The magic of full-text (substring) indexing

Lecture 4.3

by Marina Barsky

Task: searching for a pattern (substring)

- Query: is *cob* in *cocoa*?

pattern

input string

Input string				
0	1	2	3	4
c	o	c	o	a

Scalability problem: very long strings

- Cocoa DNA: cocoacoacoacoacoacoa... up to 3,000,000,000 characters
- You want to know whether the *cob* substring is somewhere in this input string

aabccooccooaaacaccccaboccoaaoooocbacacccaccoocccoaooaccc
caooccacaoooaccacccccocaacacocoooacoaaccccoabacoccococaa
cobcooacooaoccoacaccaabcccocccocoocaaoacooccoacoccccaocba
aaccbcocboabaocacccccccooocbaoccbocacacccaccaoocccoccoccao
cccccccaacoccccoocaooccocaaoocccooccbocaccbaocoaccbcaoca
caaocccccoccacooooaooocaocbaccocaoboocbccaboocacaaacaao
bcacooccooocoboocacaocoabooaobaocbaccaaoobaaccbcccoaaca
ccccoaccbbcacaccoaaccccccocccoaobcaccooocoooocccoccccacco
ooocoaccacccoaccoccocoabacooaaccccococacooaoc…

Preprocessing: sort substrings

- In order to facilitate pattern search:
  - Generate every substring of the input string
  - Keep all substrings lexicographically sorted
- We can do a binary search on sorted substrings

- We call lexicographically sorted keys indexes

Input string					
0	1	2	3	4	5
c	o	c	o	a	$

sentinel

Recap: Suffix Array

- Take all *suffixes** of the input string, and sort them alphabetically (lexicographically)
- Then record a start position of each sorted suffix - to get a *suffix array SA*

Input string X					
0	1	2	3	4	5
c	o	c	o	a	$

*Substring $S_i = X[i \ldots N]$ is called a *suffix* of *X*
It starts at position *i* and runs till the last character of *X*

Suffix S_2 = '*coa$*'
Suffix S_4 = '*a$*'
Suffix S_0 = '*cocoa$*' = *X*
Suffix $S_6 = \varepsilon$

Suffix array of *cocoa$*

Input string					
0	1	2	3	4	5
c	o	c	o	a	$

- The collection of sorted suffixes becomes:

Suffix array (SA) index		
Row	Sorted suffixes	SA (suffix start)
0	$	5
1	a$	4
2	coa$	2
3	cocoa$	0
4	oa$	3
5	ocoa$	1

Only the last column is called a suffix array

Suffix array of *cocoa$*

Input string of length *N*					
0	1	2	3	4	5
c	o	c	o	a	$

- The collection of sorted suffixes becomes:

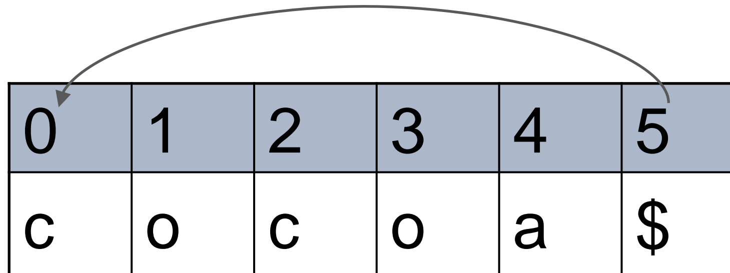
Suffix array (SA) index		
Row	Sorted suffixes	SA (suffix start)
0	$	5
1	a$	4
2	coa$	2
3	cocoa$	0
4	oa$	3
5	ocoa$	1

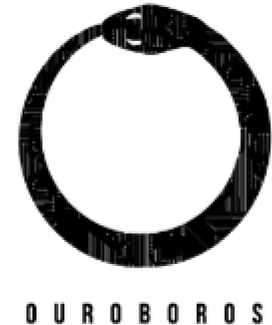
Now we can search for pattern '*cob*' in time O(M log *N*)

Better idea:

Burrows-Wheeler Transform

Circular strings





- Extract N different circular strings of length N:

[0] cocoa$

[1] ocoa$c

[2] coa$co

[3] oa$coc

[4] a$coco

[5] $cocoa

Sorted circular strings

Input string					
0	1	2	3	4	5
c	o	c	o	a	$

Row	Sorted circular strings	Sorted suffixes	SA
0	***$**cocoa*	*$*	5
1	***a$**coco*	*a$*	4
2	***coa$**co*	*coa$*	2
3	***cocoa$***	*cocoa$*	0
4	***oa$**coc*	*oa$*	3
5	***ocoa$**c*	*ocoa$*	1

Burrows-Wheeler Matrix

Magic: setup

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	*$*	*c*	*o*	*c*	*o*	*a*
1	*a*	*$*	*c*	*o*	*c*	*o*
2	*c*	*o*	*a*	*$*	*c*	*o*
3	*c*	*o*	*c*	*o*	*a*	*$*
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

Burrows-Wheeler Matrix

Input string					
0	1	2	3	4	5
c	o	c	o	a	$

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	*$*	*c*	*o*	*c*	*o*	*a*
1	*a*	*$*	*c*	*o*	*c*	*o*
2	*c*	*o*	*a*	*$*	*c*	*o*
3	*c*	*o*	*c*	*o*	*a*	*$*
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

Burrows-Wheeler Transform

BWT

The magic begins: BWT replaces original string

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	$	*c*	*o*	*c*	*o*	*a*
						o
						o
						$
4	*o*	*a*	*$*	*c*	*o*	*c*
5	*o*	*c*	*o*	*a*	*$*	*c*

If we store only BWT (column L), we do not actually need our original string anymore, because **we can always reconstruct the original string from the BWT string**

Characters with ranks

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	$1	c	o	c	o	a1
1	a1	$	c	o	c	o1
2	c1	o	a	$	c	o2
3	c2	o	c	o	a	$1
4	o1	a	$	c	o	c1
5	o2	c	o	a	$	c2

Characters with ranks

Burrows-Wheeler Matrix						
Row	F	1	2	3	4	L
0	$1	c	o	c	o	a1
1	a1	$	c	o	c	o1
2	c1	o	a	$	c	o2
3	c2	o	c	o	a	$1
4	o1	a	$	c	o	c1
5	o2	c	o	a	$	c2

LF mapping table

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

Reconstructing original string: 1/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxxxx$***

Reconstructing original string: 2/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxxxa$***

Reconstructing original string: 2/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxxxa$***

Reconstructing original string: 3/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxxoa$***

Reconstructing original string: 4/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxcoa$***

Reconstructing original string: 4/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xxcoa$***

Reconstructing original string: 5/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2
o	4 +1

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***xocoa$***

Reconstructing original string: 6/6

LF-mapping table	
Interval of all suffixes starting with:	begins at row (in suffix array):
$	0
a	1
c	2 +1
o	4

BWT	
Row	L
0	a1
1	o1
2	o2
3	$1
4	c1
5	c2

- Original string: ***cocoa$***

BWT is a self-index

- Having only a BWT string and a small LF-table, we do not need to store the original string anymore, as it can always be easily reconstructed whenever needed
- In a similar spirit we can use BWT for pattern search

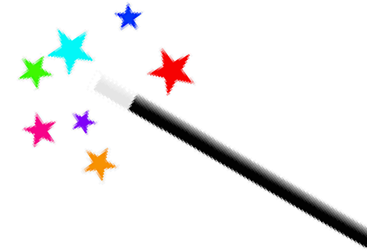
This makes it an index!

FM-index (Ferragina and Manzini, 2005) consists of **2 tables**

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Magic continues!



Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Pattern: *oco*

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Pattern: *oc**o***

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Pattern: *o**co***

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Pattern: ***oco***

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

LF-mapping table	
Interval of all suffixes starting with:	Rows (in suffix array):
$	0
a	1
c	2-3
o	4-5

Pattern: ***oco***

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

0	1	2	3	4	5
c	**o**	**c**	**o**	a	$

Found: ***oco*** at position 1

Pattern search in FM-index

BWT (only L-column)			
Row	F	L	SA
0	$1	a1	5
1	a1	o1	4
2	c1	o2	2
3	c2	$1	0
4	o1	c1	3
5	o2	c2	1

0	1	2	3	4	5
c	**o**	**c**	**o**	a	$

The search time is propoprtional to the length of the pattern and is independent of the input length N

Sorting suffixes remains a major scalability bottleneck for constructing FM indexes

Row	Sorted circular strings	Sorted suffixes
0	*$cocoa*	*$*
1	*a$coco*	*a$*
2	*coa$co*	*coa$*
3	*cocoa$*	*cocoa$*
4	*oa$coc*	*oa$*
5	*ocoa$c*	*ocoa$*

Our new external-memory algorithm: Suffix Rank

- fully sequential I/Os
- potential for parallelization on shared-nothing architectures
- scales to arbitrarily large inputs
- guaranteed running time: O(log N) scans over disk data
- simplicity

Suffix array construction based on Prefix Doubling

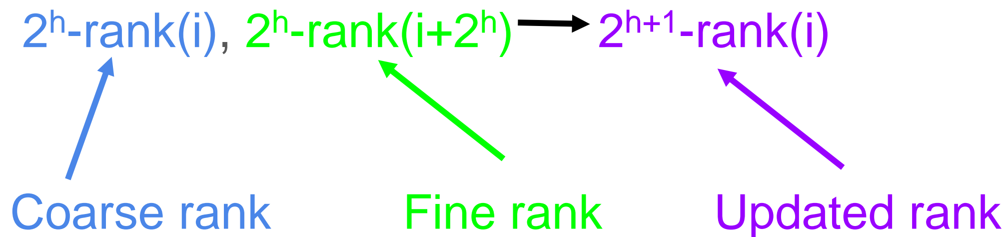
In-Memory Algorithm

- Introduced by Manber and Myers, 1993
- Improved by Larsson and Sadakane (qsufsort), 2007

Prefix doubling: idea

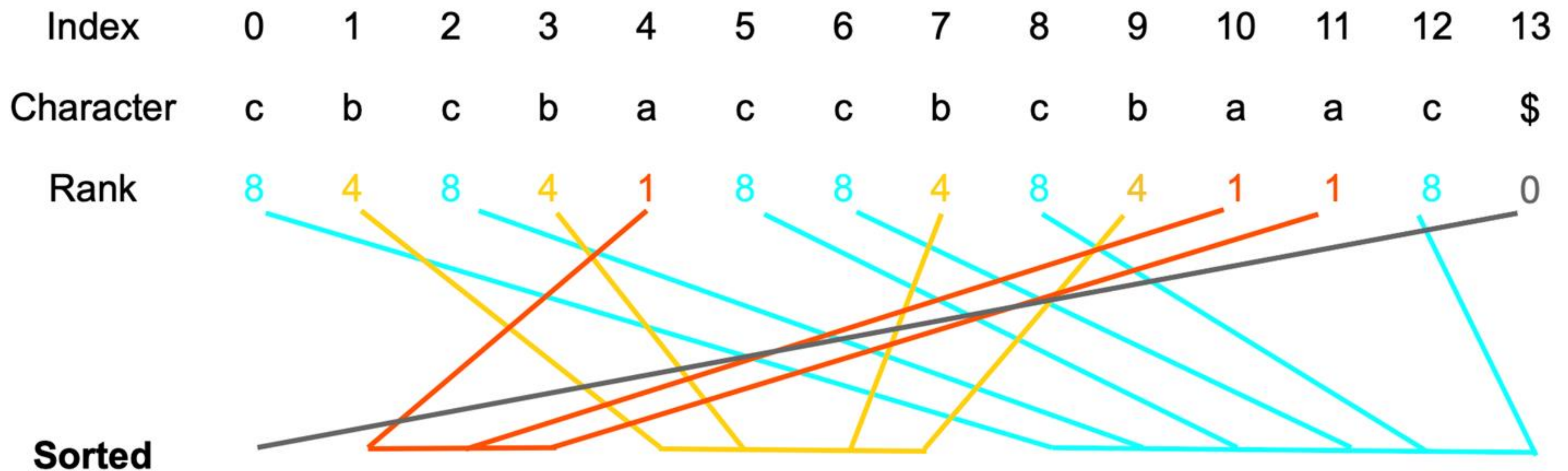
k-rank: position in suffix array based on first k characters

On iteration h, use 2^h-ranks to calculate 2^{h+1}-ranks



CAAAGTCC

qsufsort (Larsson and Sadakane, 2007)



Begin by assigning 1-ranks based on the first character of each suffix

Divide suffixes into groups by common rank

qsufsort (Larsson and Sadakane, 2007)

Produce 2-rank by refining 1-rank with the ranks of the next suffixes (which are already known)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Character	c	b	c	b	a	c	c	b	c	b	a	a	c	$
Rank	8	4	8	4	1 →	8	8	4	8	4	1 →	1 →	8	0

Sorted

Now we know that S_{10} is lexicographically smaller than S_4, and will be placed before it in SA

S_{10} gets it final rank: 1
S_4 and S_{11} both get the next rank 2

qsufsort (Larsson and Sadakane, 2007)

Produce 2-rank by refining 1-rank with the ranks of the next suffixes (which are already known)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Character	c	b	c	b	a	c	c	b	c	b	a	a	c	$
Rank	8	4	8	4	1	8	8	4	8	4	1	1	8	0

Sorted

After we found the order of S_i based on 2 first characters of each suffix, we can refine each 2-character prefix by the rank of the suffix at position $i+2^1$

Then – by the rank of suffix at position $i + 2^2$

In log *N* iterations every suffix will get its final rank: the position in the SA

Suffix rank: a new scalable algorithm for indexing large string collections
M. Barsky, J. Gabor, M. Consens, A.Thomo

External Memory: Suffix Rank

External Memory

GGTCTAGATGACACTTCAAAGTCCCTTCTTTCTTTCTAAAACTGTCC$

Divide string into chunks, each small enough to fit in RAM

GGTCTAGATGACACTT CAAAGTCCCTTCTTTC TTTCTAAAACTGTCC$

Chunk 1 Chunk 2 Chunk 3

Assign ranks **globally**, order suffixes **locally**

$: 0, A: 1, C: 12, G: 25, T: 31

Work independently with chunk 1

Char	Rank
$	0
A	1
C	12
G	25
T	31

G	G	T	C	T	A	G	A	T	G	A	C	A	C	T	T
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

… chunk 2

Char	Rank
$	0
A	1
C	12
G	25
T	31

C	A	A	A	G	T	C	C	C	T	T	C	T	T	T	C
12	1	1	1	25	31	12	12	12	31	31	12	31	31	31	12

… chunk 3

Char	Rank
$	0
A	1
C	5
G	25
T	31

T	T	T	C	T	A	A	A	A	C	T	G	T	C	C	$
31	31	31	12	31	1	1	1	1	12	31	25	31	12	12	0

Create Order Array for each chunk

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

Create Order Array for each chunk

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

5	7	10	12												

1

Create Order Array for each chunk

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

5	7	10	12	3	11	13									

1

12

Create Order Array for each chunk

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

5	7	10	12	3	11	13	0	1	6	9					

1 12 25

Create Order Array for each chunk

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31

5	7	10	12	3	11	13	0	1	6	9	2	4	8	14	15

1 12 25 31

Order array for chunk 1: suffixes sorted by the first character

Do the same for the other chunks

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	1	1	1	25	31	12	12	12	31	31	12	31	31	31	12

1	2	3	0	6	7	8	11	15	4	5	9	10	12	13	14

1 12 25 31

Order array for chunk 2: suffixes sorted by the first character

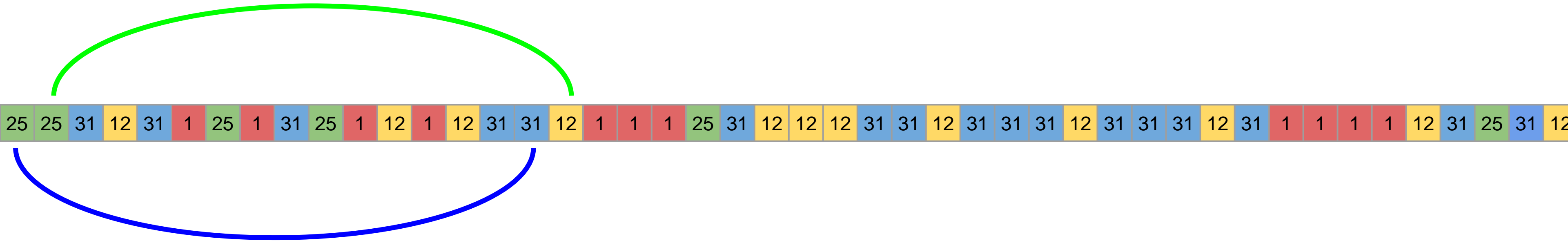
Do the same for the other chunks

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
31	31	31	12	31	1	1	1	1	12	31	25	31	12	12	0

15	5	6	7	8	3	9	13	14	11	0	1	2	4	10	12

0 1 12 25 31

Order array for chunk 3: suffixes sorted by the first character



Refine ranks

Rearrange local Suffix Array and write counts to disk

					5		7			10		12			
					1		1			1		1			
					25		31			12		12			

10	12	5	7												

To Disk

1 12 x2

1 25 x1

1 31 x1

Write counts to disk

			3								11		13		
			12								12		12		
			31								1		31		

				3	11	13									

To Disk

1 12 x2

1 25 x1

1 31 x1

Write counts to disk

		2		4				8						14	15
		31		31				31						31	31
		12		1				25						31	12

											4	2	15	8	14

To Disk

1 12 x2

1 25 x1

1 31 x1

12 1 x1

12 31 x2

25 1 x2

25 25 x1

25 31 x1

31 1 x1

31 12 x2

31 25 x1

31 31 x1

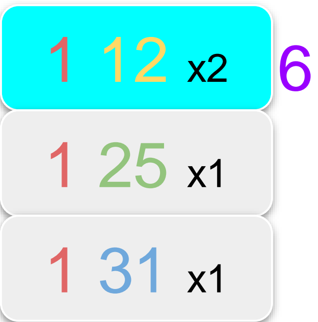
Merge and resolve the global rank of each suffix

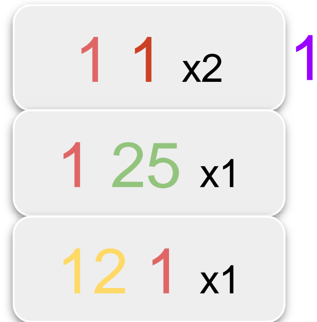
Chunk 1	Chunk 2	Chunk 3
1 12 x2	1 1 x2	1 1 x3
1 25 x1	1 25 x1	1 12 x1
1 31 x1	12 1 x1	12 0 x1

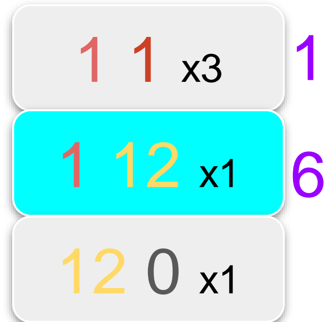
Merge and resolve

Chunk 1	Chunk 2		Chunk 3	
1 12 x2	1 1 x2	1	1 1 x3	1
1 25 x1	1 25 x1		1 12 x1	
1 31 x1	12 1 x1		12 0 x1	

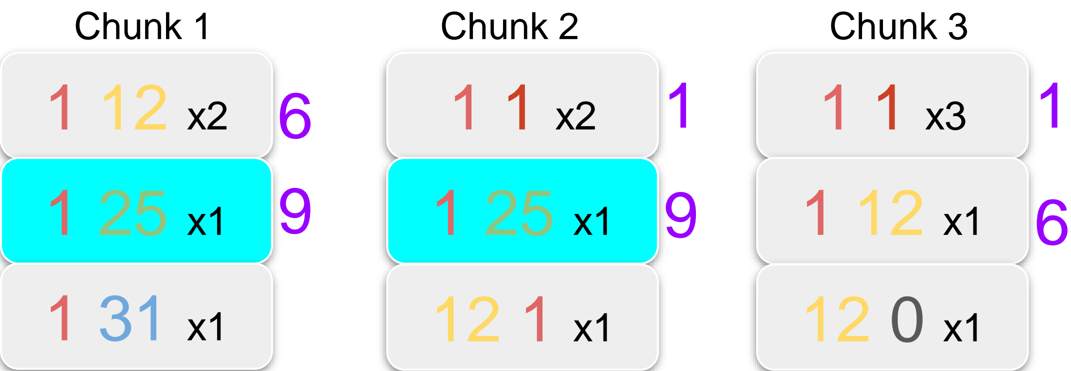
Merge and resolve







Merge and resolve



...

Update ranks in each chunk with global values

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31
25	31	12	31	1	25	1	31	25	1	12	1	12	31	31	12

10	12	5	7	11	3	13	6	9	0	1	4	2	15	8	14

6	9	11*	13	18	25	27*	28	31	33	40	42

Update

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	1	12	1	12	31	31
25	31	12	31	1	25	1	31	25	1	12	1	12	31	31	12

10	12	5	7	11	3	13	6	9	0	1	4	2	15	8	14

6	9	11*	13	18	25	27*	28	31	33	40	42

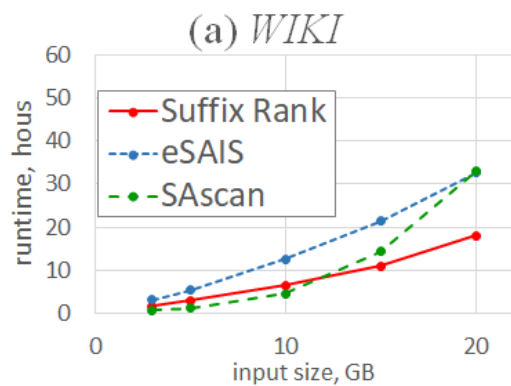
Update

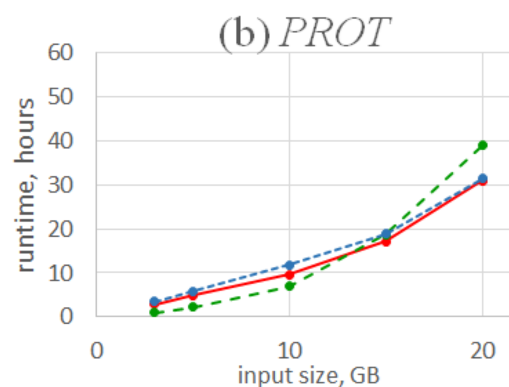
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
25	25	31	12	31	1	25	1	31	25	6	12	1	12	31	31
25	31	12	31	1	25	1	31	25	1	12	1	12	31	31	12

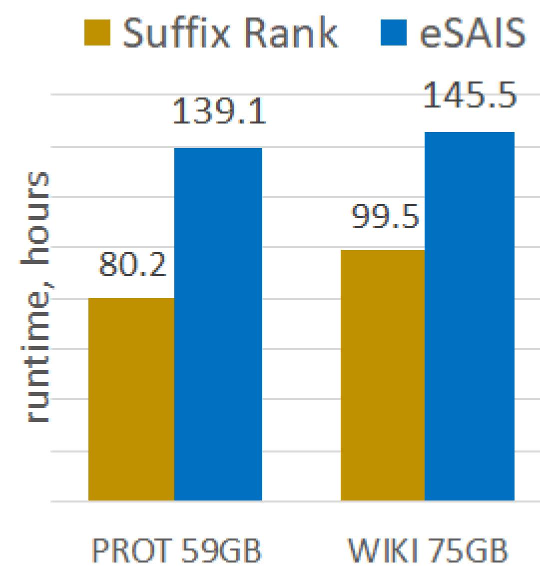
10	12	5	7	11	3	13	6	9	0	1	4	2	15	8	14

6	9	11*	13	18	25	27*	28	31	33	40	42

Experiments: HDD







Experiments: SSD