Divide + Conquer Multiplication:

Consider the problem of binary integer multiplication. How are such multiplications typically performed?

- Think back to the "partial products" algorithm from gradeschool. That is, taking single digit products and summing.

E.g.
$$\begin{array}{r} 12 \\ \times\ 13 \\ \hline 36 \\ 120 \\ \hline 156 \end{array}$$
or in binary
$$\begin{array}{r} 1100 \\ \times\ 1101 \\ \hline 1100 \\ 00000 \\ 110000 \\ 1100000 \\ \hline 10011100 \end{array}$$

Assuming addition is $O(n)$, it takes $O(n)$ time to compute each partial product and $O(n)$ time to combine it with the running sum of all partial products $\Rightarrow O(n^2)$ algo.

Consider the following approach (assume x, y are n-bit integers for n a power of 2):

$$x = \boxed{\qquad\qquad x \qquad\qquad} = \boxed{\qquad\qquad x_L \qquad\qquad}\ \boxed{\qquad\qquad x_R \qquad\qquad} = 2^{n/2} x_L + x_R$$

$$y = \boxed{\qquad\qquad y \qquad\qquad} = \boxed{\qquad\qquad y_L \qquad\qquad}\ \boxed{\qquad\qquad y_R \qquad\qquad} = 2^{n/2} y_L + y_R$$

E.g. if $x = 1011\ 0110$ then $x_L = 1011$, $x_R = 0110$

and $x = 1011 \cdot 2^4 + 0110$

Given this representation, we can write the product

$$\begin{aligned} xy &= (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) \\ &= 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

- additions are $O(n)$ as are multiplications by powers of 2 (left-shift)
- Significant ops are 4 $n/2$-bit multiplications

$$x_L y_L,\ x_L y_R,\ x_R y_L,\ x_R y_R$$

- handle these via 4 recursive calls. If we write down the recurrence for this it looks like:

$$T(n) \leq 4T(n/2) + O(n)$$

- remember that solving this gives

$$T(n) \in O(n^{\log_2 4}) = O(n^2) \quad \ldots \quad \ddot{\frown}$$

- this is no better than our original partial products algorithm!

The problem is that we are doing too many recursive multiplications.

Can we do better?

Consider the following identity (discovered by Gauss) for multiplying complex numbers:

$$(a+bi)(c+di) = ac - bd + (bc+ad)i$$

Gauss noted that, despite the fact that it looks like there are $\underline{4}$ multiplications required here, there are only $\underline{3}$ ... why?

$$(bc+ad) = (a+b)(c+d) - ac - bd$$

so, if we have ac, bd already computed, we can use it to compute $(bc+ad)$ with only $\underline{1}$ additional multiplication.

How can we use this in our algorithm?

Idea:

Represent $x_L y_R + x_R y_L$ by $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

So, overall, we need only 3 recursive multiplications

$x_L y_L$, $x_R y_R$ and $(x_L + x_R)(y_L + y_R)$

Using this "trick", our recurrence becomes:

$$\begin{aligned} T(n) &\leq 3T(n/2) + O(n) \\ &\leq 3T(n/2) + cn \end{aligned}$$

thus, solving this recurrence we get

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$$

This can be realized by the following algorithm

```
multiply(x, y):
    n = max(size of x, size of y)
    if n = 1: return $x \cdot y$

    $x_L, x_R$ = leftmost $\lceil n/2 \rceil$ bits of x, remaining rightmost bits of x
    $y_L, y_R$ = leftmost $\lceil n/2 \rceil$ bits of y, remaining rightmost bits of y
    $P_1$ = multiply($x_L, y_L$)
    $P_2$ = multiply($x_R, y_R$)
    $P_3$ = multiply($x_L + x_R, y_L + y_R$)
    return $(P_1 \cdot 2^n) + [(P_3 - P_1 - P_2) \cdot 2^{n/2}] + P_2$
```

CSE 373: Suffix Arrays & the DC3 algorithm

Stony Brook University

All slides in this lecture **not** marked with "*" of Ben Langmead.

Suffix array

T\$ = abaaba\$ ⟵ As with suffix tree, *T* is part of index

SA(*T*) =
(SA = "Suffix Array")

6	\$
5	a \$
2	a a b a \$
3	a b a \$
0	a b a a b a \$
4	b a \$
1	b a a b a \$

m + 1 integers

Suffix array of *T* is an array of integers in [0, *m*] specifying the lexicographic order of *T*\$'s suffixes

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
1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

sort the suffixes alphabetically

$\longrightarrow$

the indices just "come along for the ride"

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

slide courtesy of Carl Kingsford

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
1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

sort the suffixes alphabetically

$\xrightarrow{\hspace{2cm}}$

the indices just "come along for the ride"

8
6
2
5
1
7
4
3

slide courtesy of Carl Kingsford

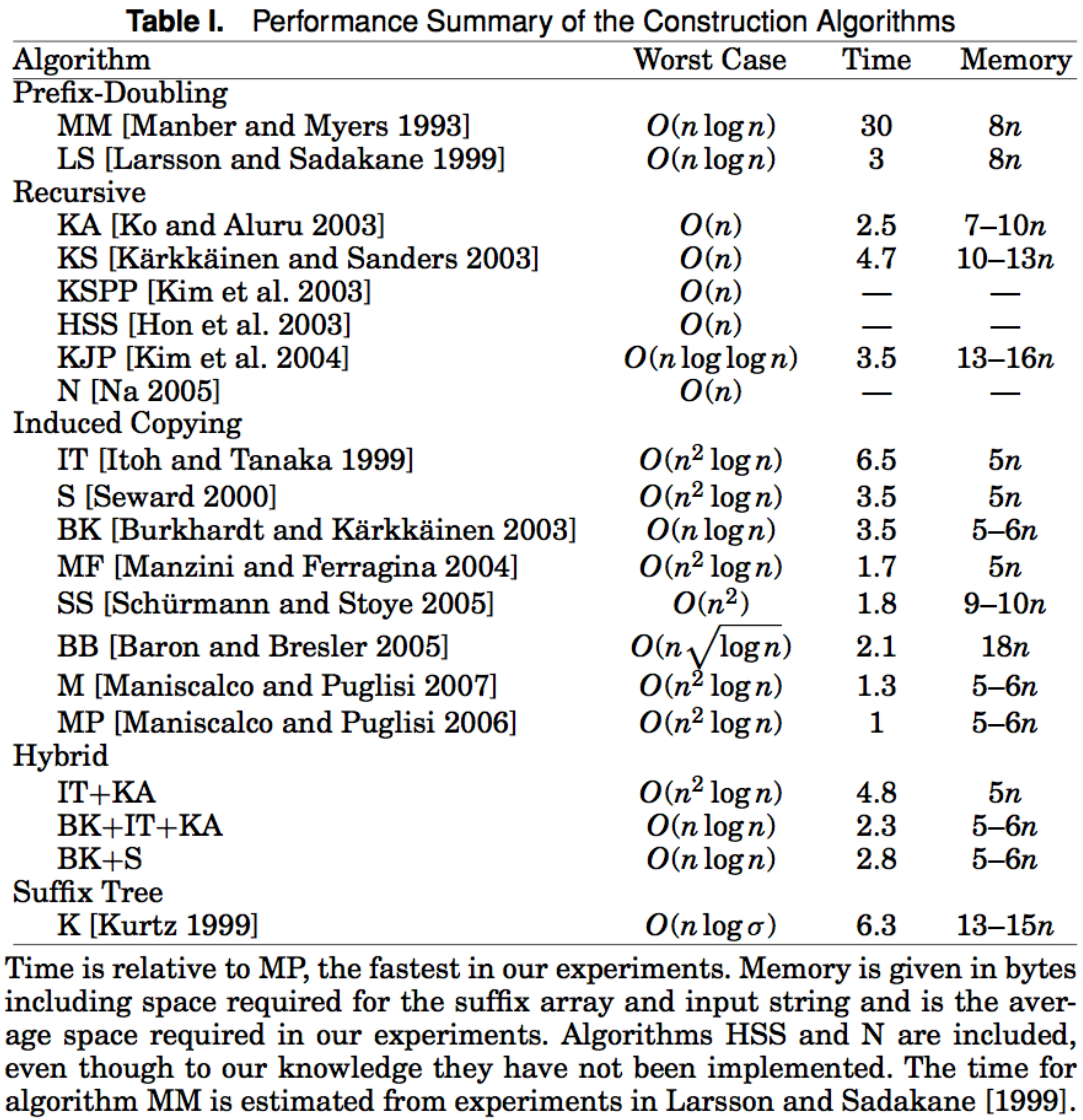


Table I. Performance Summary of the Construction Algorithms

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.

*slide courtesy of Carl Kingsford

Suffix array: querying

Is *P* a substring of *T*?

1. For *P* to be a substring, it must be a prefix of ≥1 of *T*'s suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Is *P* a substring of *T*?

1. For *P* to be a substring, it must be a prefix of ≥1 of *T*'s suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Is *P* a substring of *T*?

Do binary search, check whether *P* is a prefix of the suffix there

How many times does *P* occur in *T*?

Two binary searches yield the range of suffixes with *P* as prefix; size of range equals # times *P* occurs in *T*

Worst-case time bound?

O(log$_2$ *m*) bisections, *O*(*n*) comparisons per bisection, so O(*n* log *m*)

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

Suffix array: querying

Consider further: binary search for suffixes with *P* as a prefix

Assume there's no **$** in *P*. So *P* can't be equal to a suffix.

Initialize $\underset{\text{“left”}}{\underset{\uparrow}{\boldsymbol{l}}} = 0$, $\underset{\text{“center”}}{\underset{\uparrow}{\boldsymbol{c}}} = \mathit{floor}(m/2)$ and $\underset{\text{“right”}}{\underset{\uparrow}{\boldsymbol{r}}} = m$ (just past last elt of SA)

Notation: We'll use use **SA[*l*]** to refer to the suffix corresponding to suffix-array element ***l***. We could write *T*[**SA[*l*]**:], but that's too verbose.

Throughout the search, invariant is maintained:

SA[*l*] < *P* < **SA[*r*]**

Suffix array: querying

Throughout search, invariant is maintained:

$\mathbf{SA}[l] < P < \mathbf{SA}[r]$

What do we do at each iteration?

Let c = floor(($r + l$) / 2)

If $P < \mathbf{SA}[c]$, either stop or let $r = c$ and iterate

If P > $\mathbf{SA}[c]$, either stop or let $l = c$ and iterate

When to stop?

$P < \mathbf{SA}[c]$ and $c = l + 1$ - answer is c

$P > \mathbf{SA}[c]$ and $c = r - 1$ - answer is r

Suffix array: querying

```
def binarySearchSA(t, sa, p):
    assert t[-1] == '$' # t already has terminator
    assert len(t) == len(sa) # sa is the suffix array for t
    if len(t) == 1: return 1
    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]
    while True:
        c = (l + r) // 2
        # determine whether p < T[sa[c]:] by doing comparisons
        # starting from left-hand sides of p and T[sa[c]:]
        plt = True # assume p < T[sa[c]:] until proven otherwise
        i = 0
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far
        if plt:
            if c == l + 1: return c
            r = c
        else:
            if c == r - 1: return r
            l = c
```

# loop iterations ≈ length of Longest Common Prefix (LCP) of *P* and **SA[*c*]**

If we already know something about LCP of *P* and **SA[*c*]**, we can save work

Suffix array: binary search

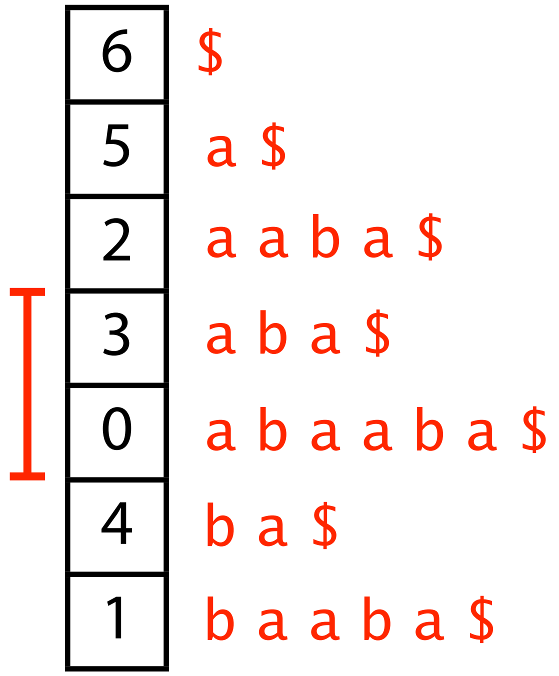
We can do the same thing for a sorted list of suffixes:

```
from bisect import bisect_left, bisect_right

t = 'abaaba$'
suffixes = sorted([t[i:] for i in xrange(len(t))])

st, en = bisect_left(suffixes, 'aba'),
         bisect_left(suffixes, 'abb')

print(st, en) # output: (3, 5)
```



Suffix array: sorting suffixes

One idea: Use your favorite sort, e.g., quicksort

0	a b a a b a $
1	b a a b a $
2	a a b a $
3	a b a $
4	b a $
5	a $
6	$

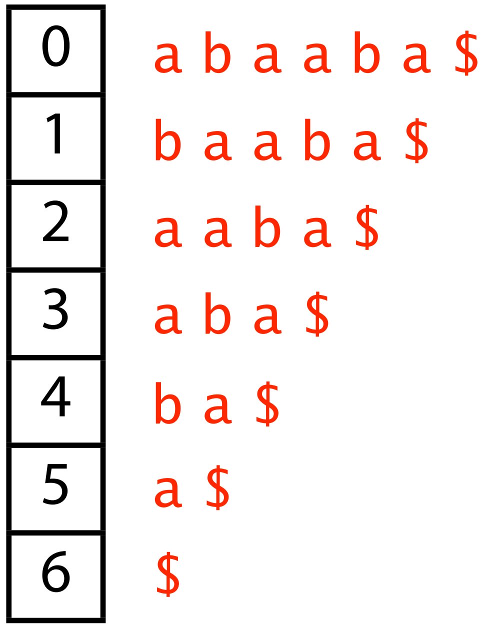
```
def quicksort(q):
    lt, gt = [], []
    if len(q) <= 1:
        return q
    for x in q[1:]:
        if x < q[0]:
            lt.append(x)
        else:
            gt.append(x)
    return quicksort(lt) + q[0:1] + quicksort(gt)
```

Expected time: O($m^2 \log m$)

Not $O(m \log m)$ because a suffix comparison is O(m) time

Suffix array: sorting suffixes

One idea: Use a sort algorithm that's aware that the items being sorted are strings, e.g. "multikey quicksort"



Essentially $O(m^2)$ time

Bentley, Jon L., and Robert Sedgewick. "Fast algorithms for sorting and searching strings." *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1997

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

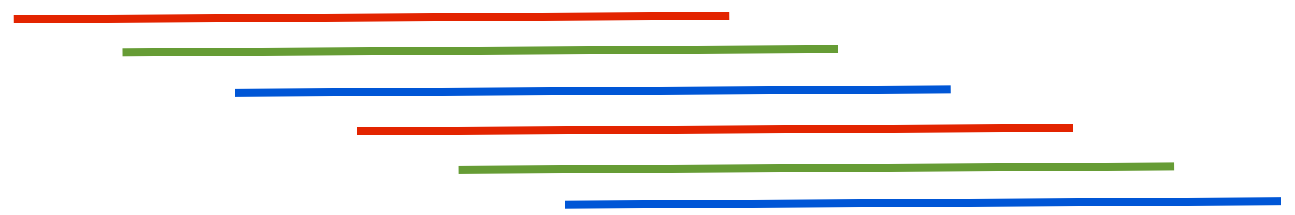
- Original suffix array paper suggested an $O(m \log m)$ algorithm
  - Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.
- Other popular $O(m \log m)$ algorithms have been suggested
  - Larsson NJ, Sadakane K. Faster suffix sorting. 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.

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



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

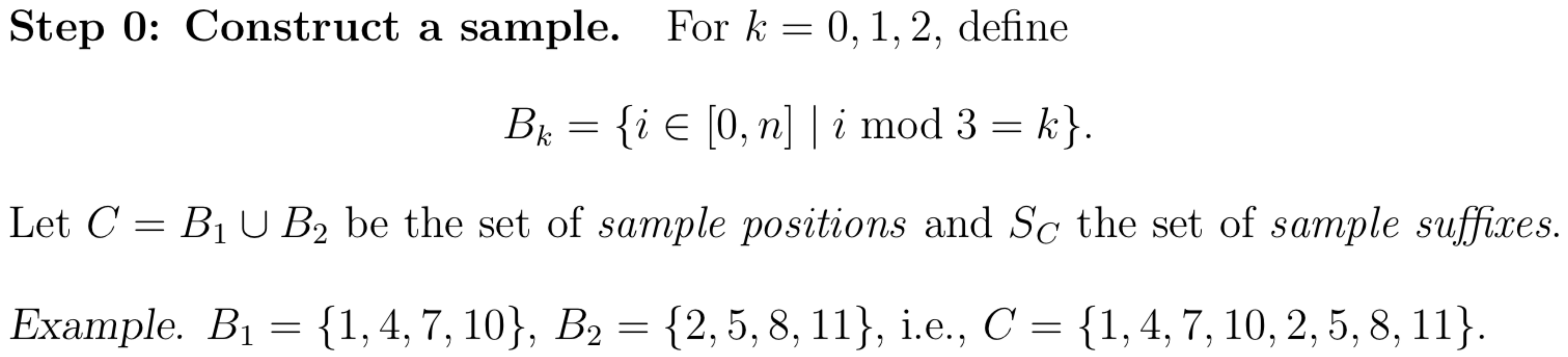
Basic Outline:

- 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.

*slide courtesy of Carl Kingsford

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

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

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: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \dots$ 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. $R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}]$.

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

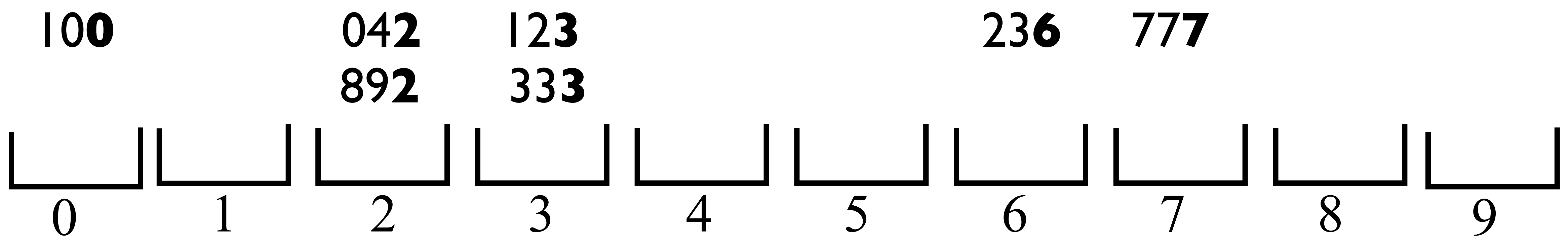
To sort the suffixes of R, first radix sort the characters of R and rename them with their ranks to obtain the string R'. If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of R' using Algorithm DC3.

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.

Interlude: Radix Sort

- O(n)-time sort for n items when items can be divided into constant # of digits.
- Put into buckets based on least-significant digit, flatten, repeat with next-most significant digit, etc.
- Example items: 10**0** 12**3** 04**2** 33**3** 77**7** 89**2** 23**6**



- # of passes = # of digits
- Each pass goes through the numbers once.

*slide courtesy of Carl Kingsford

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

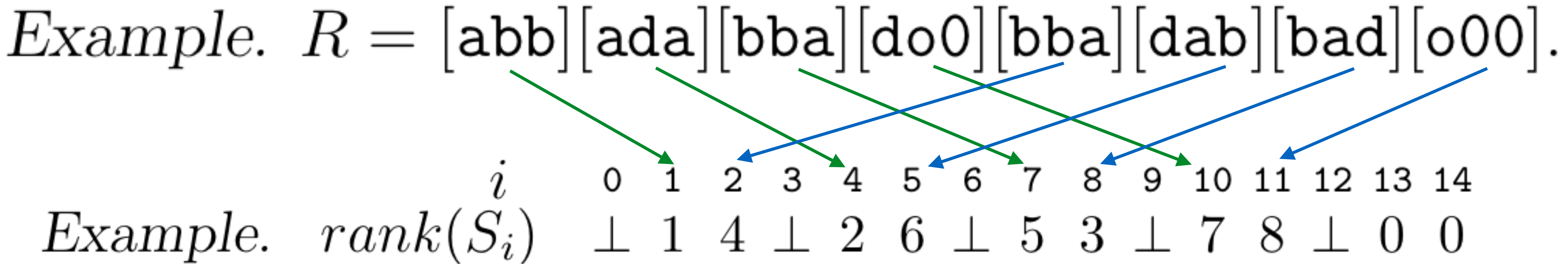
Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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.

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

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

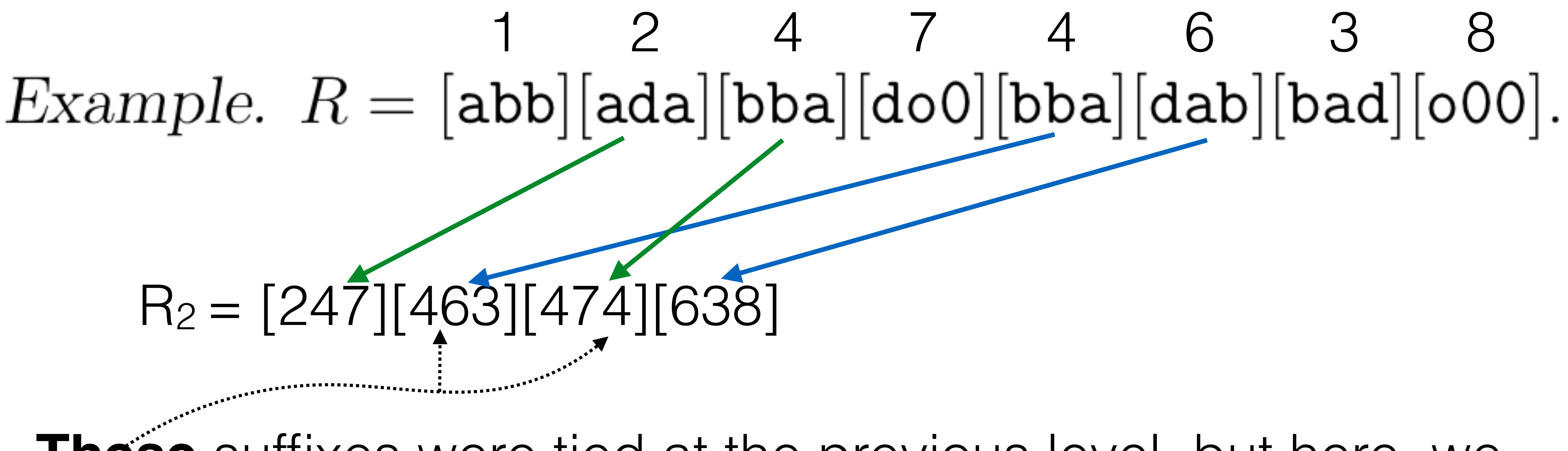
Example. $R = [\mathrm{abb}][\mathrm{ada}][\mathrm{bba}][\mathrm{do0}][\mathrm{bba}][\mathrm{dab}][\mathrm{bad}][\mathrm{o00}]$.

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

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



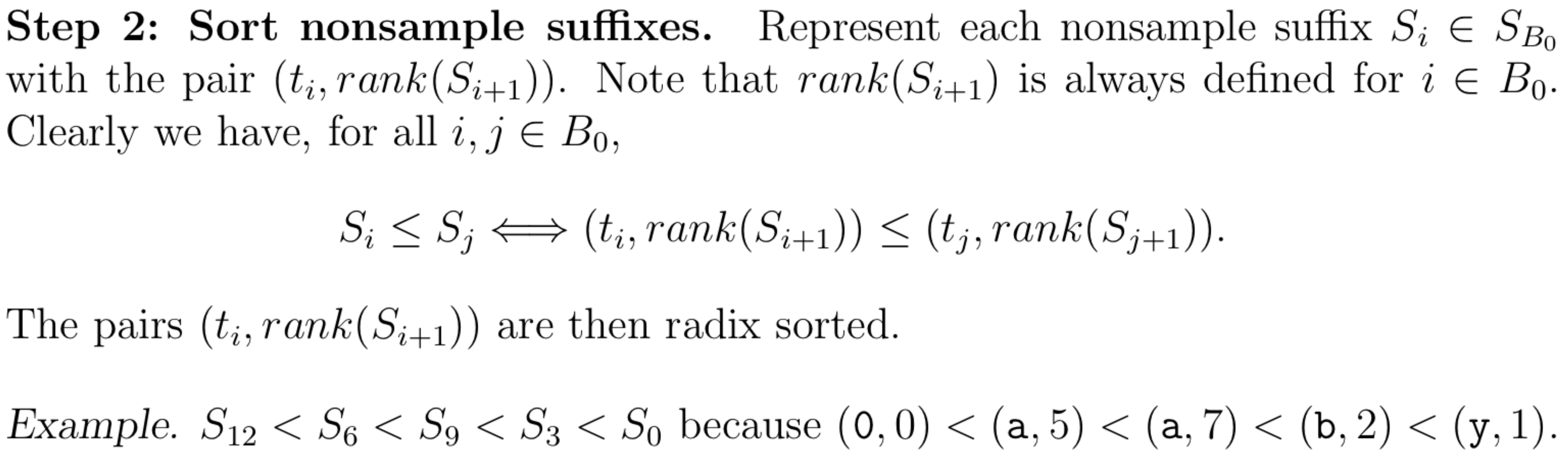
$$\begin{array}{rcccccccc} & 1 & 2 & 4 & 7 & 4 & 6 & 3 & 8 \\ \textit{Example.}\ R = & [\mathtt{abb}] & [\mathtt{ada}] & [\mathtt{bba}] & [\mathtt{do0}] & [\mathtt{bba}] & [\mathtt{dab}] & [\mathtt{bad}] & [\mathtt{o00}]. \end{array}$$

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

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 $i, j \in B_0$,

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

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:

$$
\begin{aligned}
i \in B_1 : \quad & S_i \leq S_j \iff (\overset{S_{B1}}{t_i}, \overset{S_{B2}}{rank}(S_{i+1})) \leq (\overset{S_{B0}}{t_j}, rank\overset{S_{B1}}{(S_{j+1})}) \\
i \in B_2 : \quad & S_i \leq S_j \iff (\underset{S_{B2}}{t_i}, \underset{S_{B0}}{t_{i+1}}, rank\underset{S_{B1}}{(S_{i+2})}) \leq (\underset{S_{B0}}{t_j}, \underset{S_{B1}}{t_{j+1}}, rank\underset{S_{B2}}{(S_{j+2})})
\end{aligned}
$$

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: $T(n) \le cn + T(2n/3)$ for some c.
- Guess: $T(n) \le 3cn$
- Induction step: assume that is true for all $i < n$.
- $T(n) \le cn + 3c(2n/3) = cn + 2cn = 3cn$ □

*slide courtesy of Carl Kingsford