Suffixes and Prefixes of strings

Franya Franek
Algorithms Research Group
Computing and Software
McMaster University
Hamilton, Ontario
Canada

Curtin University, Perth, April 2004

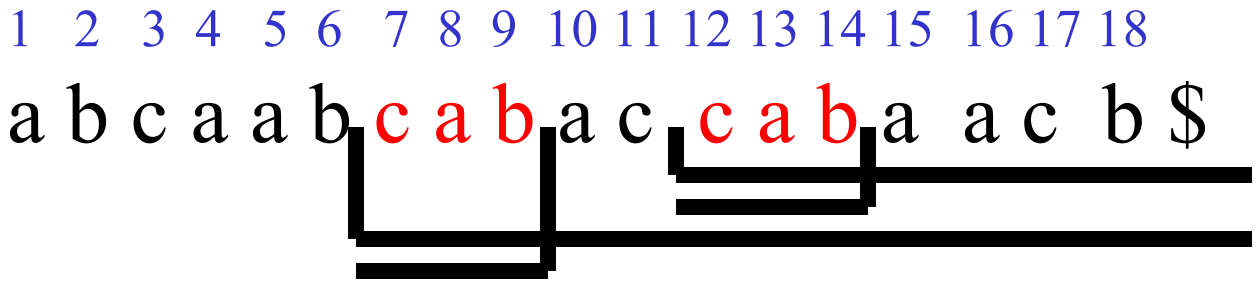
Strings over an alphabet are good models for TEXT, CHROMOSOMES, BINARY FILES, MESSAGES, WEB PAGES …… if you are interested in occurrences and/or retrieval of particular substrings (so-called **pattern matching**)

For instance, consider a string

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	b	c	a	a	b	c	a	b	a	c	c	a	b	a	a	c	b	$

The task is to identify all occurrences of a substring fast and efficiently. Instead of re-scanning the string every time we are looking for a pattern, we “prepare” a data structure to do the search easily.

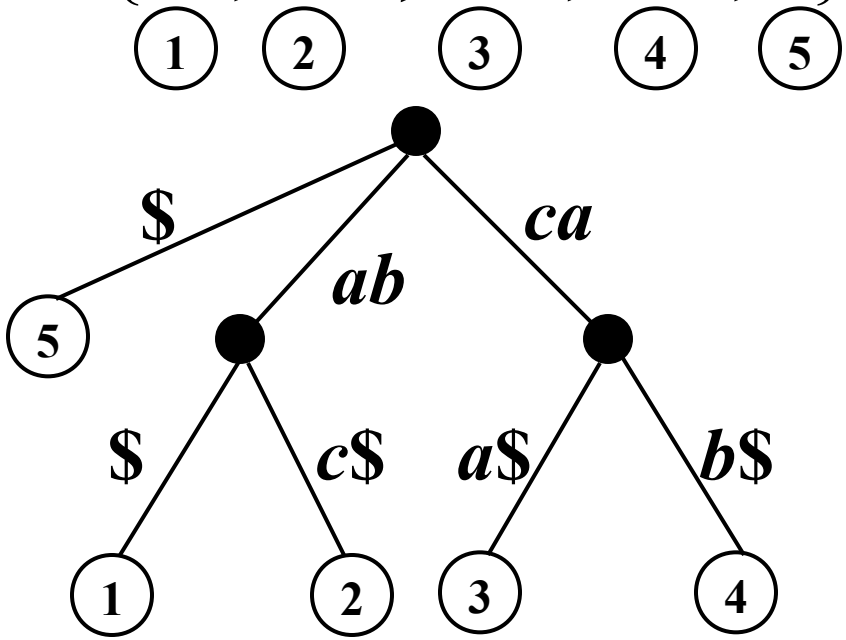
The basic idea -- any substring is a prefix of a suffix:



Patricia trie (Practical Algorithm To Retrieve Information Coded In Alphanumeric)

Morrison (*1968*)

a compacted search tree for a set of distinct strings: {***ab***, ***abc***, ***caa***, ***cab***, ε}



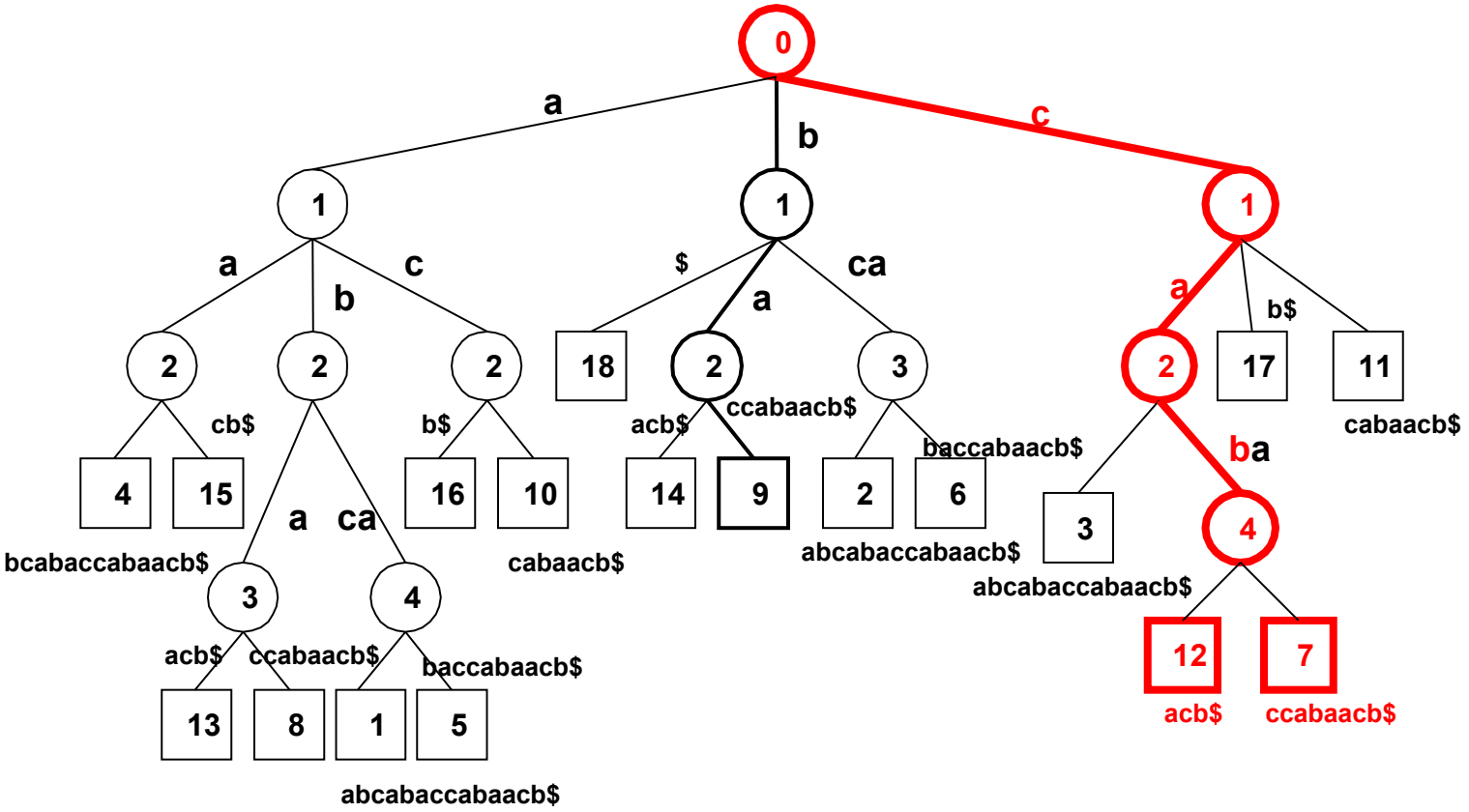
GOOD FOR QUERY OF THE TYPE:

IS A STRING *u* IN THE GIVEN SET OF STRINGS?

Suffix tree of a string x = Patricia trie of the set of all nontrivial suffixes of x *Weiner (1973)*

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

a b c a a b c a b a c c a b a a c b $

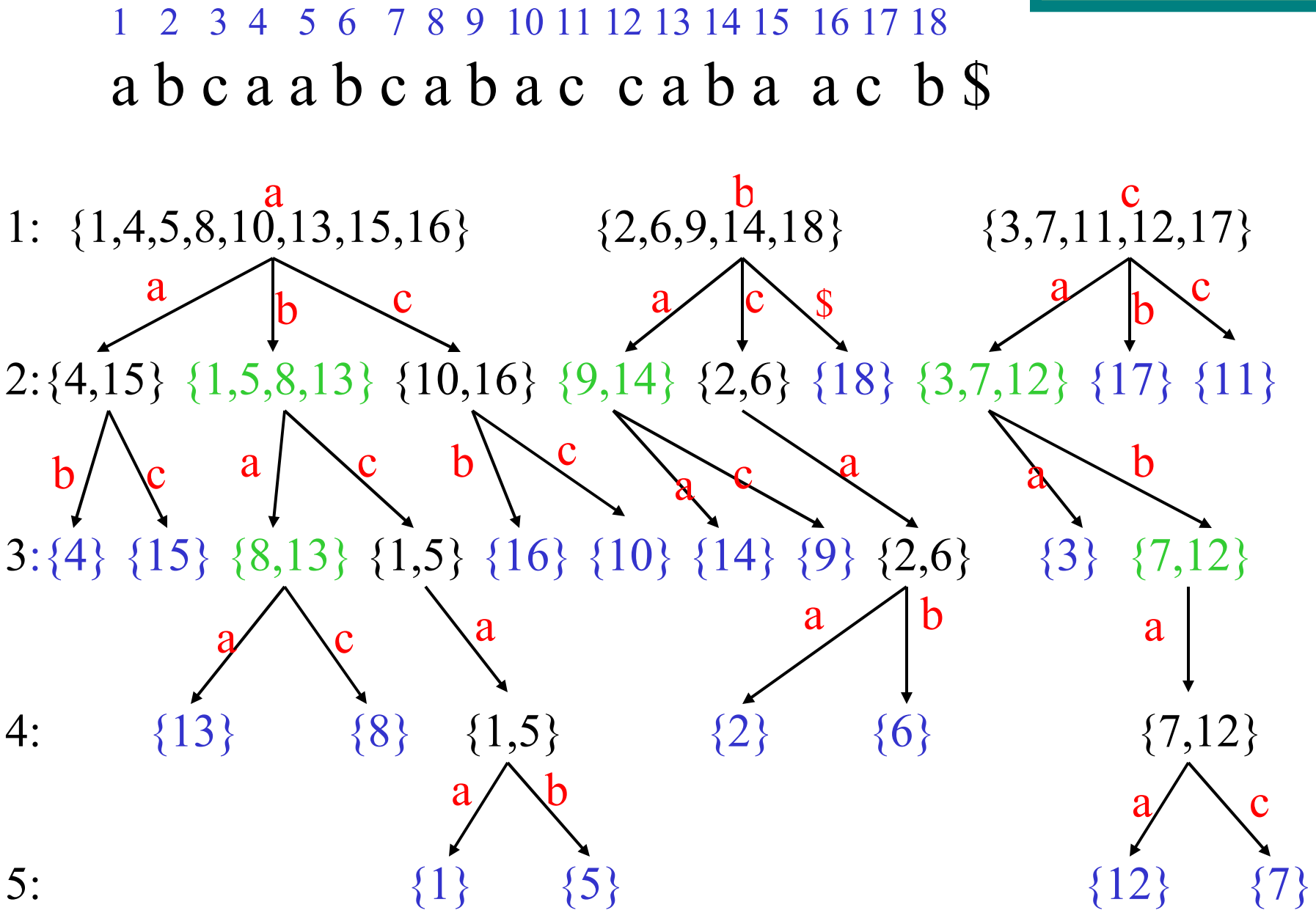


Search for a substring u in $O(|u| \log \alpha)$, where α is the size of the alphabet.

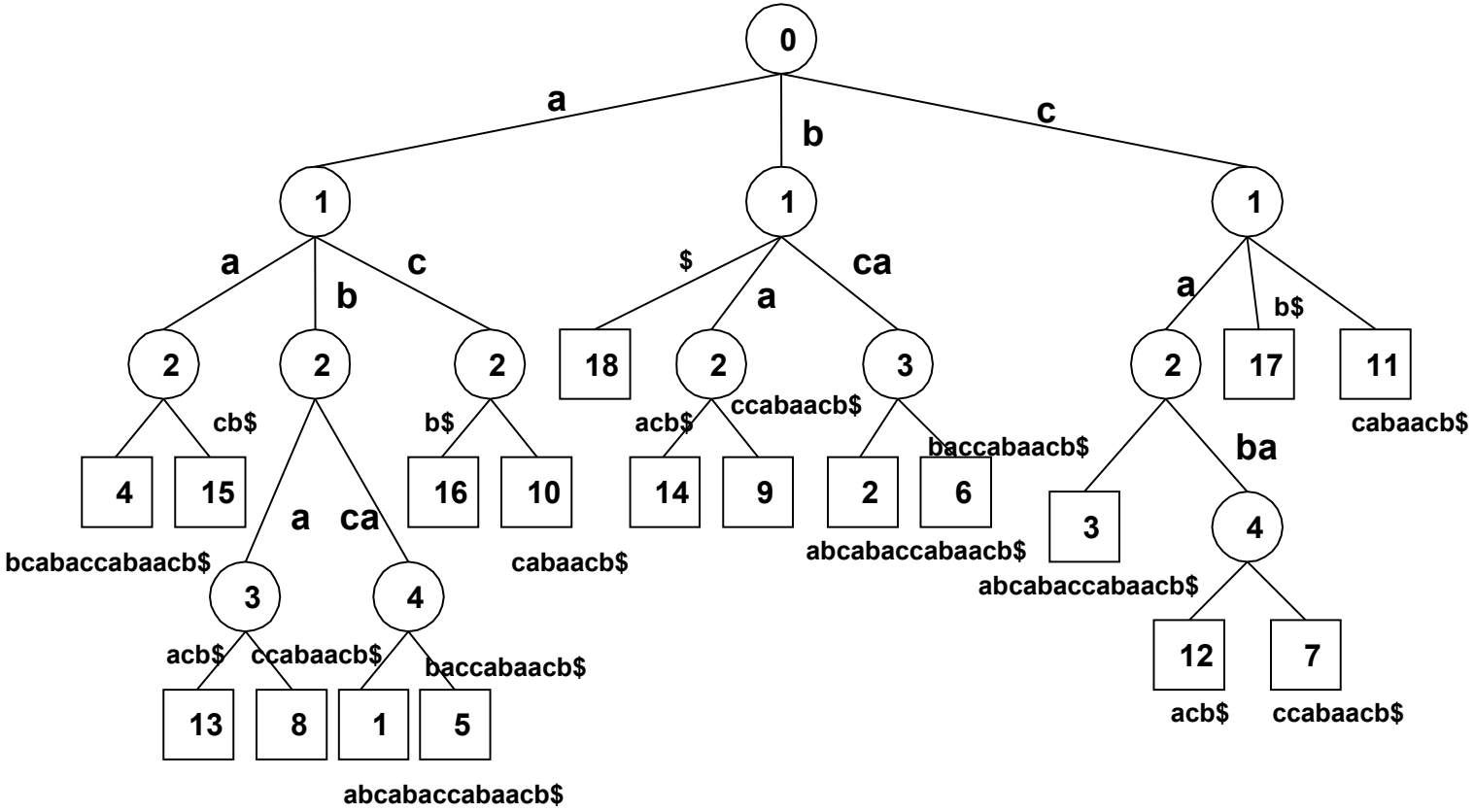
Various applications of suffix trees (also called subword trees) - *Apostolico* (*1985*)
search for reversed strings - *Chen+Seiferas* (*1985*)
applications to DNA matching - *Anderson+ Larsson+Swanson* (*1999*)
applications to data compression - e.g. *Burrows+Wheeler* (*1994*), *Fenwick* (*2001*), *Turpin+Smyth* (*2002*)

To construct a suffix tree

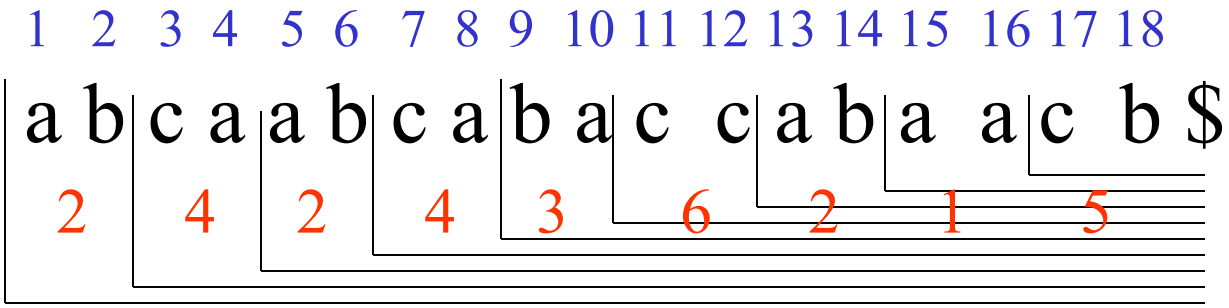
- naïve iterative algorithm $O(n^2)$
- smarter constructions in $O(n \log n)$ - iterative: *Weiner* (*1973*), *McCreight* (*1976*) -faster and less memory, *Crochemore* (*1981*) - all repetitions, *Ukkonen* (*1995*) - suffix links, on-line
  These are linear if alphabet size is fixed, i.e. for small alphabets.
- complex construction in $O(n)$ for any indexed alphabet - recursive: *Farach* (*1997*)


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
a b c a a b c a b a c c a b a a c b $
1: a {1,4,5,8,10,13,15,16} b {2,6,9,14,18} c {3,7,11,12,17}
2: {4,15} {1,5,8,13} {10,16} {9,14} {2,6} {18} {3,7,12} {17} {11}
3: {4} {15} {8,13} {1,5} {16} {10} {14} {9} {2,6} {3} {7,12}
4: {13} {8} {1,5} {2} {6} {7,12}
5: {1} {5} {12} {7}

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
4	3	2	2	4	3	4	3	2	2	1	4	3	2	2	2	1	1



1 Construct suffix tree for odd suffixes of the input string ***x*** by recursion:



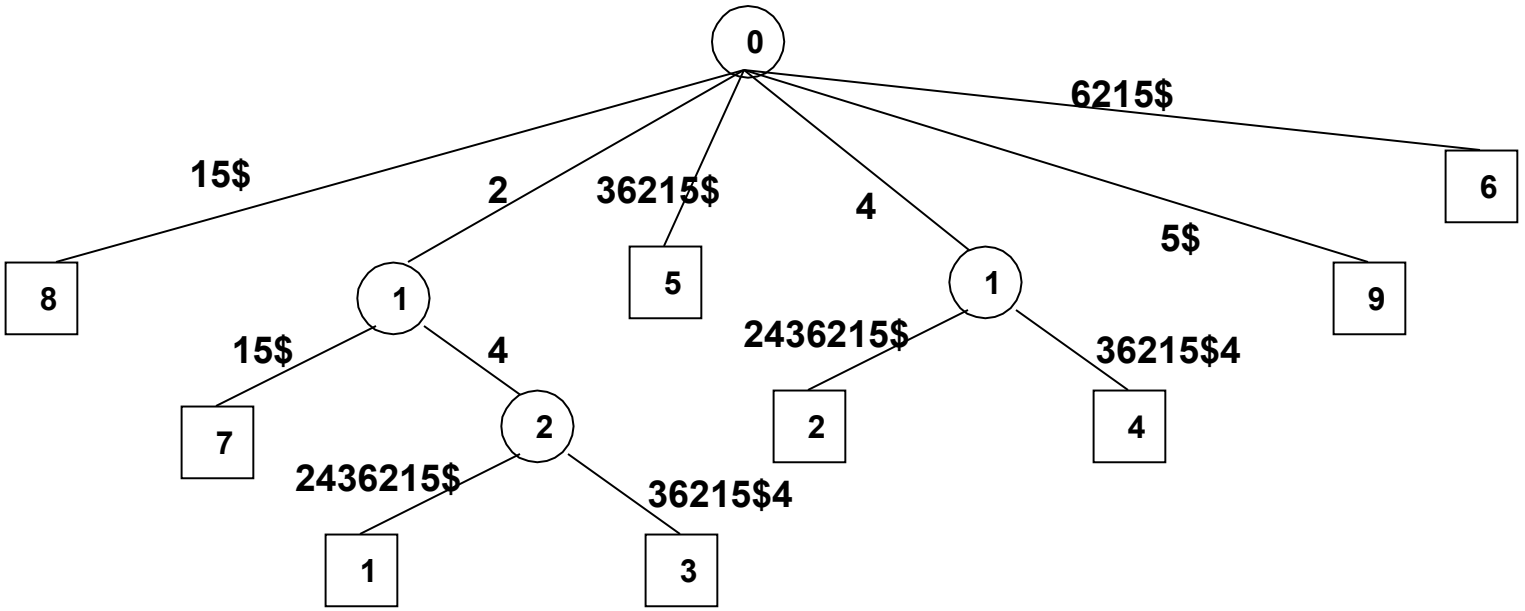
Use radix sort to sort the pairs (x[i],x[i+1]) for odd i:

aa=>1, ab=>2, ba=>3, ca=>4, cb=>5, cc=>6

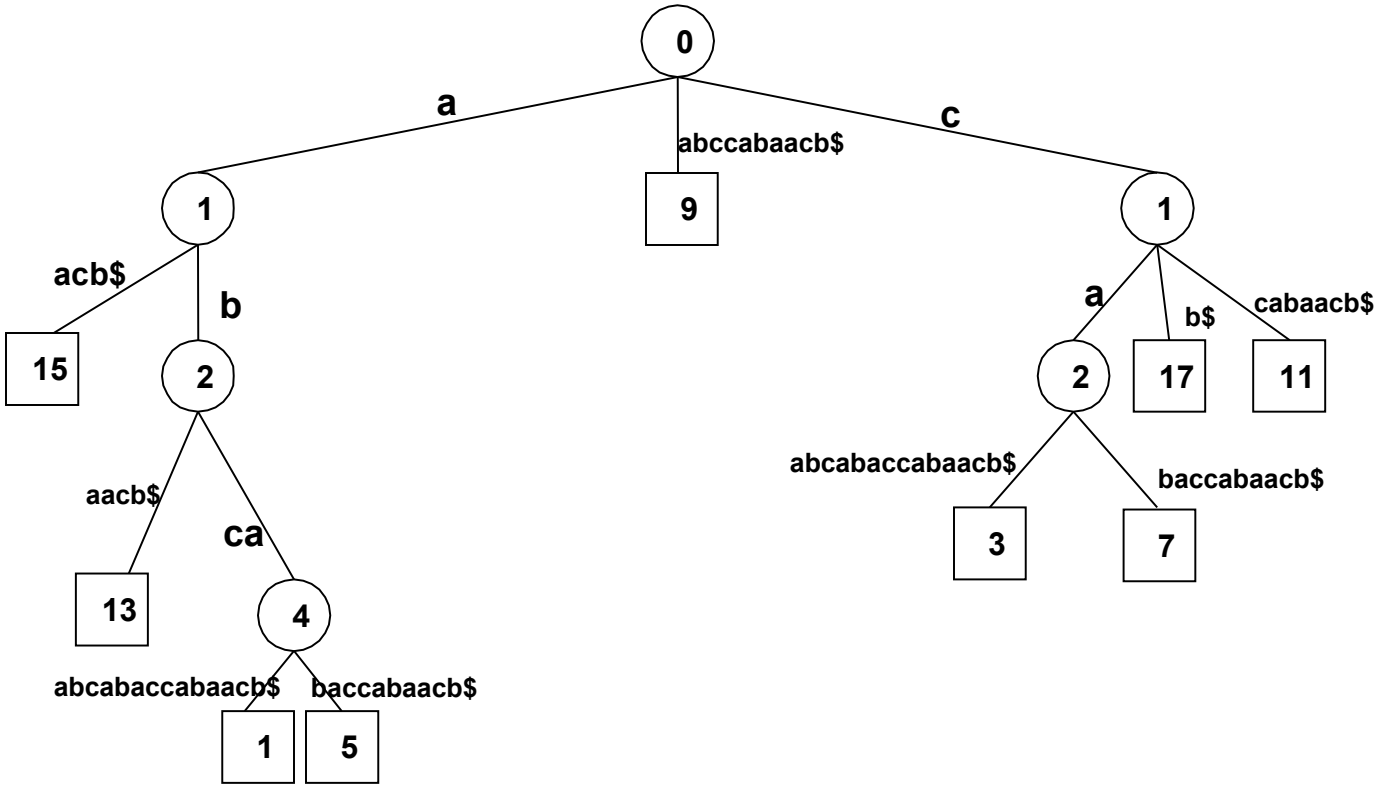
Create string ***y*** = 2 4 2 4 3 6 2 1 5 $
and by recursive call obtain its suffix tree

1 2 3 4 5 6 7 8 9

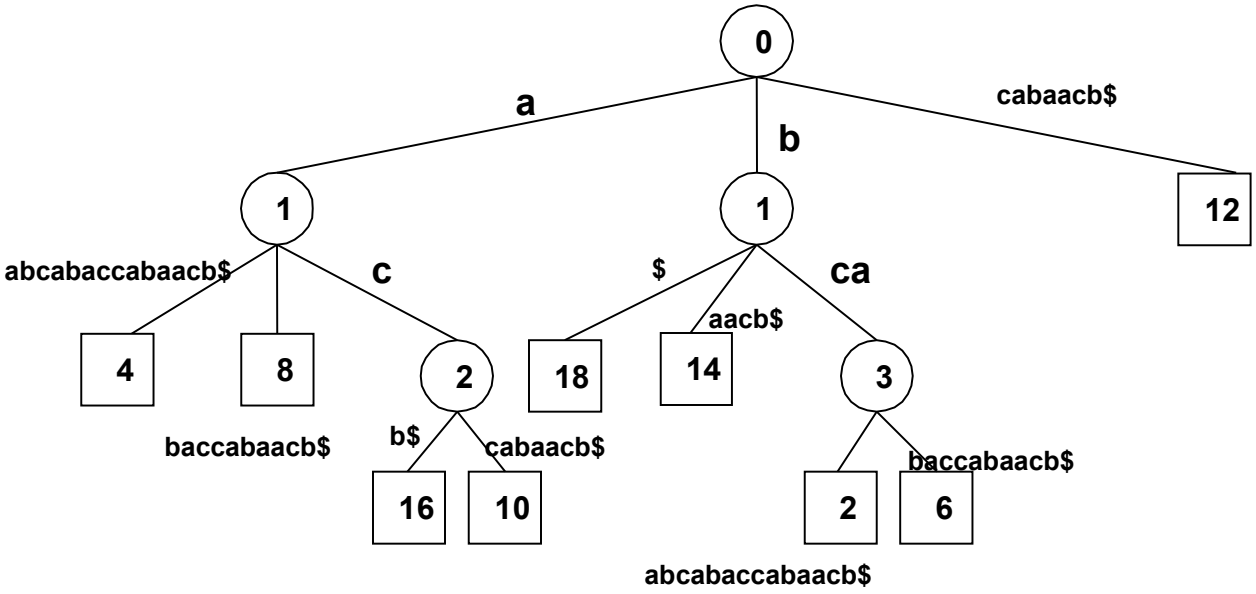
Suffix tree for y = 2 4 2 4 3 6 2 1 5 $



Massage it into a suffix tree of odd suffixes of x in linear time:



2 From this tree create the suffix tree for even suffixes, also in linear time (again using radix sort):



3 Now merge these two suffix trees into one, also in linear time.

The problem with suffix tree --- too much memory!

5|***x***| .. 10|***x***| words required - *Kurtz* (*1999*) reduced suffix tree!

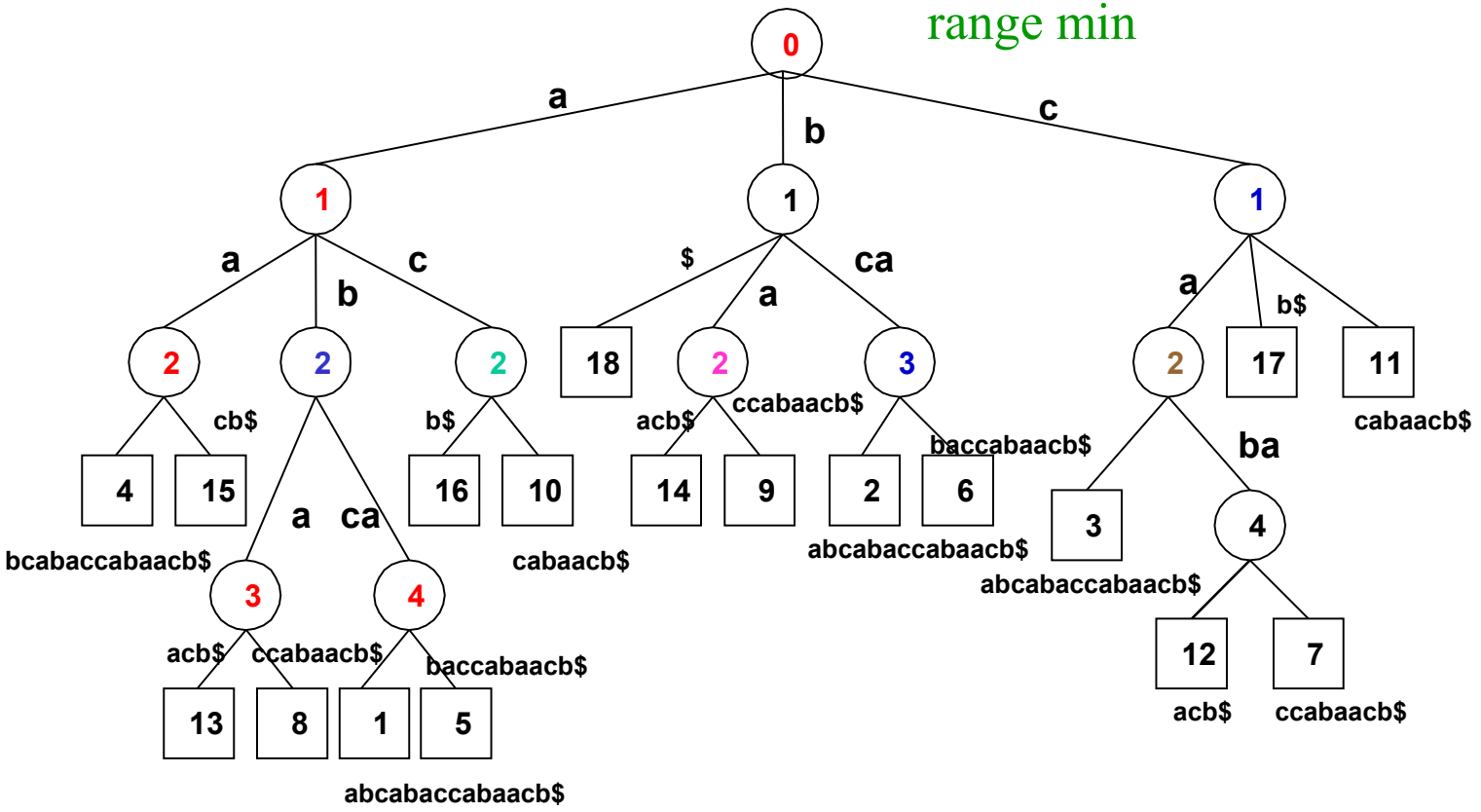
The construction also requires a lot of additional (working) memory.

This is unfeasible and impractical for large strings (e.g. DNA - tens/hundreds of millions of "letters").

Manber+Mayers (*1993*) introduced suffix arrays as an alternative to suffix trees.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	a	b	c	a	a	b	c	a	b	a	c	c	a	b	a	a	c	b	$
suff	4	15	13	8	1	5	16	10	18	14	9	2	6	3	12	7	17	11
lcp		2	1	3	2	4	1	2	0	1	2	1	3	0	2	4	1	1

range min



MM: Search for a substring u in $O(|u| \log n)$, construct ***suff*** in $O(n \log n)$, expected $O(n)$, construct ***lcp*** in $O(n \log n)$, expected $O(n)$.

Kasai et al (*2001*): linear time algorithm to compute ***lcp*** from ***suff***. Problem reduced to suffix sorting.

Abouelhoda et al (*2002*): search for u in $O(|u|)$, with additional linear time preprocessing.

Problems requiring top-down or bottom-up traversal of suffix tree with the same asymptotic complexity using suffix arrays.

Suffix sorting in linear time -- 2003 breakthroughs

Three papers came out claiming linear time recursive algorithms for suffix sorting. They all tried "Farach's" approach:

split suffixes into G_1 and G_2

1. *sort G_1 using recursive reduction of the problem*
2. *sort G_2 using the order of G_1*
3. *merge G_1 and G_2*

Kärkkäinen+Sanders: the simplest, the most elegant, the most memory efficient. The question is: how fast?

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	**b**	**c**	**a**	**a**	**b**	**c**	**a**	**b**	**a**	**c**	**c**	**a**	**b**	**a**	**a**	**c**	**b**

Suppose to have all beige suffixes (2 and 5) sorted.

Then 6 ~ 9 determined by $x[6] \sim x[9]$, or if $x[6]=x[9]$, determined by 7 ~ 10
Thus radix sort with keys of size 2 will do. So we have all gray suffixes sorted.

How to merge beige and gray suffixes?

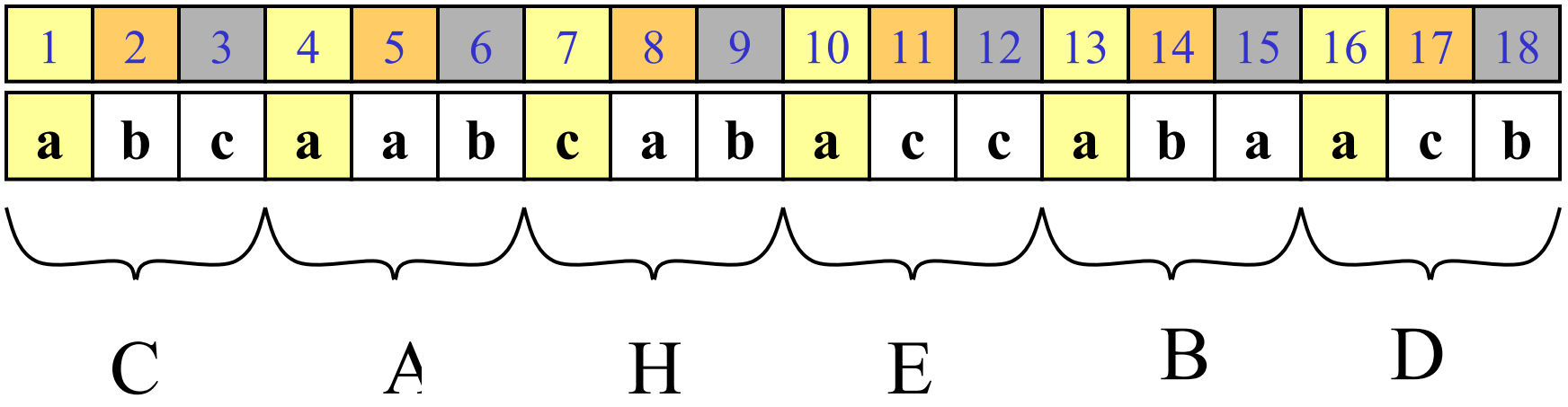
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	**b**	**c**	**a**	**a**	**b**	**c**	**a**	**b**	**a**	**c**	**c**	**a**	**b**	**a**	**a**	**c**	**b**

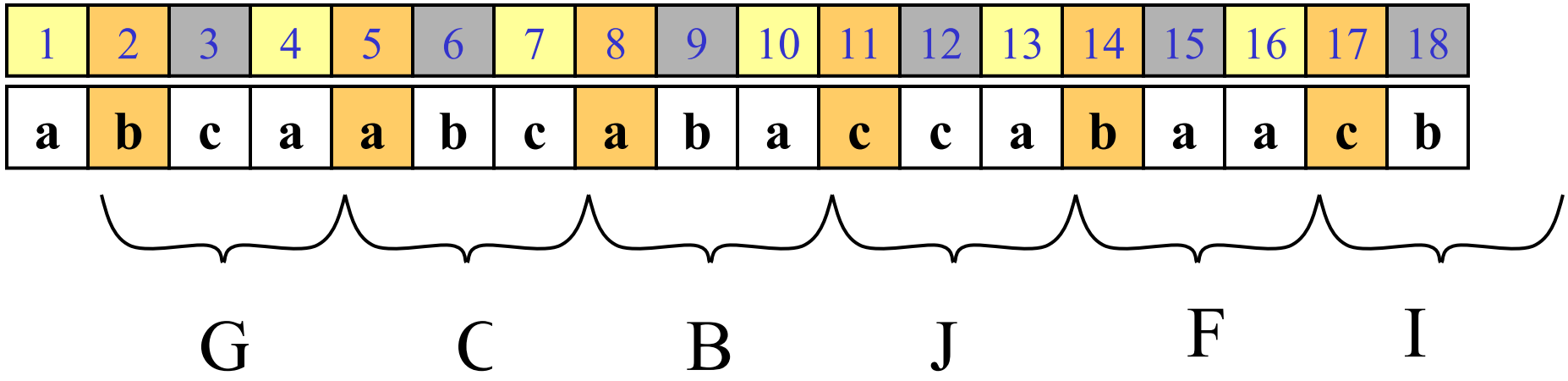
Simple comparison-based merge:

 determined by the first letter or by

 determined by the first letter or by

So, how to sort beige suffixes? Like Farach!





Using radix sort, sort the triples.

1	4	7	10	13	16	2	5	8	11	14	17	
C	A	H	E	B	D	G	C	B	J	F	I	
	A	H	E	B	D	G	C	B	J	F	I	4
				B	D	G	C	B	J	F	I	13
								B	J	F	I	8
C	A	H	E	B	D	G	C	B	J	F	I	1
							C	B	J	F	I	5
					D	G	C	B	J	F	I	16
			E	B	D	G	C	B	J	F	I	10
										F	I	14
						G	C	B	J	F	I	2
		H	E	B	D	G	C	B	J	F	I	7
											I	17

This approach works for any “division” as long as the beige blocks are bigger than the gray blocks. Of course, using bigger beige blocks requires longer radix sort, however it decreases the recursion and memory use.

In many ways, using blocks of size 3 optimizes the solution, see results of a crude simulation:

N=100
2+1: total=1415.000000,rec=8,mem=943.333333
3+2: total=1541.866667,rec=6,mem=566.400000
N=1000
2+1: total=14890.000000,rec=14,mem=9926.666667
3+2: total=16202.666667,rec=10,mem=5952.000000
N=10000
2+1: total=149855.000000,rec=20,mem=99903.333333
3+2: total=163209.200000,rec=15,mem=59954.400000
N=100000
2+1: total=1499840.000000,rec=25,mem=999893.333333
3+2: total=1633170.000000,rec=19,mem=599940.000000
N=1000000
2+1: total=14999770.000000,rec=31,mem=9999846.666667
3+2: total=16333150.400000,rec=24,mem=5999932.800000
N=10000000
2+1: total=14999770.000000,rec=31,mem=9999846.666667
3+2: total=16333150.400000,rec=24,mem=5999932.800000

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a	**b**	**c**	**a**	**a**	**b**	**c**	**a**	**b**	**a**	**c**	**c**	**a**	**b**	**a**	**a**	**c**	**b**

By recursive reduction order all beige suffixes.

 determined by the first letter or by

Problem: how to merge beige and gray suffixes. Based on constant-time solution of range minimum problem, requires $O(n)$ preprocessing and large additional memory. However, the recursion is faster (1/2) than for *Kärkkäinen+Sanders* (2/3).

Problem: the proof of Lemma 3 incorrect, I have several counterexamples; the authors have not responded to any communication to either clarify my misunderstanding or provide some code.

Are the linear suffix sorting algorithms practical? They seem to require at least $4|\boldsymbol{x}|$ working memory. *Larsson+Sadakane* (*1999*): sorting suffixes as independent strings - for most real-world data very fast, though worst-case complexity is $\Omega(n^2)$, requires very little extra space (for instance **bzip2** by *Seward*).

Manzini+Ferragina (*2002*): very fast, very little extra memory (0.03n), however worst-case complexity is also is $\Omega(n^2)$. They posed a problem: lightweight ($O(n \log n)$, sublinear memory) algorithm?

Burkhardt+Kärkkäinen (*2003*): an $O(n \log n)$ suffix sorting algorithms with $O(n / \sqrt{\log n})$ memory requirement. Based on the idea of **difference covers** (VLSI design, distributed mutual exclusion -- *Colbourn+Ling* (*2000*)).

For any pair of suffixes $\boldsymbol{x}[i..n]$, $\boldsymbol{x}[j..n]$ find the smallest k such that the order of $\boldsymbol{x}[i+k..n]$ and $\boldsymbol{x}[j+k..n]$ is known (*anchor pair*).

A *difference cover D modulo v*: set of integers $0..v\text{-}1$ such that for any $0 < i < v$ there are $i_1, i_2 \in D$ so that $i = i_1 - i_2 \pmod{v}$. For $\forall\ i, j$ compute $k = \delta(i, j) \in [0, v)$ so that $((i+k) \bmod v)$, and $((j+k) \bmod v)$ are both in D (can be done in $O(v)$). Then sort all suffixes whose starting position is in D. The sort of all suffixes is transformed to a sort on keys of length $\leq v+1$.

Note that *Kärkkäinen+Sanders* algorithm uses *D modulo* 3!

Colbourn+Ling (*2000*): For every v, a difference cover *D modulo* v of size $|D| \leq \sqrt{1.5v}+6$ can be computed in $O(\sqrt{v})$ time.

Can suffix array really "replace" the string? *Bannai, Inenaga, Shinohara, and Takeda* (*2003*): given an array, conditions can be checked if it is a suffix array of a string (must be a permutation of n) and such a string with a minimal alphabet is inferred from the array in $O(n)$ time.

www.cas.mcmaster.ca/~franek