Full-text indexing : The Suffix Array

UNIVERSITY OF MARYLAND

Slides marked with */+ adopted from Ben Langmead/Carl Kingsford, respectively

Maintain T as part of
the index

*

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
0	cattcat$
1	attcat$
2	ttcat$
3	tcat$
4	cat$
5	at$
6	t$
7	$

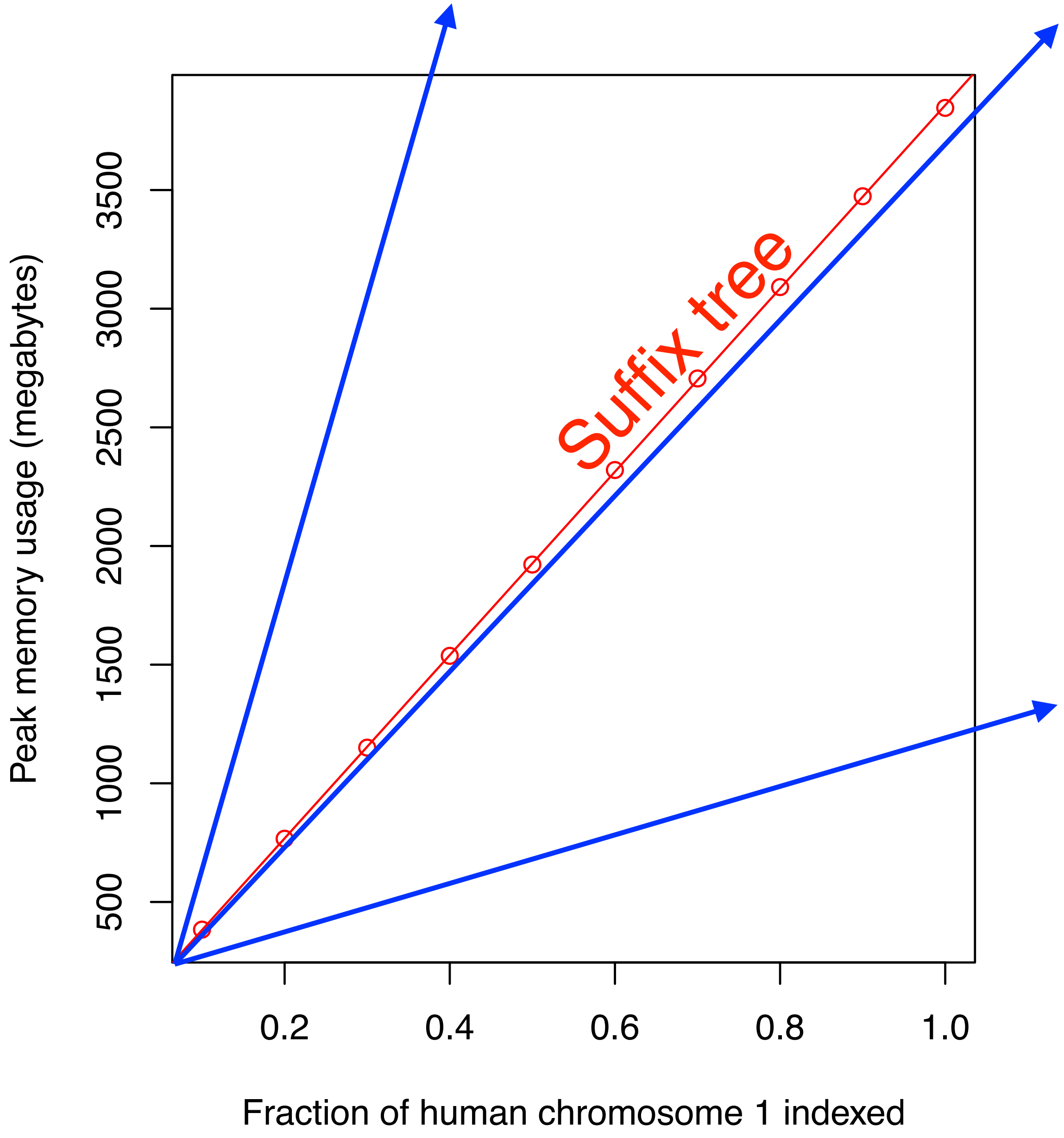
sort the suffixes alphabetically →

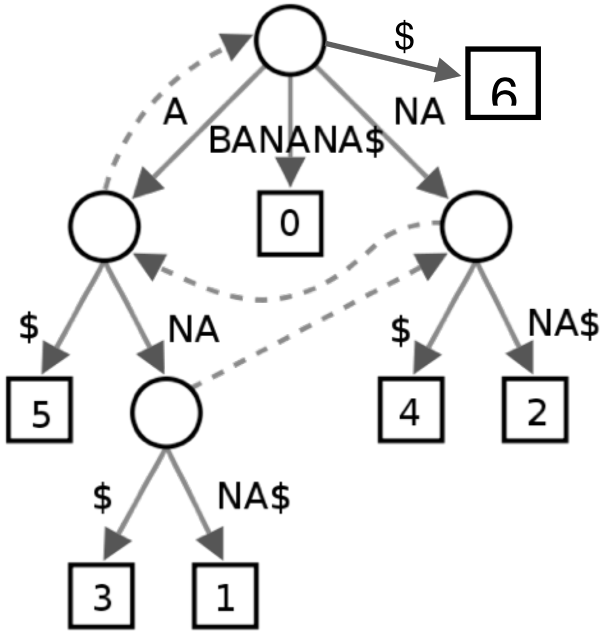
the indices just "come along for the ride"

7	$
5	at$
1	attcat$
4	cat$
0	cattcat$
6	t$
3	tcat$
2	ttcat$

Suffix array

O(m) space, like suffix tree   Is “constant factor” worse, better, same?



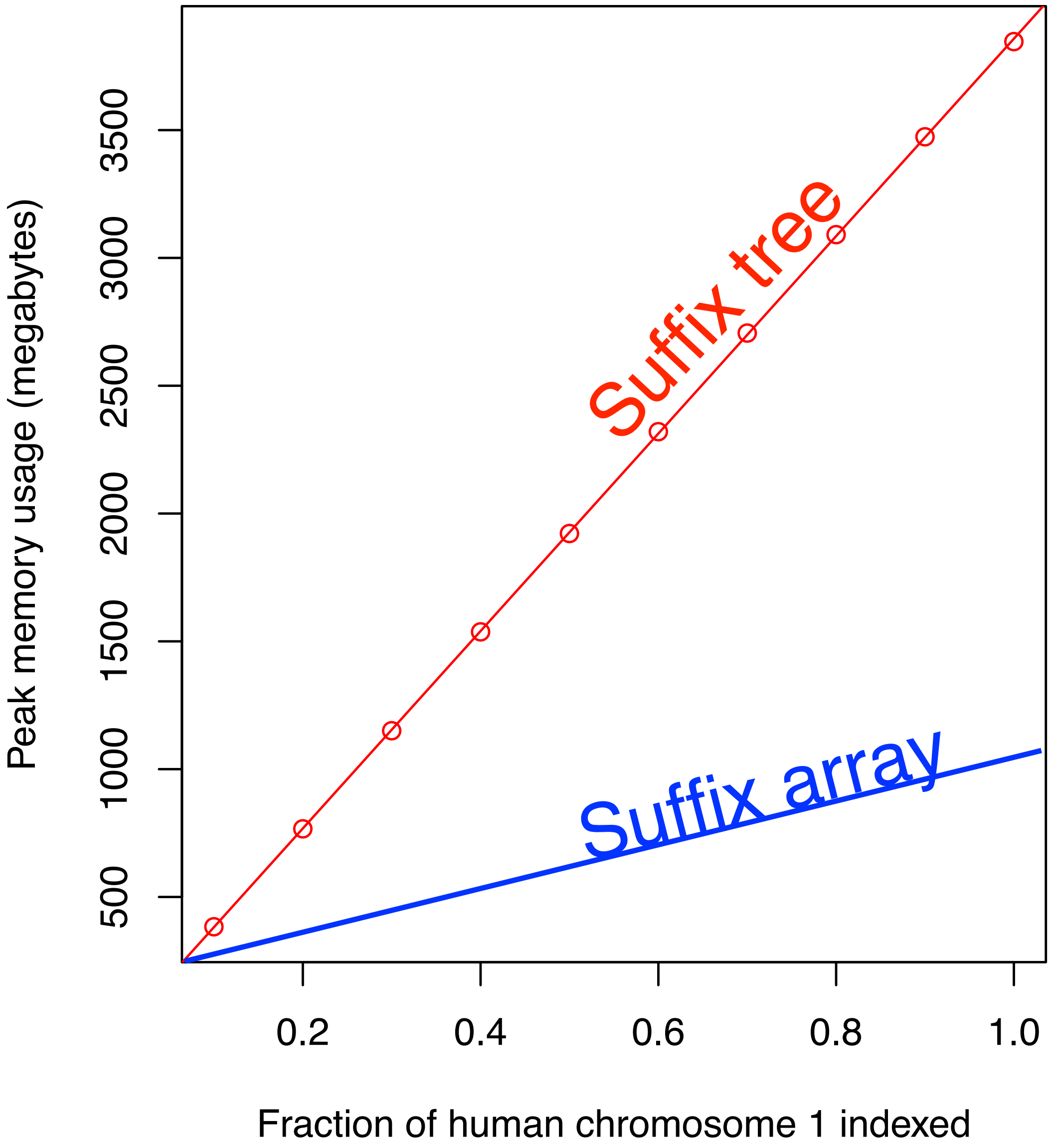


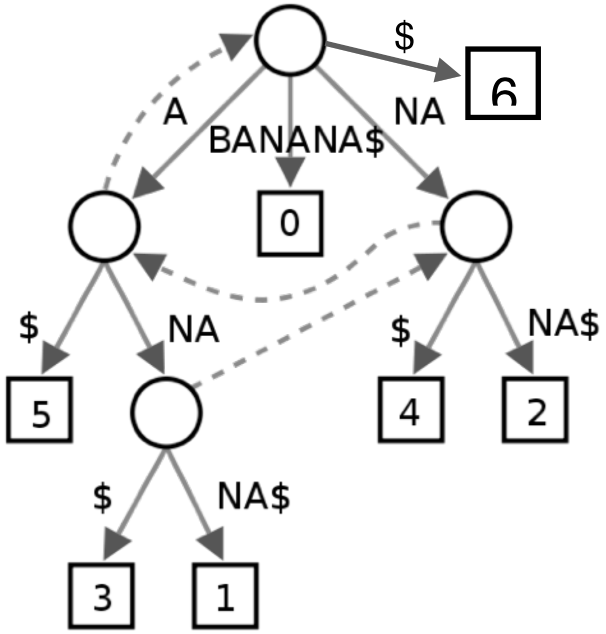
6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

*

Suffix array

32-bit integers sufficient for human genome, so fits in
~4 bytes/base × 3 billion bases ≈ 12 GB. Suffix tree is >45 GB.





6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

*

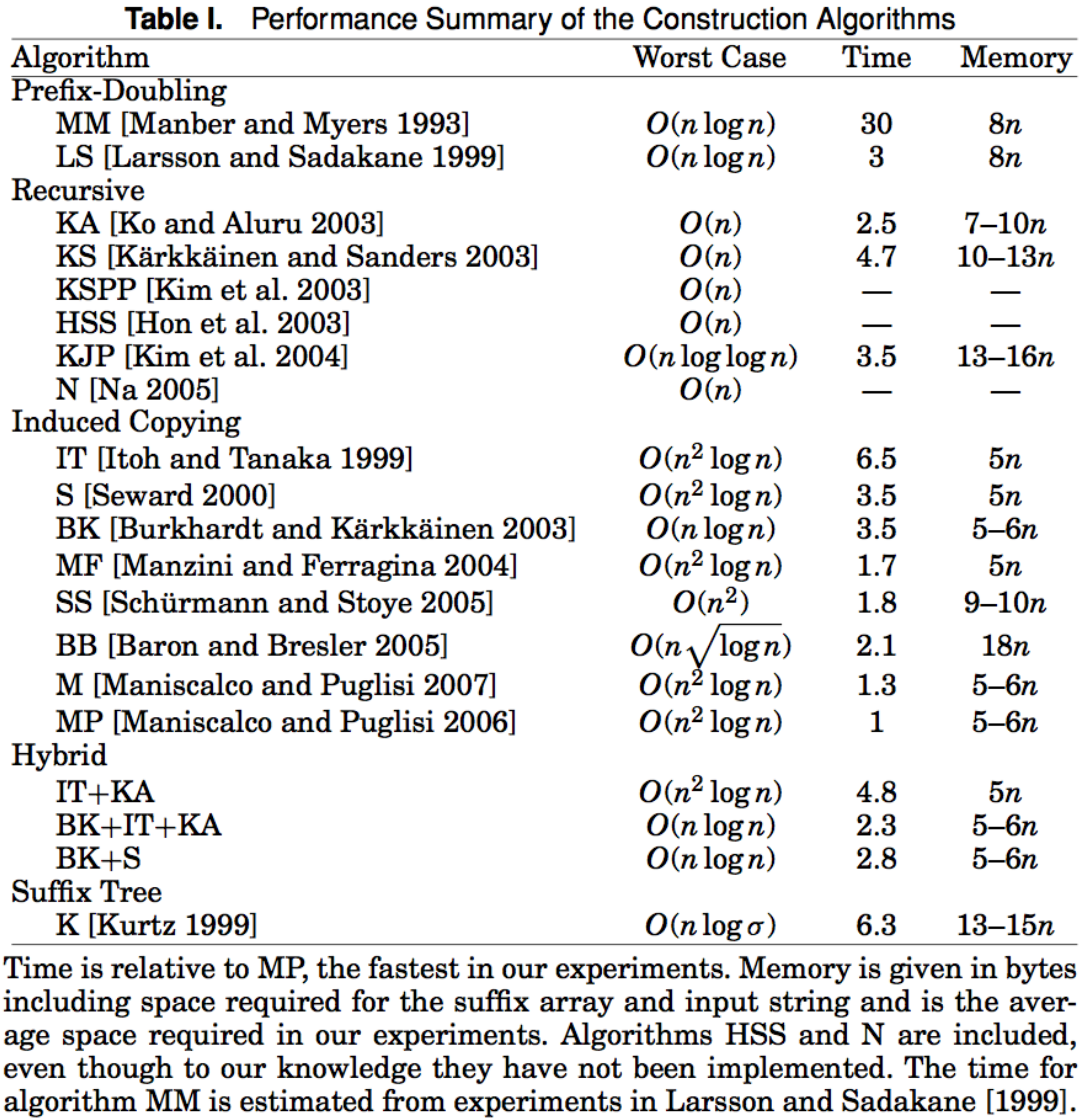


Table I. Performance Summary of the Construction Algorithms

Algorithm	Worst Case	Time	Memory
Prefix-Doubling			
MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive			
KA [Ko and Aluru 2003]	$O(n)$	2.5	7–10n
KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	10–13n
KSPP [Kim et al. 2003]	$O(n)$	—	—
HSS [Hon et al. 2003]	$O(n)$	—	—
KJP [Kim et al. 2004]	$O(n \log \log n)$	3.5	13–16n
N [Na 2005]	$O(n)$	—	—
Induced Copying			
IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	5–6n
MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	9–10n
BB [Baron and Bresler 2005]	$O(n \sqrt{\log n})$	2.1	$18n$
M [Maniscalco and Puglisi 2007]	$O(n^2 \log n)$	1.3	5–6n
MP [Maniscalco and Puglisi 2006]	$O(n^2 \log n)$	1	5–6n
Hybrid			
IT+KA	$O(n^2 \log n)$	4.8	$5n$
BK+IT+KA	$O(n \log n)$	2.3	5–6n
BK+S	$O(n \log n)$	2.8	5–6n
Suffix Tree			
K [Kurtz 1999]	$O(n \log \sigma)$	6.3	13–15n

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].

Puglisi, Smyth, Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

*

*

Pay attention to this observation!

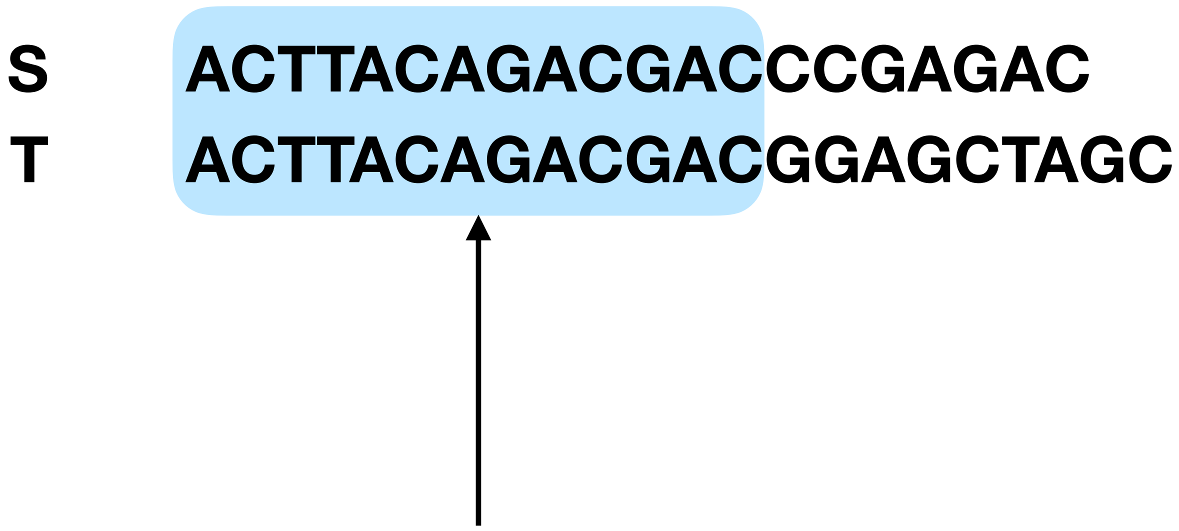
Almost all full-text indices (definitely the ones we will learn about), work based on the observation that ***every substring of T is a prefix of some suffix of T****.* These indices then focus on how to *organize* suffixes of T in a manner amenable to efficient search.

*

*

Longest Common Prefix

The longest common prefix of two strings s,t is simply the length of the prefix they share prior to the first difference (or the termination of either string).



S ACTTACAGACGACCCGAGAC

T ACTTACAGACGACGGAGCTAGC

LCP(S,T) = ACTTACAGACGAC

|LCP(S,T)| = 13

Below, to avoid extra notation, we will use LCP(S,T) as shorthand for |LCP(S,T)|

*

worst case still O(n log m), but we're closer.

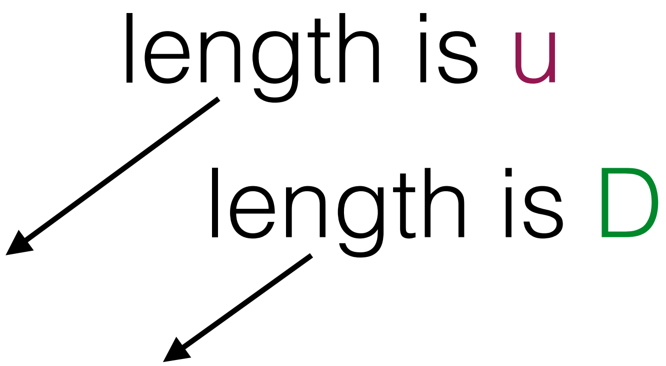
worst case example
$S=ac_{M-2}b$, P=c

*

Imagine we had pre-computed LCP(i,j) for all suffixes i and j in the original text T.

worst case still O(n log m), but we're closer.

Assume, wlog, that
D = LCP(SA[l], SA[c]) ≥ D’ = LCP(SA[c], SA[r])
otherwise there are symmetric cases.



length is u

length is D

key: u has already been computed by previous iterations, and D can be looked-up in constant time

*

or, if D’ = LCP(P, SA[r]) is larger, 3 symmetric cases.

*

In this case, we compute LCP(P[u:], SA[c][u:]).
c becomes our new l,
and now we know the new LCP(P, SA[l]), b/c we just computed it!

*

In this case, we compute
LCP(P[u:], SA[c][u:]).
c becomes our new r,
and now we know the new
LCP(P, SA[r]), b/c we just
computed it!

*

*

*

Sketch of Running Time

Thm. *Given the LCP(X,Y) values, searching for a string P in a suffix array of length m now takes O(|P| + log m) time.*

In case 1 & 2, we make O(1) comparisons and bisect left or right — there are at most O(log m) bisections.

In case 3 we try to match characters starting at some offset between SA[c] and P. If they match, those characters will never be compared again, so there are at most O(|P|) such comparisons.

Mismatching characters may be compared more than once.

But there can be only 1 mismatch / bisection. There are O(log *m*) bisections, so there are at most O(log *m*) mismatches.

∴Total # of comparisons = O(|P| + log *m*).

□

How to pre-compute LCP

- To perform this “efficient” search, we must be able to look up LCP(SA[c], SA[l]) and LCP(SA[c], SA[r]).
- How can we pre-compute this information *efficiently?*
  - Which LCP values do we need (*hint: not all of them)*?
  - Given LCP for left and right sub-interval of a search, how can we compute LCP for the containing interval?

*

*

*

*

*

```
$ len
```

*

$

*

$

*

$

*

$ t

- suf

*

:W

$

NOTE: These arrays are “shifted” by 1 — the value in LCP_LC corresponding to (0, 1, 2) is at LCP_LC[0], not LCP_LC[1]. So, to look up LCP(SA[l], SA[c]) we look at LCP_LC[c-1]

*

:W

Can be done in:

$

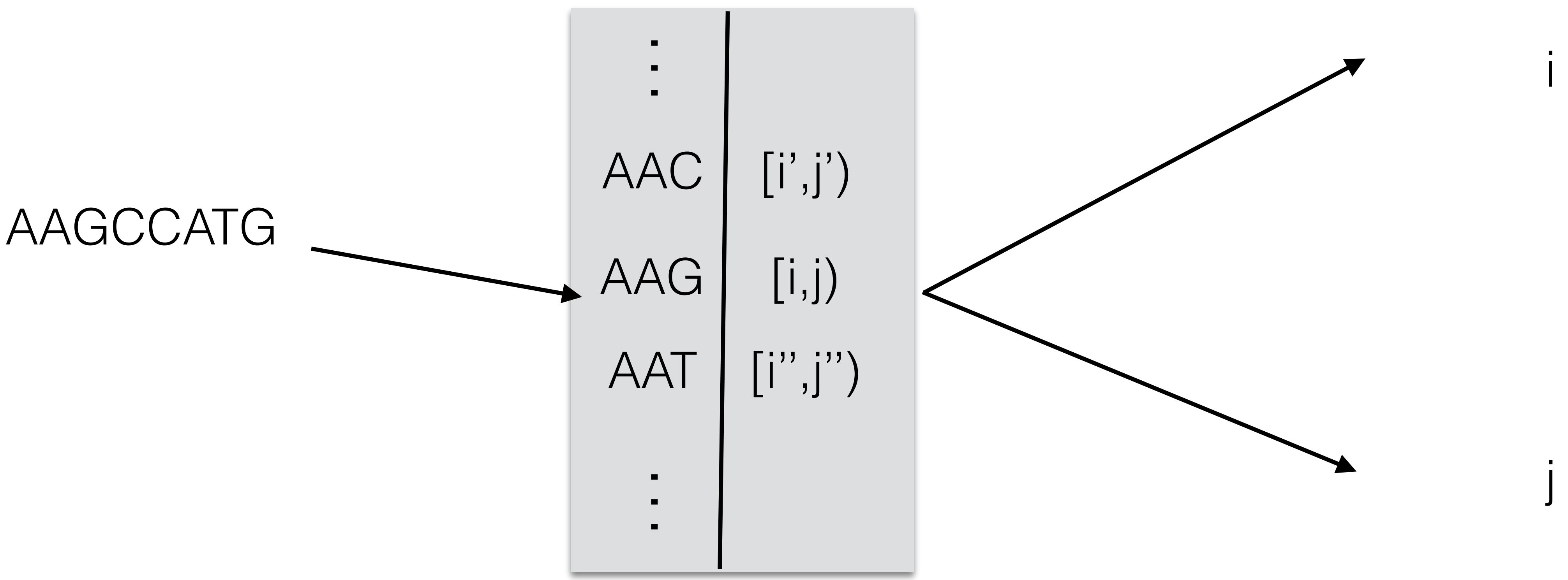
NOTE: These arrays are “shifted” by 1 — the value in LCP_LC corresponding to (0, 1, 2) is at LCP_LC[0], not LCP_LC[1]. So, to look up LCP(SA[l], SA[c]) we look at LCP_LC[c-1]

*

*

Another “practical” speedup

- Imagine you will never search for patterns of length < k (e.g. 4-mers are non-informative in any moderately-sized genome)
- Consider the following “enhanced” suffix array:
  - Build a hash-table from k-mers to suffix array intervals. Now, any pattern of length k’ > k must start with some hashed prefix of length k. Generally, the interval that needs to be searched is **much** smaller



AAGCCATG

⋮

AAC [i’,j’)

AAG [i,j)

AAT [i’’,j’’)

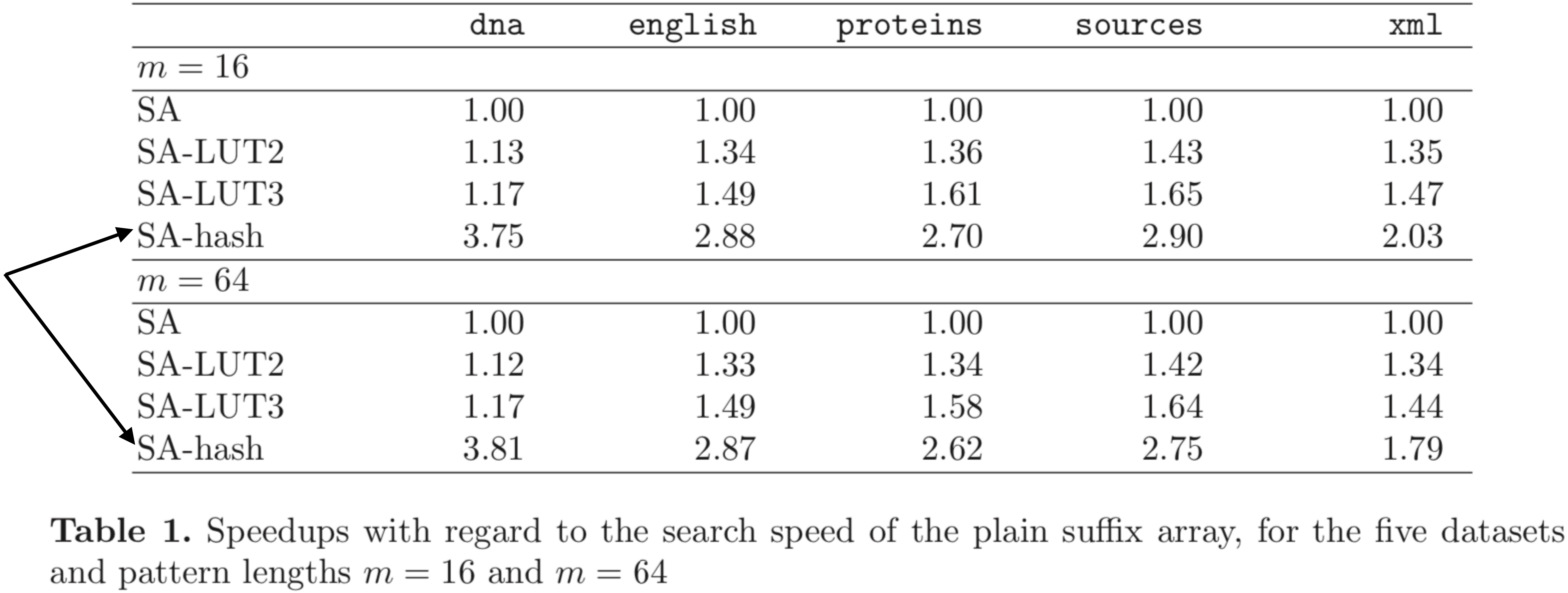
⋮

i

j

Now, you only need to search the interval [i,j) — O(n * log(j-i)) time

Can provide considerable speedup



	dna	english	proteins	sources	xml
$m = 16$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.13	1.34	1.36	1.43	1.35
SA-LUT3	1.17	1.49	1.61	1.65	1.47
SA-hash	3.75	2.88	2.70	2.90	2.03
$m = 64$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.12	1.33	1.34	1.42	1.34
SA-LUT3	1.17	1.49	1.58	1.64	1.44
SA-hash	3.81	2.87	2.62	2.75	1.79

Table 1. Speedups with regard to the search speed of the plain suffix array, for the five datasets and pattern lengths $m = 16$ and $m = 64$

k=12
Linear
probing
Hash at
α=0.5

Some other clever ideas[#]:

- Use a k-ary (B-tree) layout
- Use a lookup table where keys are concatenated Huffman codes of fixed bit length
- Use alternative strategy (doubling/galloping) to find the right SA boundary

“Two Simple Full-Text Indexes Based on the Suffix Array”, Szymon Grabowski and Marcin Raniszewski

[#] Kowalski, Tomasz, et al. "Suffix arrays with a twist." *arXiv preprint arXiv:1607.08176* (2016).

*

Suffix array: sorting suffixes

Another idea: Use a sort algorithm that's aware that the items being sorted are all suffixes of the same string

Original suffix array paper suggested an O(m log m) algorithm

Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.

Other popular O(m log m) algorithms have been suggested

Larsson NJ, Sadakane K. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

More recently O(m) algorithms have been demonstrated!

Kärkkäinen J, Sanders P. "Simple linear work suffix array construction." Automata, Languages and Programming (2003): 187-187.

Ko P, Aluru S. "Space efficient linear time construction of suffix arrays." Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2003.

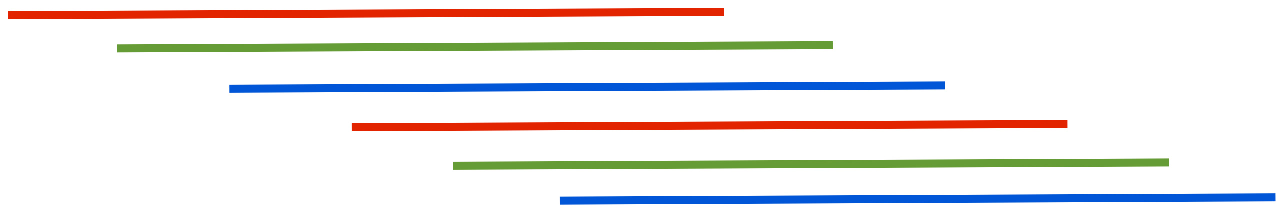
The Skew Algorithm (aka DC3)

Kärkkäinen & Sanders, 2003

- **Main idea: Divide suffixes into 3 groups:**
  - Those starting at positions i=0,3,6,9,.... (i mod 3 = 0)
  - Those starting at positions 1,4,7,10,... (i mod 3 = 1)
  - Those starting at positions 2,5,8,11,... (i mod 3 = 2)

- For simplicity, assume text length is a multiple of 3 after padding with a special character.

$$\begin{array}{lccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \mathbf{12} \\ T[0,n] = & \mathtt{y} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{o} & \mathbf{0} \end{array}$$



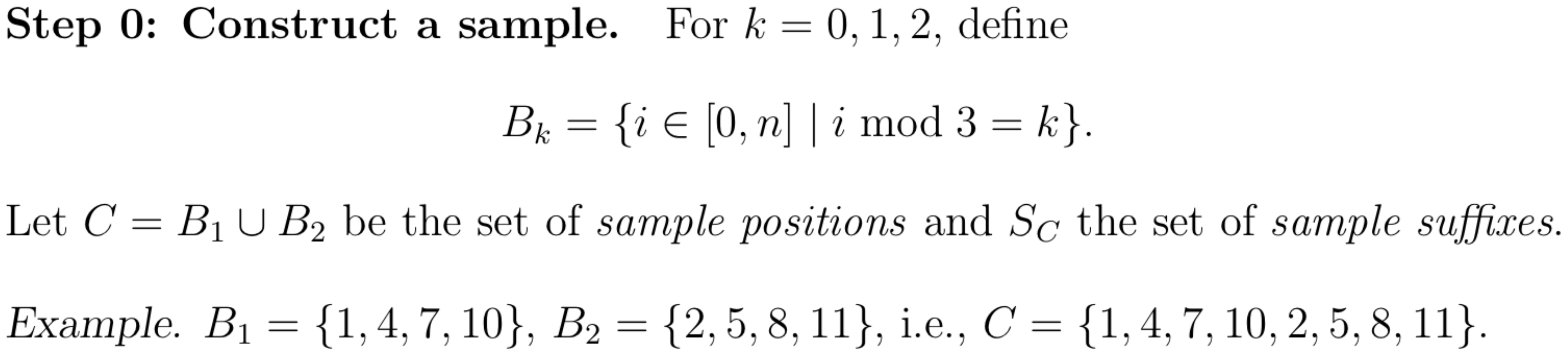
$$SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$$

Basic Outline:

- Recursively handle suffixes from the i mod 3 = 1 and i mod 3 = 2 groups.
- Merge the i mod 3 = 0 group at the end.

Step 0 — Constructing a sample

These are called the "sample suffixes"



Step 0: Construct a sample. For $k = 0, 1, 2$, define

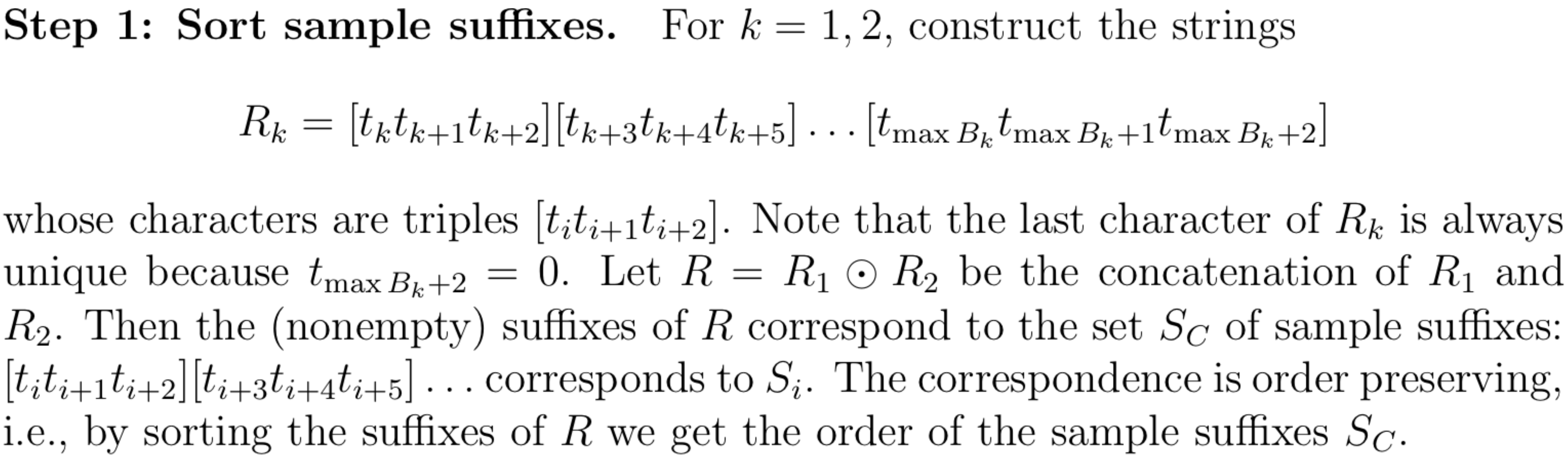
$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and S_C the set of *sample suffixes*.

Example. $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, i.e., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

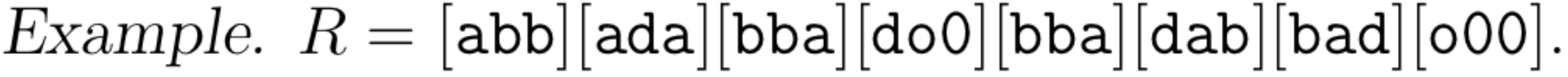
Step 1 — Sorting the sample



Step 1: Sort sample suffixes. For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}][t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}]$$

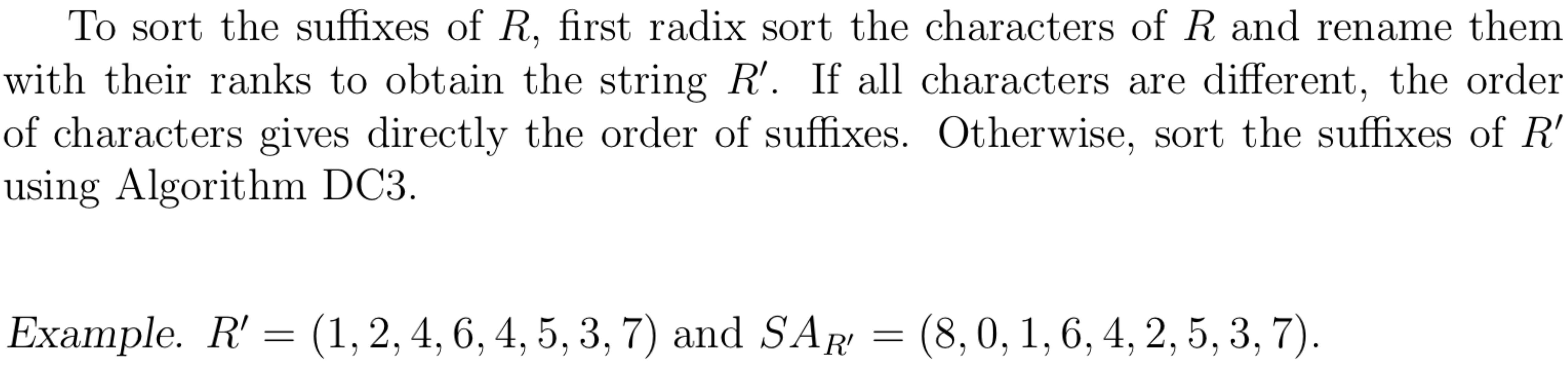
whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of R_k is always unique because $t_{\max B_k + 2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2. Then the (nonempty) suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \ldots$ corresponds to S_i. The correspondence is order preserving, i.e., by sorting the suffixes of R we get the order of the sample suffixes S_C.



Example. $R = [\texttt{abb}][\texttt{ada}][\texttt{bba}][\texttt{do0}][\texttt{bba}][\texttt{dab}][\texttt{bad}][\texttt{o00}]$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1 — Sorting the sample

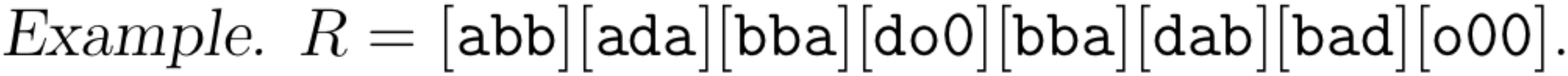


To sort the suffixes of R, first radix sort the characters of R and rename them with their ranks to obtain the string R'. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of R' using Algorithm DC3.

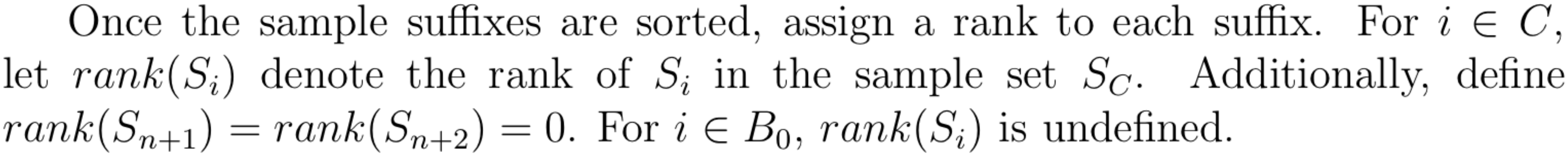
Example. $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

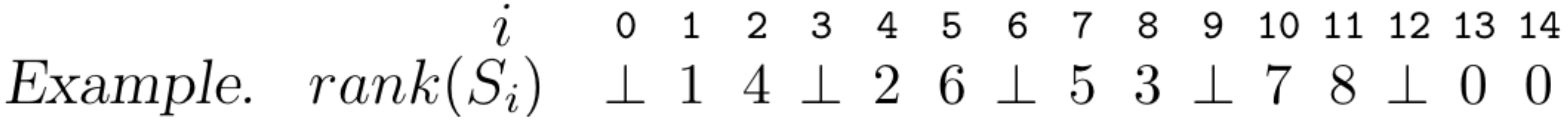
Step 1.5 — Sorting the sample



Example. $R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}]$.



Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.



Example.															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

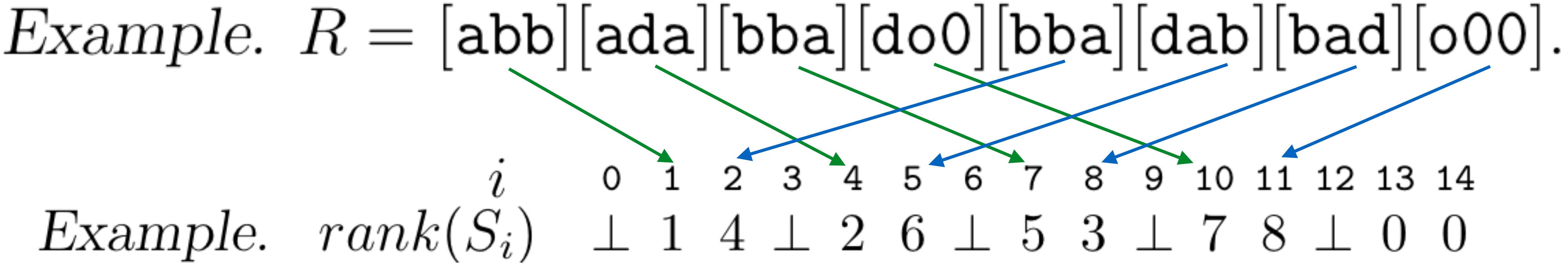
Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1.5 — Sorting the sample

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

Example. i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

	0	1	2	3	4	5	6	7	8	9	10	11
$T[0,n) =$	y	a	b	b	a	d	a	b	b	a	d	o

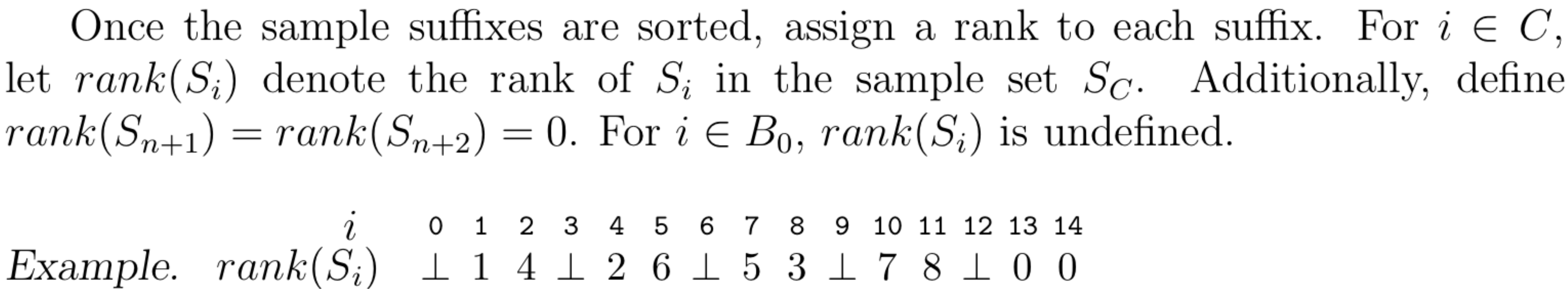


Example. $R =$ [abb][ada][bba][do0][bba][dab][bad][o00].

Example. i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

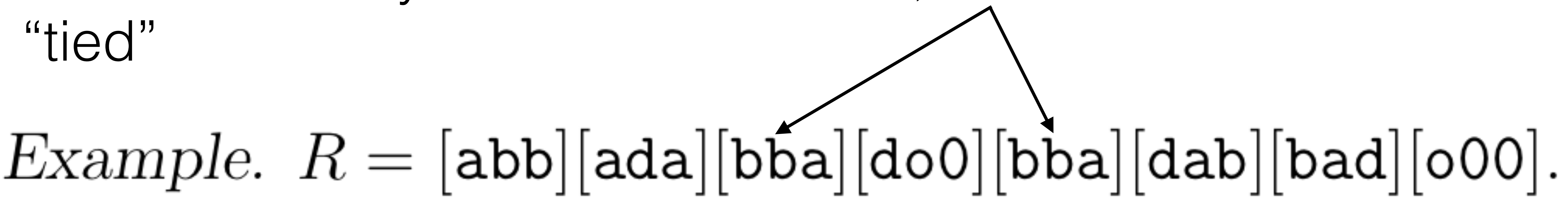
Step 1.5 — Sorting the sample



Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C. Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Example.	$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Note: After only 1 level of recursion, these suffixes would be "tied"



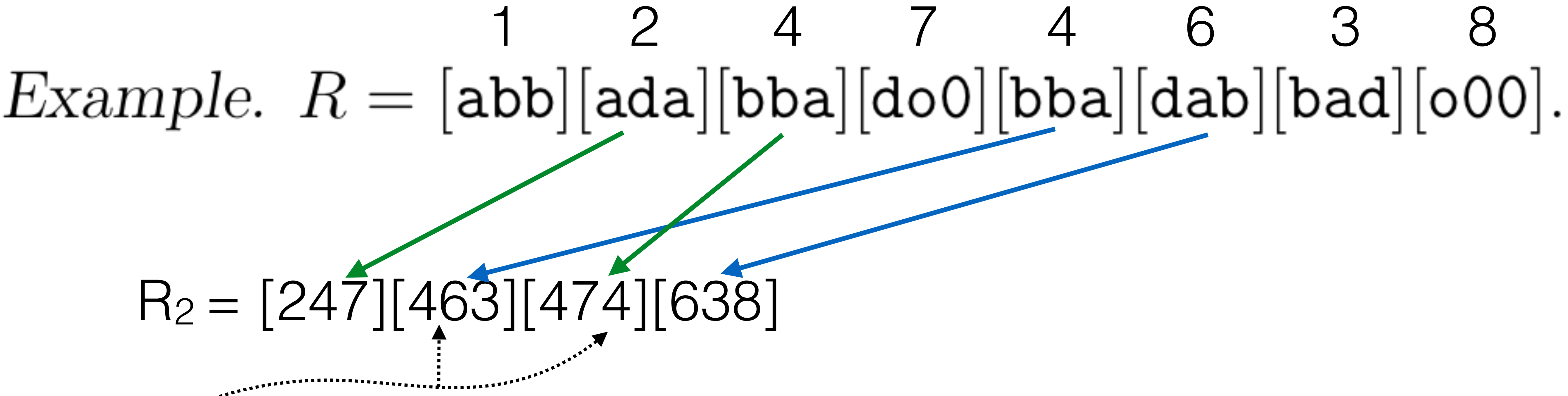
Example. $R = [\text{abb}][\text{ada}][\text{bba}][\text{do0}][\text{bba}][\text{dab}][\text{bad}][\text{o00}]$.

The resolved ranks here represent what we'd get after a second level of recursion.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1.5 — Sorting the sample

$$\begin{array}{rcccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ T[0,n) = & \mathtt{y} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{o} \end{array}$$



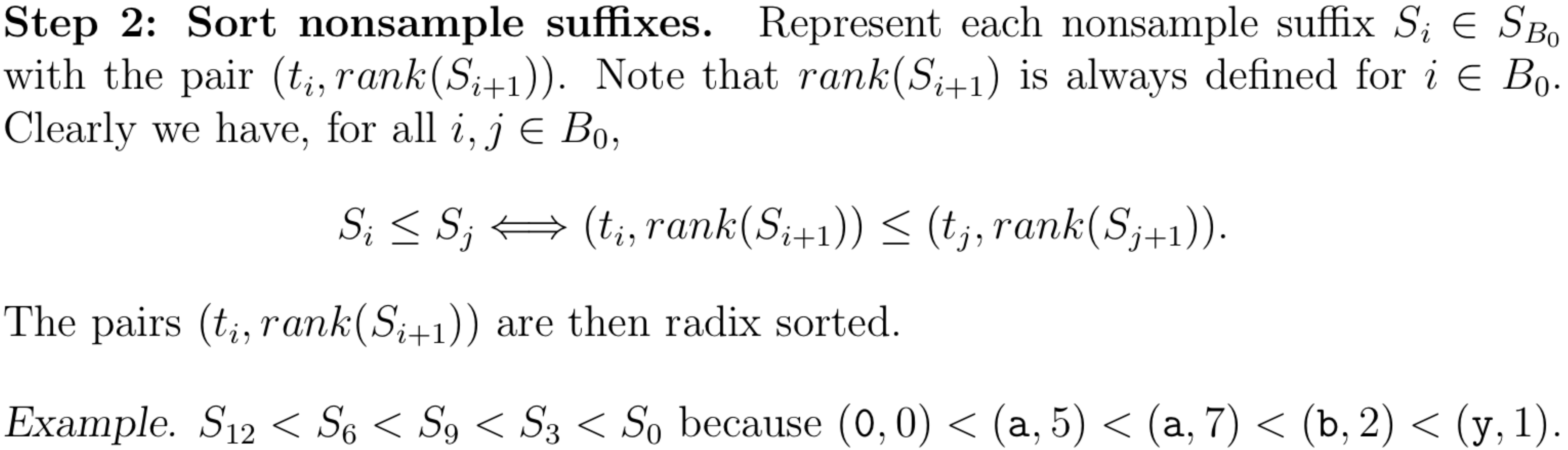
$$\begin{array}{lcccccccc} & 1 & 2 & 4 & 7 & 4 & 6 & 3 & 8 \\ \textit{Example.}\ R = & [\mathtt{abb}] & [\mathtt{ada}] & [\mathtt{bba}] & [\mathtt{do0}] & [\mathtt{bba}] & [\mathtt{dab}] & [\mathtt{bad}] & [\mathtt{o00}]. \end{array}$$

R_2 = [247][463][474][638]

These suffixes were tied at the previous level, but here, we can resolve them. The *lexical renaming* allows us to compare longer and longer suffixes of the text.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 2 — Sorting the non-sample suffixes



Step 2: Sort nonsample suffixes. Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

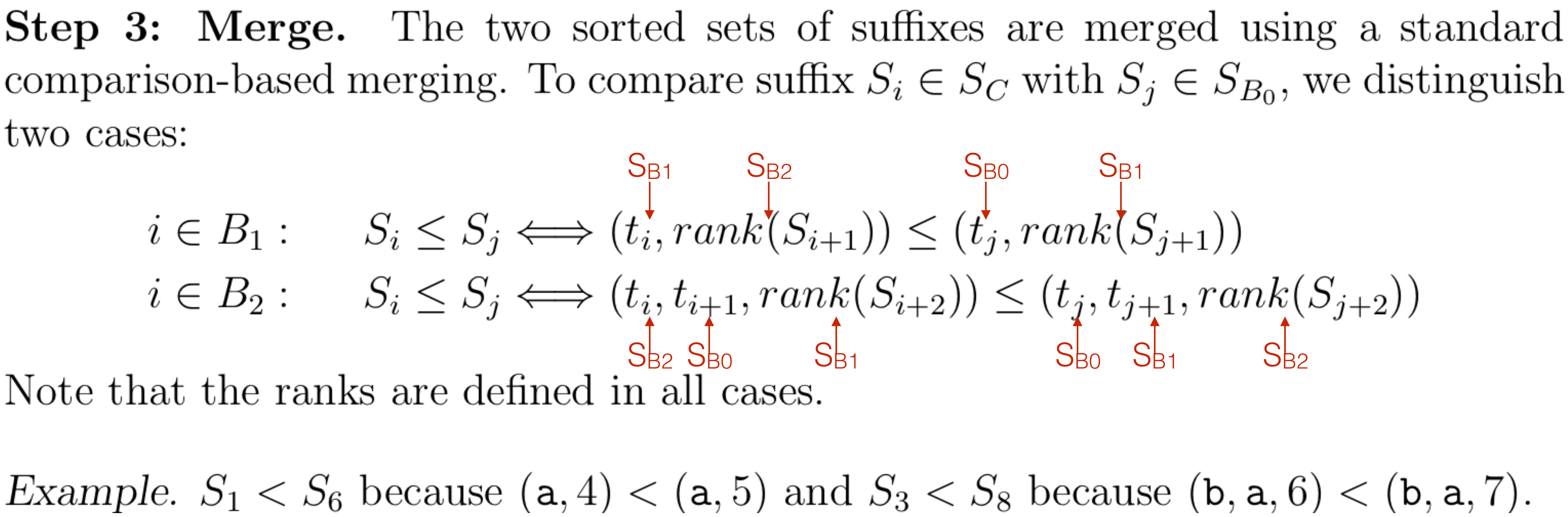
$$S_i \leq S_j \iff (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1})).$$

The pairs $(t_i, rank(S_{i+1}))$ are then radix sorted.

Example. $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(\mathtt{0}, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 2 — Sorting the non-sample suffixes



Step 3: Merge. The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$i \in B_1: \qquad S_i \leq S_j \iff (\overset{S_{B1}}{t_i}, \overset{S_{B2}}{rank}(S_{i+1})) \leq (\overset{S_{B0}}{t_j}, \overset{S_{B1}}{rank}(S_{j+1}))$$

$$i \in B_2: \qquad S_i \leq S_j \iff (\underset{S_{B2}}{t_i}, \underset{S_{B0}}{t_{i+1}}, \underset{S_{B1}}{rank}(S_{i+2})) \leq (\underset{S_{B0}}{t_j}, \underset{S_{B1}}{t_{j+1}}, \underset{S_{B2}}{rank}(S_{j+2}))$$

Note that the ranks are defined in all cases.

Example. $S_1 < S_6$ because $(\mathtt{a}, 4) < (\mathtt{a}, 5)$ and $S_3 < S_8$ because $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Running Time

$$T(n) = O(n) + T(2n/3)$$

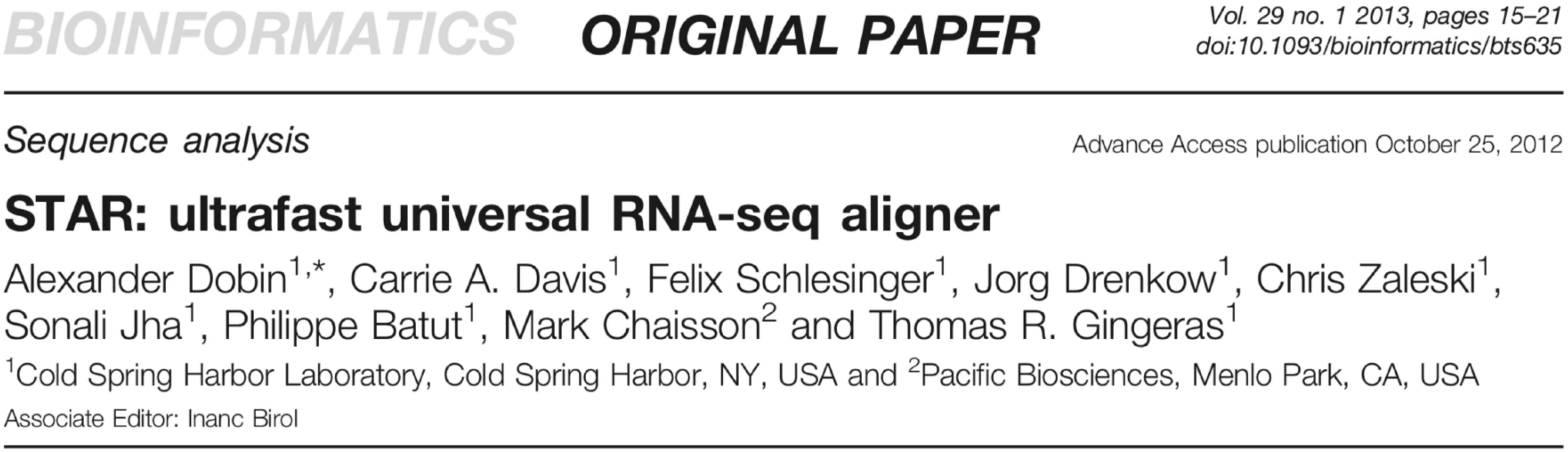
time to sort and merge

array in recursive calls is 2/3rds the size of starting array

Solves to $T(n) = O(n)$:

- Expand big-O notation: $T(n) \leq cn + T(2n/3)$ for some c.
- Guess: $T(n) \leq 3cn$
- Induction step: assume that is true for all $i < n$.
- $T(n) \leq cn + 3c(2n/3) = \mathrm{cn} + 2\mathrm{cn} = 3\mathrm{cn}$ □

Using the suffix array for read alignment: STAR



BIOINFORMATICS **ORIGINAL PAPER**

Vol. 29 no. 1 2013, pages 15–21
doi:10.1093/bioinformatics/bts635

Sequence analysis

Advance Access publication October 25, 2012

STAR: ultrafast universal RNA-seq aligner

Alexander Dobin[1,*], Carrie A. Davis[1], Felix Schlesinger[1], Jorg Drenkow[1], Chris Zaleski[1], Sonali Jha[1], Philippe Batut[1], Mark Chaisson[2] and Thomas R. Gingeras[1]

[1]Cold Spring Harbor Laboratory, Cold Spring Harbor, NY, USA and [2]Pacific Biosciences, Menlo Park, CA, USA

Associate Editor: Inanc Birol

Seeding through SA search for MMPs

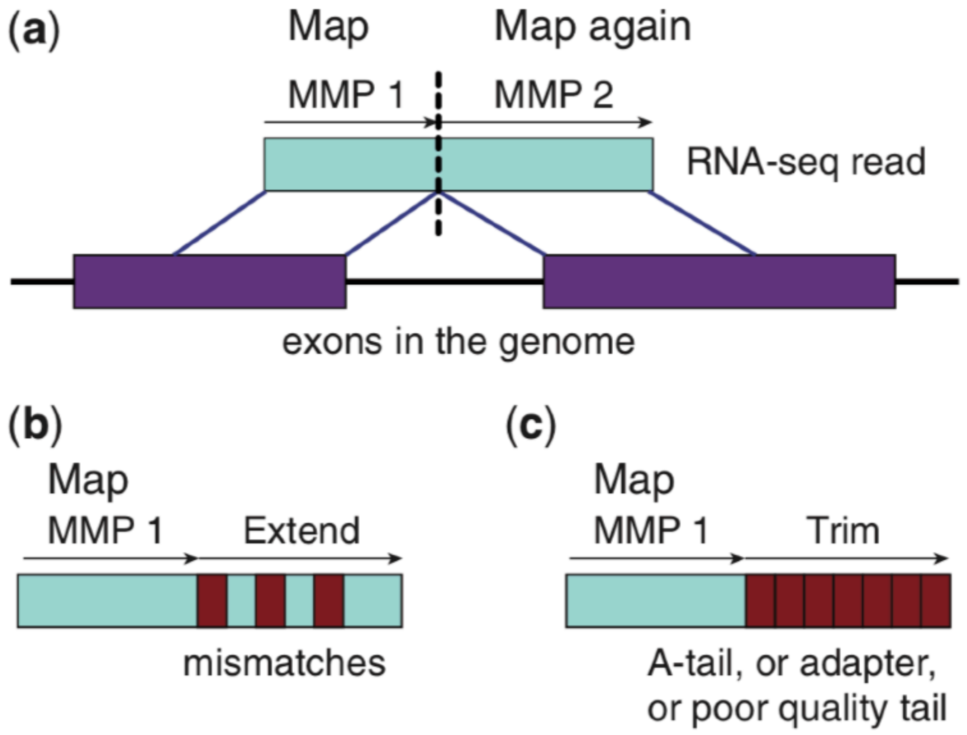


Fig. 1. Schematic representation of the Maximum Mappable Prefix search in the STAR algorithm for detecting (**a**) splice junctions, (**b**) mismatches and (**c**) tails

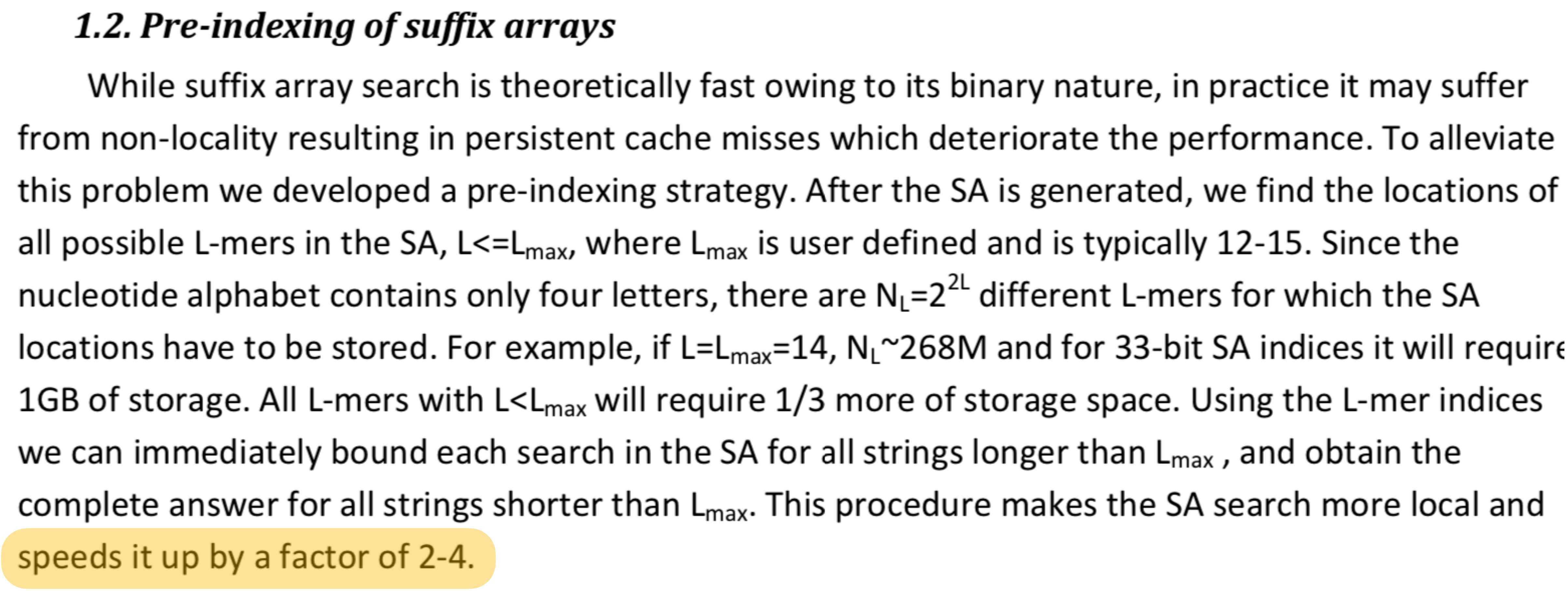
With read sequence R, read location i, and reference sequence G, the MMP(R,i,G) is defined as the longest substring (R_i, R_{i+1}, …, $R_{i+MML-1}$) where MML is the maximum mappable length.

MMPs are computed starting at 5’ end, but also at regular intervals in the read. The read is also searched in the 3’->5’ direction.

Question: How do you search for an MMP in the suffix array?

Seeding through SA search for MMPs

To speed up suffix array search even further, STAR takes advantage of the heuristic we discussed above:



1.2. Pre-indexing of suffix arrays

While suffix array search is theoretically fast owing to its binary nature, in practice it may suffer from non-locality resulting in persistent cache misses which deteriorate the performance. To alleviate this problem we developed a pre-indexing strategy. After the SA is generated, we find the locations of all possible L-mers in the SA, $L<=L_{max}$, where L_{max} is user defined and is typically 12-15. Since the nucleotide alphabet contains only four letters, there are $N_L=2^{2L}$ different L-mers for which the SA locations have to be stored. For example, if $L=L_{max}=14$, N_L~268M and for 33-bit SA indices it will require 1GB of storage. All L-mers with $L<L_{max}$ will require 1/3 more of storage space. Using the L-mer indices we can immediately bound each search in the SA for all strings longer than L_{max} , and obtain the complete answer for all strings shorter than L_{max}. This procedure makes the SA search more local and speeds it up by a factor of 2-4.

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.

Seeds are clustered into windows and “stitched”

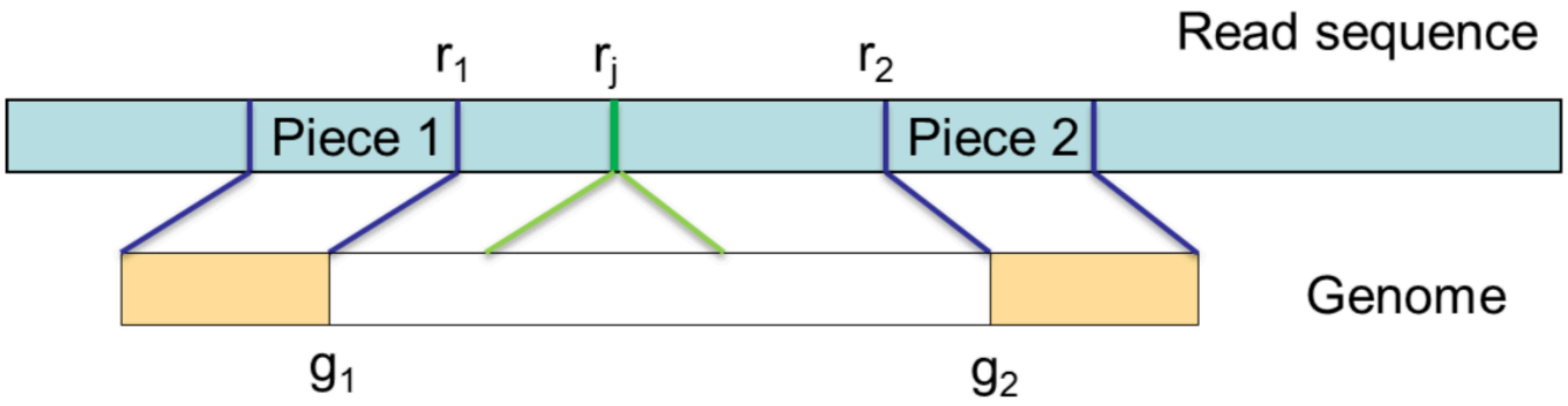
Seeds are filtered by frequency of occurrence to select “anchors” (essentially, infrequently occurring seeds)

Alignment windows (genomic regions) are selected around anchors

All co-linear matches within an alignment window (anchor and non-anchor) are stitched together to form a linear alignment for the whole fragment (ends of a paired-end read are treated as a single fragment)

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.

Stitching and Extension



$$\max_{r_1<r_j<r_2}\left\{\sum_{r=1}^{r_j-r_1}\begin{bmatrix}1 & if\ R(r_1+r)=G(g_1+r)\,\&\,R(r_1+r)\neq G(g_1+r+\Delta)\\ -1 & if\ R(r_1+r)\neq G(g_1+r)\,\&\,R(r_1+r)=G(g_1+r+\Delta)\\ 0 & otherwise\end{bmatrix}-P_{gap}(r_j)\right\}$$

Note: this is a modified recurrence that allows only **1** gap between the two “pieces” being stitched together. This leads to a runtime that is proportional to r_2 - r_1, but this places structural constraints on the types of alignments that can be found.

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.

Scoring alignments

$$S = +\sum_{match} P_m - \sum_{mismatch} P_{mm} - \sum_{inserion} P_{ins} - \sum_{deletion} P_{del} - \sum_{gap} P_{gap} \,.$$

The alignment is scored in a straightforward way. Here, ins / del are indels in the stitching, while “gap” is taken to be an intronic gap between parts of a read or read ends. Gaps are scored differently, with a logarithmic penalty in their length.

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.

Collecting Alignment Results

Finally, alignments from all alignment windows are collected and sorted by score and alignments within a user-specified distance from the best-scoring alignment are reported.

STAR has other abilities we won’t discuss in detail (e.g. finding chimeric alignments by letting reads span multiple alignment windows), and has been *heavily* updated since publication (still in active development). It’s now also commonly used for e.g. fusion detection and can even align circular transcripts or allow back-splicing in alignments.

We will explore the *results* from the STAR paper in a later lecture along with results from other “full text” aligners.

Dobin, Alexander, et al. "STAR: ultrafast universal RNA-seq aligner." *Bioinformatics* 29.1 (2013): 15-21.