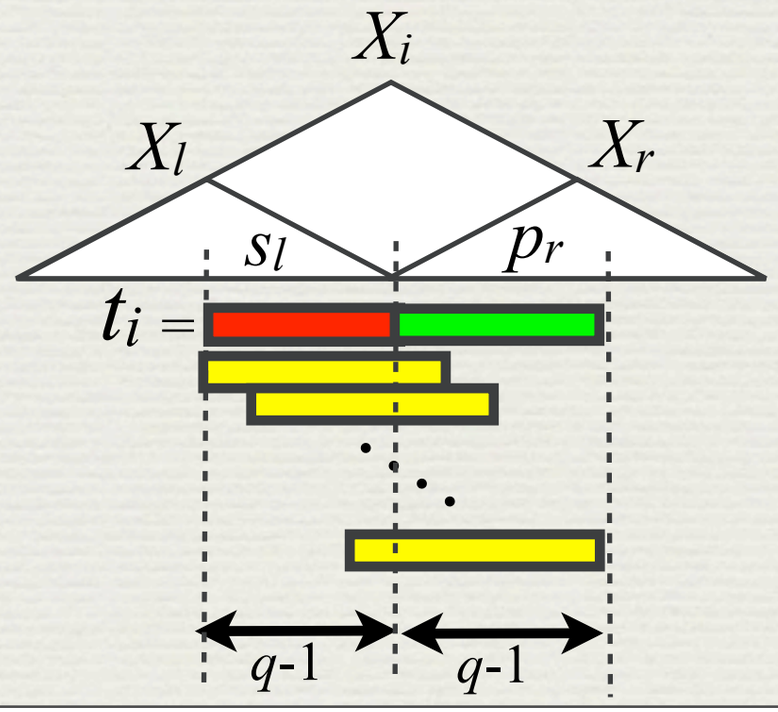
Our Strategy

- STEP 1. for all $X_i = X_l\, X_r$, compute $vOcc(X_i)$
- STEP 2. for all $X_i = X_l\, X_r$, compute decompressed prefix p_i and suffix s_i of length at most q-1, and $t_i = s_l p_r$



Our Strategy

- STEP 1. for all $X_i = X_l X_r$, compute $vOcc(X_i)$
- STEP 2. for all $X_i = X_l X_r$, compute decompressed prefix p_i and suffix s_i of length at most q-1, and $t_i = s_l p_r$
- STEP 3. solve the multi weighted q-gram frequencies for all t_i and $vOcc(X_i)$

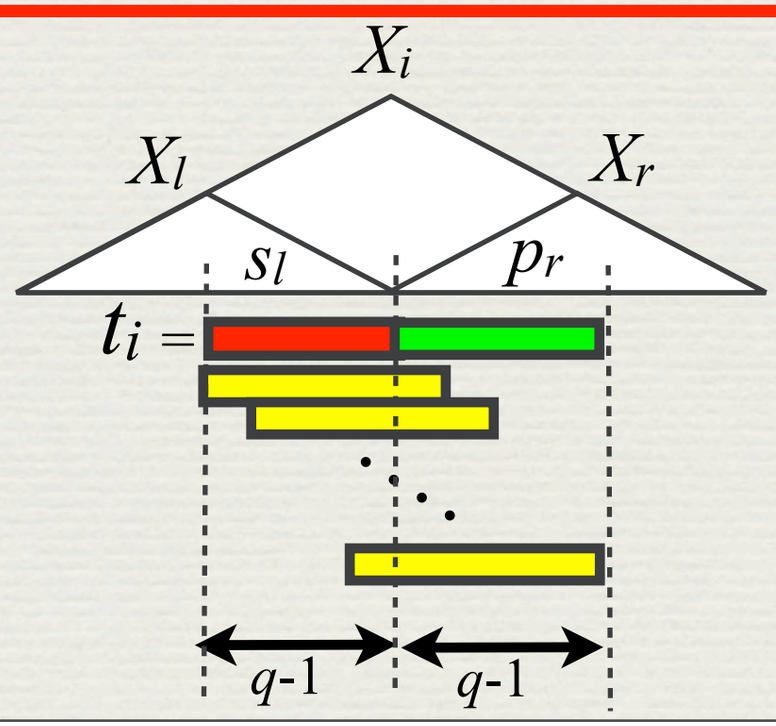


Our Strategy

- STEP 1. for all $X_i = X_l X_r$, compute $vOcc(X_i)$
- STEP 2. for all $X_i = X_l X_r$, compute decompressed prefix p_i and suffix s_i of length at most q-1, and $t_i = s_l p_r$
- STEP 3. solve the multi weighted q-gram frequencies for all t_i and $vOcc(X_i)$

STEP 1 and 2 can be computed in $O(qn)$ time and space by a simple dynamic programming

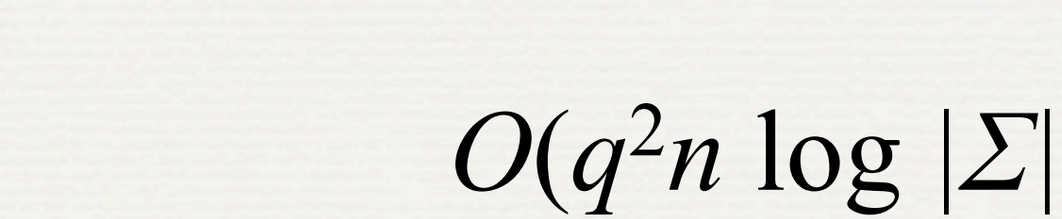
From now, we describes STEP 3



$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



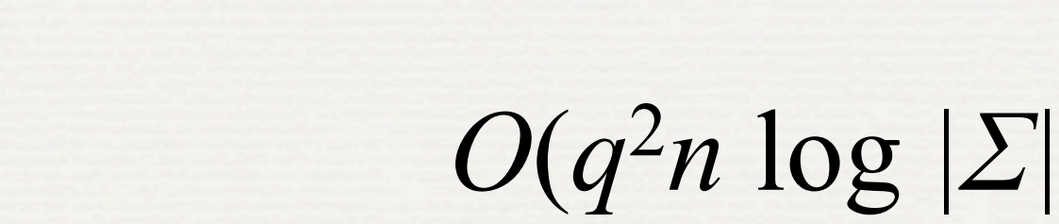
$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	0
aba	0
baa	0
⋮	⋮

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



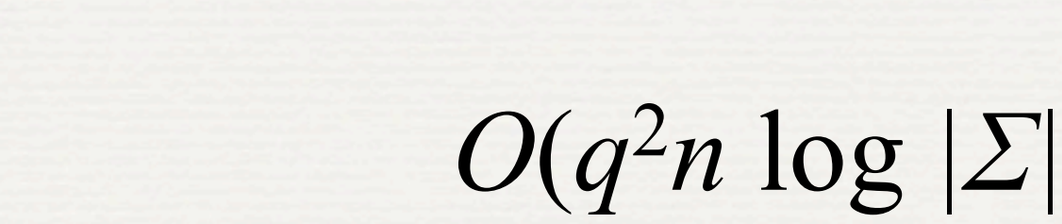
$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	0
aba	0
baa	0
⋮	⋮

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



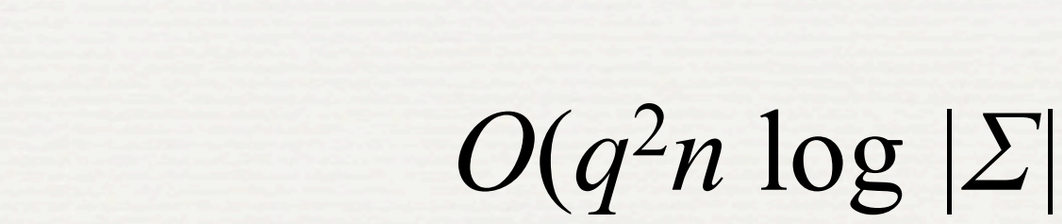
$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	4
aba	0
baa	0
⋮	⋮

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



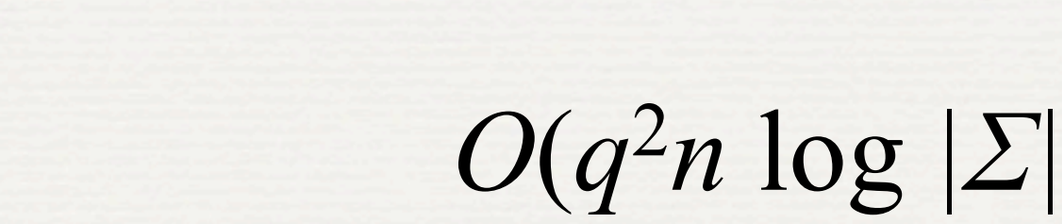
$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	4
aba	0
baa	0
⋮	⋮

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	4
aba	4
baa	0
⋮	⋮

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example

$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

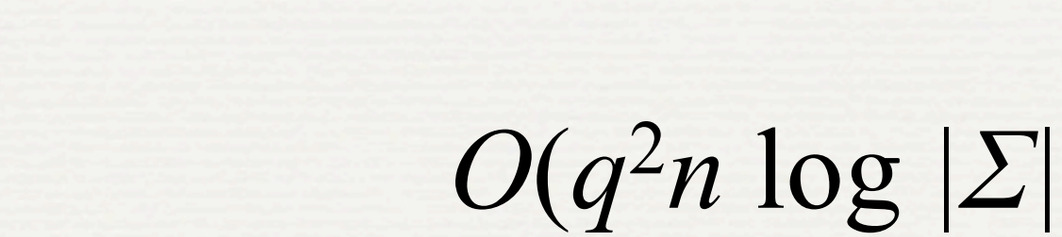
q-gram	Frequency
⋮	⋮
aab	4
aba	4
baa	0
⋮	⋮

- Each access to associative array: $O(q \log |\Sigma|)$ time
- Total: $O(q^2 n \log |\Sigma|)$ time, $O(q^2 n)$ space

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

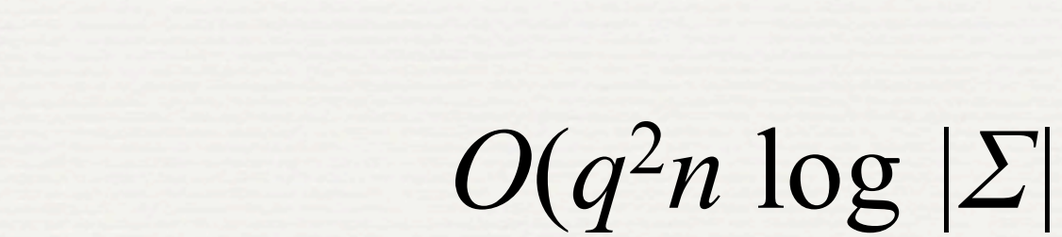
q-gram	Frequency
⋮	⋮
aab	4
aba	4
baa	0
⋮	⋮

- Each access to associative array: $O(q \log |\Sigma|)$ time
- Total: $O(q^2 n \log |\Sigma|)$ time, $O(q^2 n)$ space

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

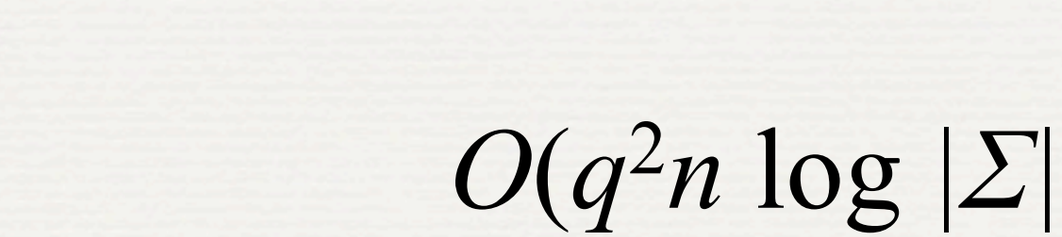
q-gram	Frequency
⋮	⋮
aab	4
aba	4
baa	3
⋮	⋮

- Each access to associative array: $O(q \log |\Sigma|)$ time
- Total: $O(q^2 n \log |\Sigma|)$ time, $O(q^2 n)$ space

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

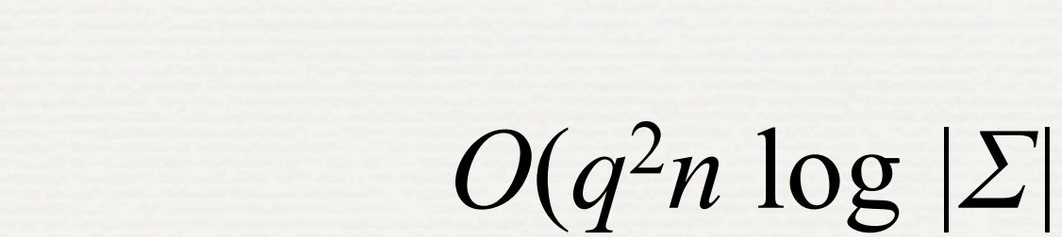
q-gram	Frequency
⋮	⋮
aab	4
aba	4
baa	3
⋮	⋮

- Each access to associative array: $O(q \log |\Sigma|)$ time
- Total: $O(q^2 n \log |\Sigma|)$ time, $O(q^2 n)$ space

$O(q^2 n \log |\Sigma|)$-time Algorithm

- Store frequency of q-grams in associative array
- For each q-gram which starts in X_l and ends in X_r, we increase its frequency by $vOcc(X_i)$

Example



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

q-gram	Frequency
⋮	⋮
aab	7
aba	4
baa	3
⋮	⋮

- Each access to associative array: $O(q \log |\Sigma|)$ time
- Total: $O(q^2 n \log |\Sigma|)$ time, $O(q^2 n)$ space

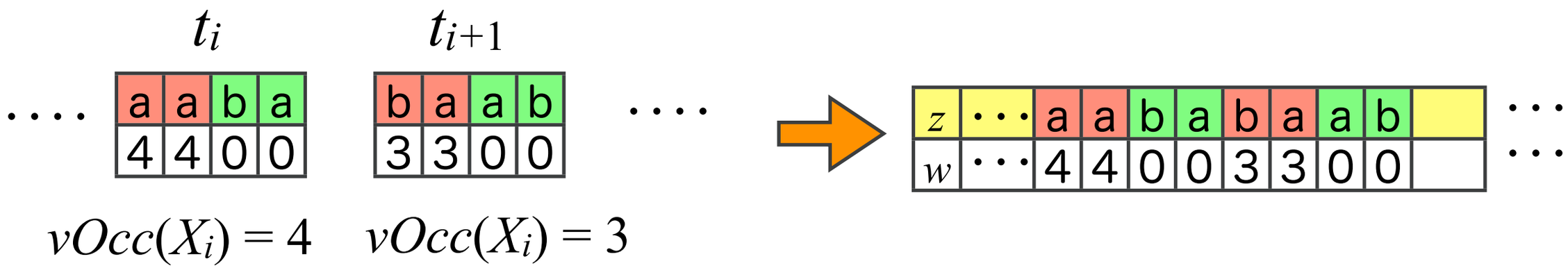
$O(qn)$-time Algorithm

Construct string z and weight array w, as follows

$z = t_1\ t_2\ t_3\ ....\ t_n$, where $|w_i| = |t_i|$,

$w = w_1\ w_2\ w_3\ ...\ w_n$

$$w_i[j] = \begin{cases} vOcc(X_i) & \text{if } j \leq |s_l| \\ 0 & \text{if } j > |s_l| \end{cases}$$



$vOcc(X_i) = 4$ $vOcc(X_i) = 3$

- For each q-gram $z[\,j\,..\,j+q\text{-}1]$, we count by $w[\,j\,]$

$O(qn)$-time Algorithm

- Construct Suffix Array and LCP Array of z
- Compute sum of w[SA[i]] in the interval where LCP[i] $\geqq q$

i	SA	LCP	w[SA[i]]	z[SA[i]]
0	16	0	1	aab
1	2	3	0	aababbaabbaabbaab
2	12	3	1	aabbaab
3	8	7	1	aabbaabbaab
4	17	1	0	ab
5	0	2	4	ababbaabbaabbaab
6	3	3	3	abbaabbaabbaab
7	13	2	0	abbaab
⋮				

$|Occ(T, \mathrm{aab})| = 3$

$|Occ(T, \mathrm{aba})| = 7$

$O(qn)$ time and space

Experiments

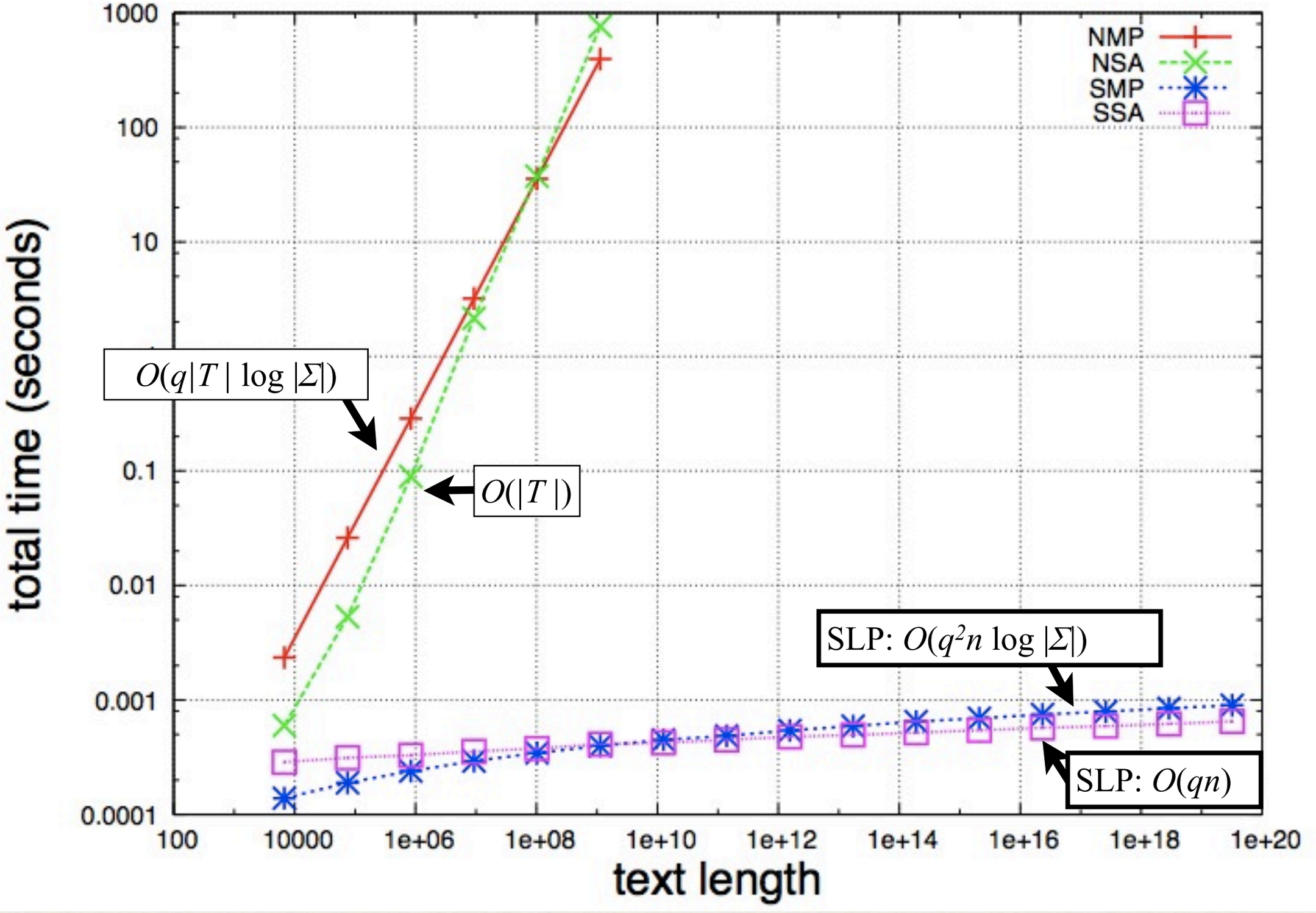
Computational Experiments

- Compared running time of 4 algorithms, after reading input into memory
  - Uncompressed String: $O(q|T| \log |\Sigma|)$
  - Uncompressed String: $O(|T|)$
  - SLP: $O(q^2 n \log |\Sigma|)$
  - SLP: $O(qn)$
- Implementation
  - language: C++
  - suffix array construction: libdivsufsort[1]
  - associative array: std::map

[1] http://code.google.com/p/libdivsufsort/

Experiment (Fibonacci Strings)

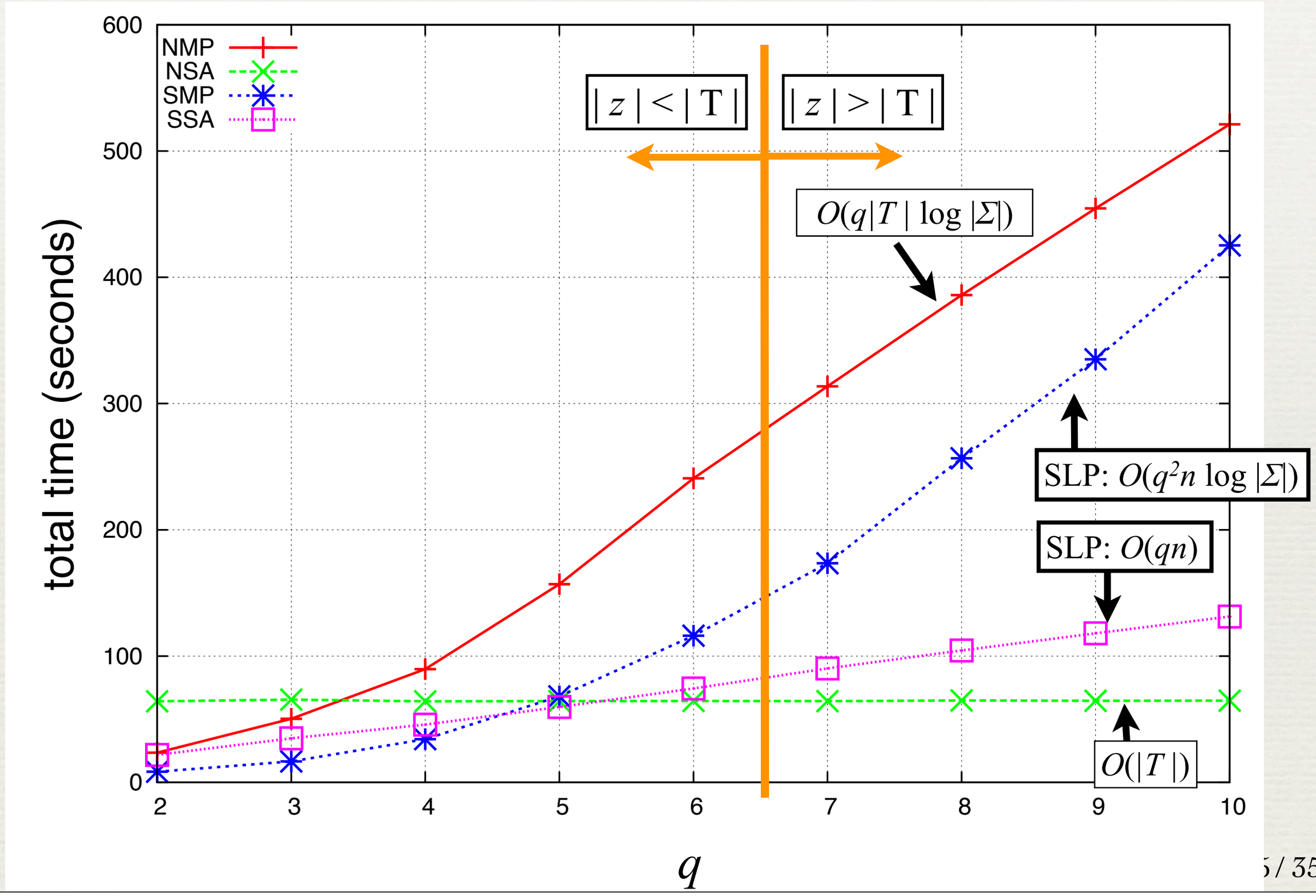
$q = 50$



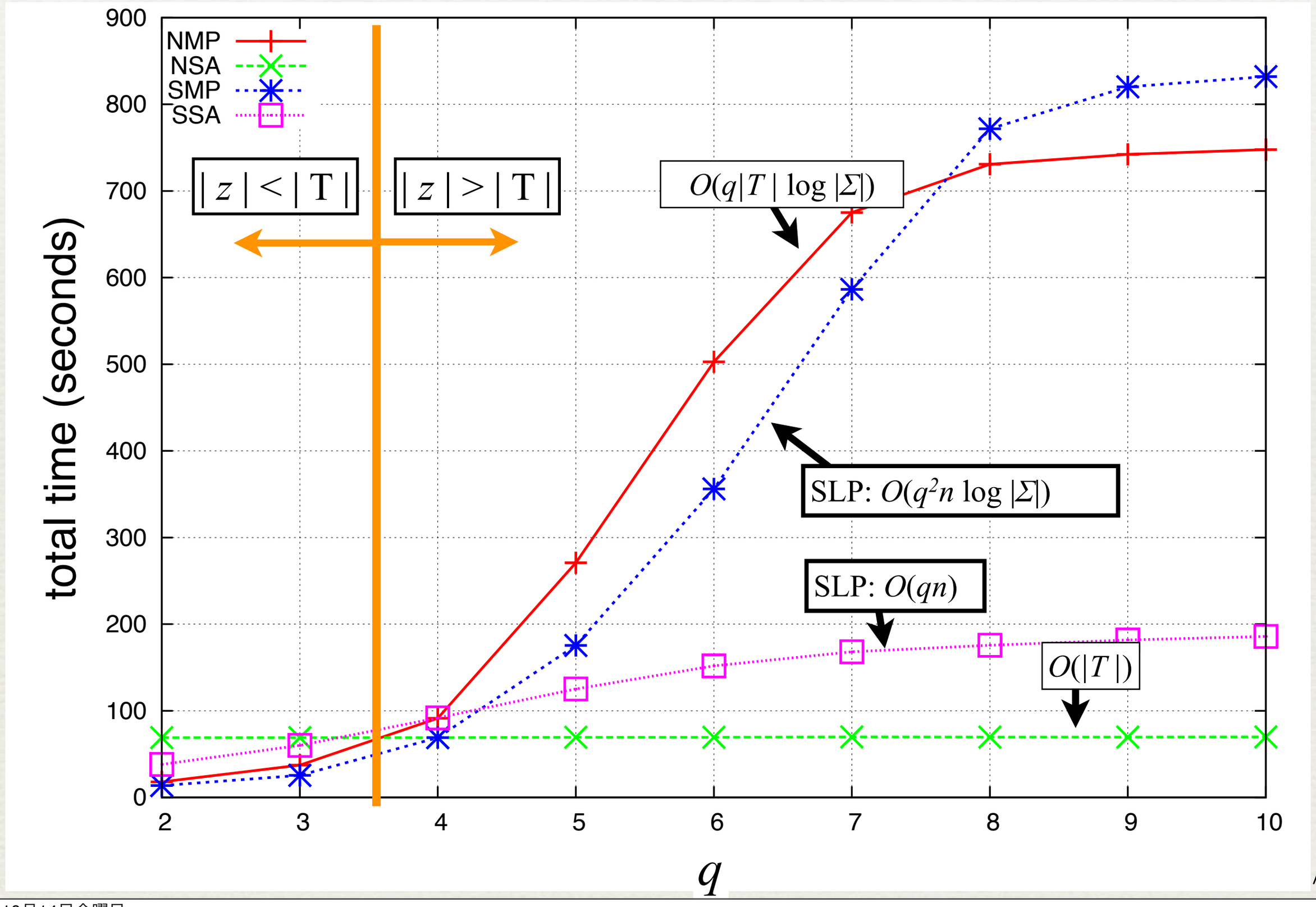
Pizza & Chili Corpus

- Data: XML, DNA, ENGLISH, and PROTEINS
  with size 200MB
- Input SLPs were generated by RE-PAIR **[Larsson & Moffat, 1999]**
- q is varied from 2 to 10

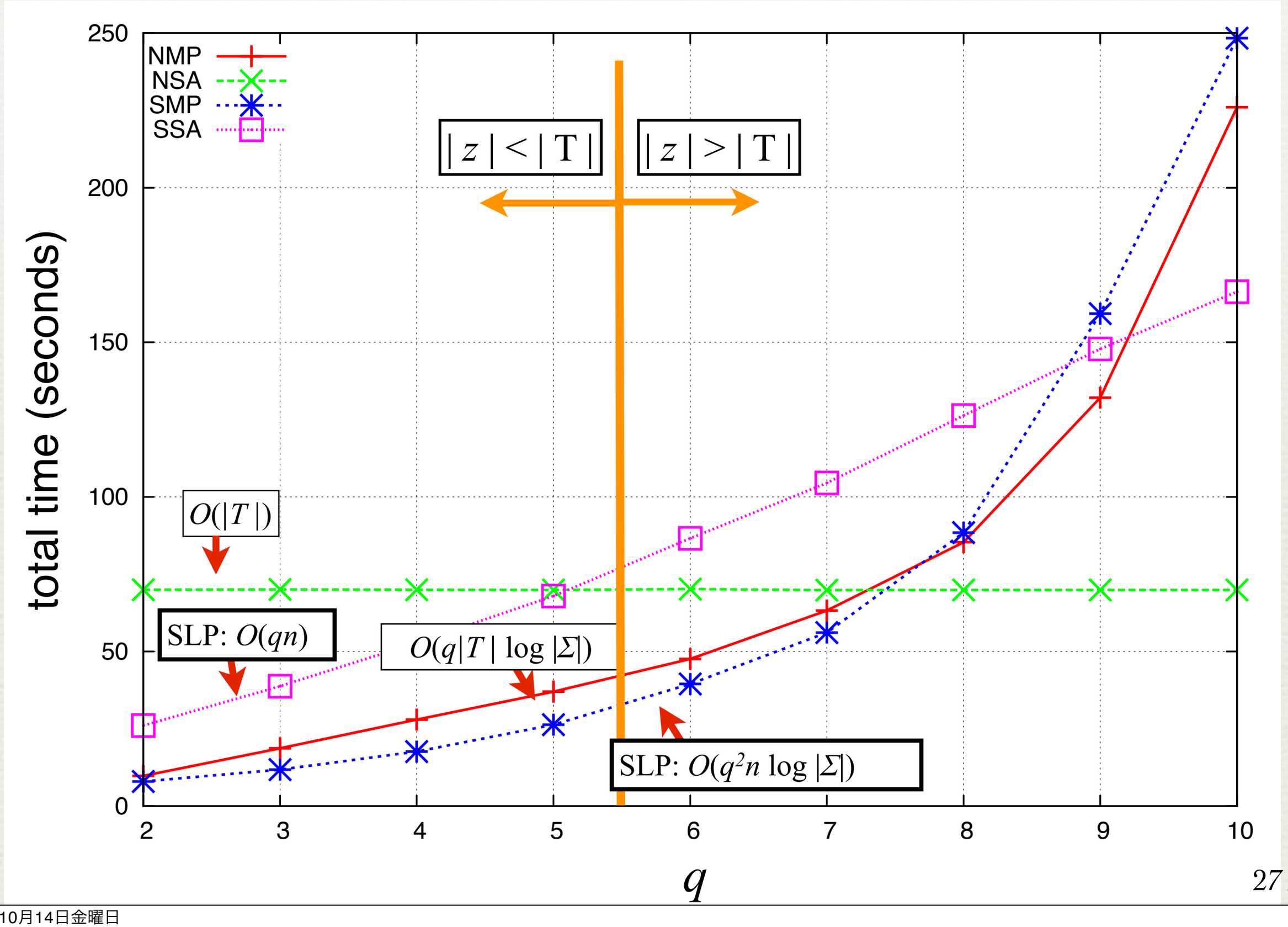
Experiment (ENGLISH 200MB)



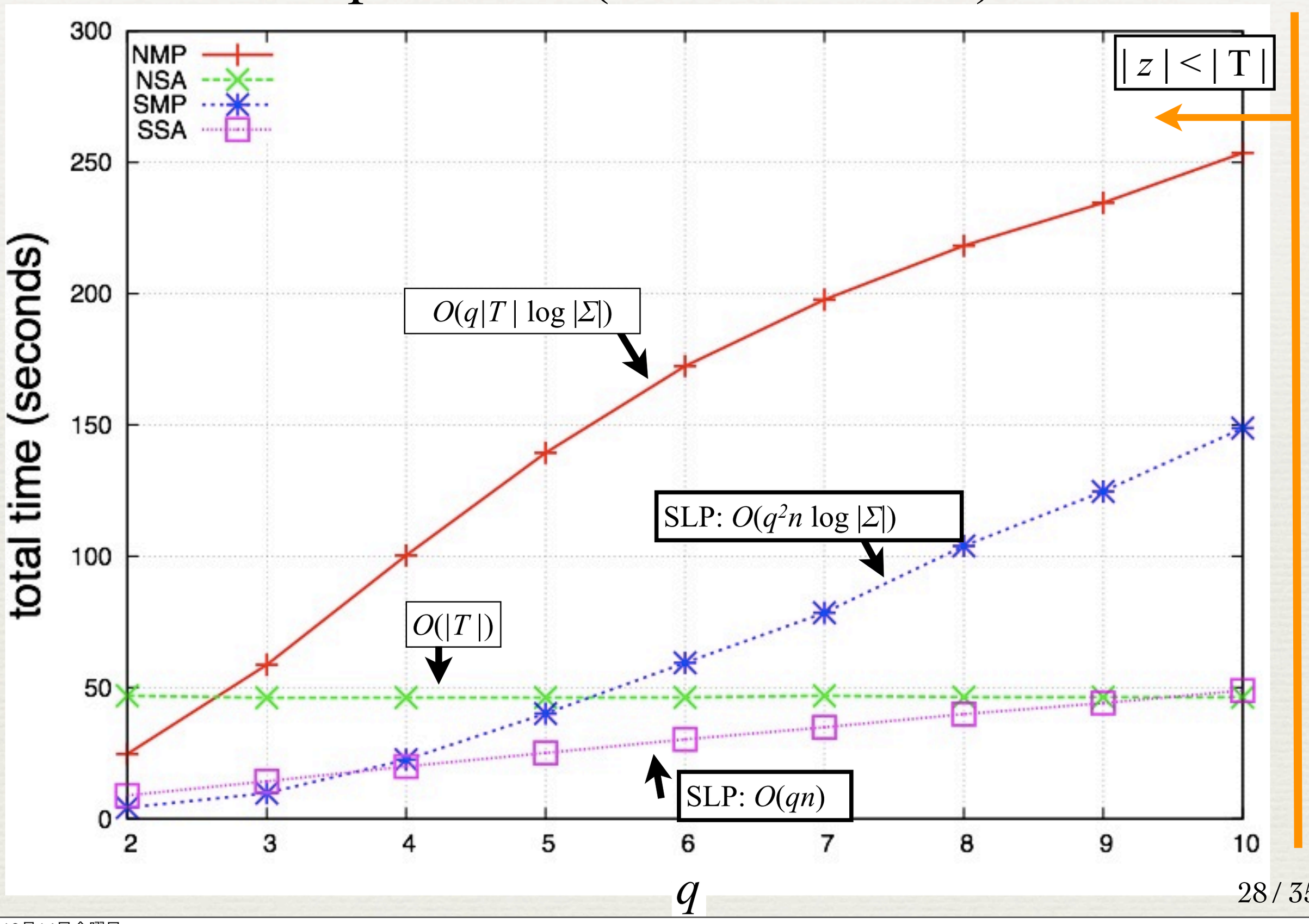
Experiment (PROTEINS 200MB)



Experiment (DNA 200MB)



Experiment (DBLP 200MB)



Summary

- We proposed new algorithms for computing q-gram frequencies on compressed string
- Proposed algorithms are simple, and **practically fast** when q is small

	uncompressed string	SLP (previous work)	SLP (Our Work)
q-gram Frequencies	$O(\|T\|)$	$O(\|\Sigma\|^q qn^2)$ [2]	$O(q^2 n \log \|\Sigma\|)$ $O(qn)$
q-gram Kernel	$O(\|T_1\| + \|T_2\|)$	None	$O(q(n_1 + n_2))$
Optimal q-gram Pattern Discovery	$O(M)$ [1]	None	$O(qN)$ [3]

[1] M: sum of data size

[2] **[Inenaga and Bannai, 2009]**

[3] N: sum of compressed data size

Thank you for listening