Simple Linear Work `Suffix Array` Construction

Juha Karkkainen and Peter Sanders

Max-Planck-Institut fur Informatik

{juha,sanders}@mpi-sb.mpg.de.

Presented by

sualeh asif

sualeh@mit.edu

outline

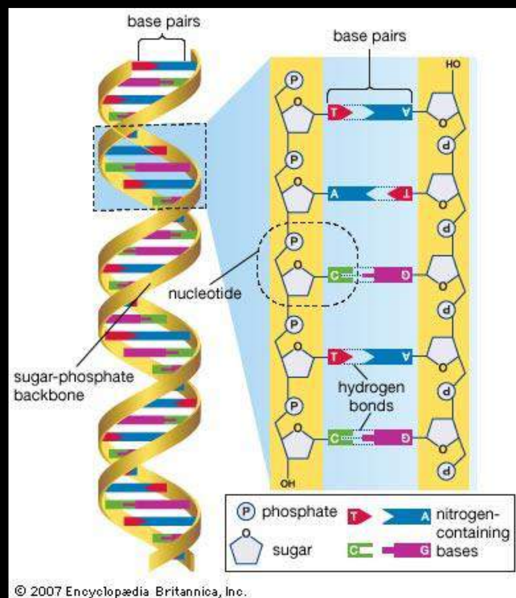
1. motivation
2. a primer on suffix arrays
3. the ***skew algorithm***: a linear work algorithm
4. the *skew algorithm* in other models
5. questions

bonus: future of suffix arrays and fast implementations

1. motivation

motivating problem

The *Longest Common Substring* problem:



Chromo-some #	Length (mm)	Number of Base Pairs	Coding Genes (Protein)	Percentage	
				of Total	Accum.
1	85	248 956 422	2 058	8.06	8.06
2	83	242 193 529	1 309	7.84	15.90
3	67	198 295 559	1 078	6.42	22.32
4	65	190 214 555	752	6.16	28.48
5	62	181 538 259	876	5.88	34.36
6	58	170 805 979	1 048	5.53	39.89
7	54	159 345 973	989	5.16	45.05
X	53	156 040 895	842	5.05	50.11
8	50	145 138 636	677	4.70	54.80
9	48	138 394 717	786	4.48	59.29
11	46	135 086 622	1 298	4.37	63.66
10	46	133 797 422	733	4.33	67.99
12	45	133 275 309	1 034	4.32	72.31
13	39	114 364 328	327	3.70	76.01
14	36	107 043 718	830	3.47	79.48
15	35	101 991 189	613	3.30	82.78
16	31	90 338 345	873	2.93	85.71
17	28	83 257 441	1 197	2.70	88.40
18	27	80 373 285	270	2.60	91.00
20	21	64 444 167	544	2.09	93.09
19	20	58 617 616	1 472	1.90	94.99
Y	20	57 227 415	71	1.85	96.8415
22	17	50 818 468	488	1.65	98.49
21	16	46 709 983	234	1.51	99.9995
mtDNA	0.0054	16 569	13	0.0005	**100.00**
Totals	**1 052**	**3 088 286 401**	**20 412**		

Find the greatest common substring between two strings S, T.

motivating problem

The *Longest Common Substring* problem:

- easy if we can efficiently maintain all substrings

motivating problem

The *Longest Common Substring* problem:

- easy if we can efficiently maintain all substrings
    - we can just check for collisions of similar substrings
- motivates us to have an easily searchable list.

motivating problem

The *Longest Common Substring* problem:

- easy if we can efficiently maintain all substrings
    - we can just check for collisions of similar substrings
- motivates us to have an easily searchable list.
    - keep it *sorted* to allow for easy searches.

motivating problem

The *Longest Common Substring* problem:

- easy if we can efficiently maintain all substrings
    - we can just check for collisions of similar substrings
- motivates us to have an easily searchable list.
    - keep it *sorted* to allow for easy searches.
    - boom! we have a suffix array

motivating problem

The *Longest Common Substring* problem:

- easy if we can efficiently maintain all substrings
    - we can just check for collisions of similar substrings
- motivates us to have an easily searchable list.
    - keep it *sorted* to allow for easy searches.
    - boom! we have a suffix array

(note: suffix trees and suffix arrays are interchangeable for applications)

motivating applications

application 1: bioinformatics

motivating applications

application 1: bioinformatics
application 2: text processing

motivating applications

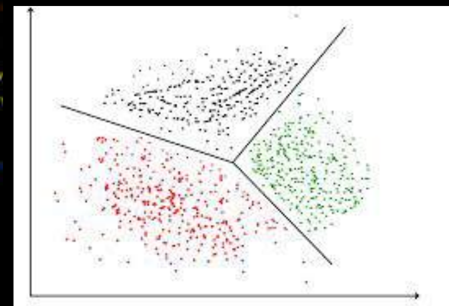
application 1: bioinformatics
application 2: text processing
application 3: data compression

Lempel-Ziv-Welch
Compression
Algorithm -
Tutorial
CSLEARNING101
thisisthe

motivating applications

application 1: bioinformatics
application 2: text processing
application 3: data compression
application 4: clustering

Lempel-Ziv-Welch Compression Algorithm - Tutorial
CSLEARNING101



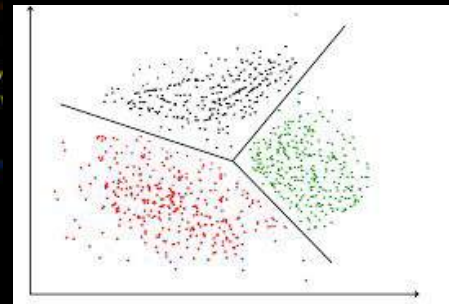
motivating applications

application 1: bioinformatics
application 2: text processing
application 3: data compression
application 4: clustering
application 5: … basically anything with strings

Lempel-Ziv-Welch Compression Algorithm - Tutorial

CSLEARNING101

thisisthe



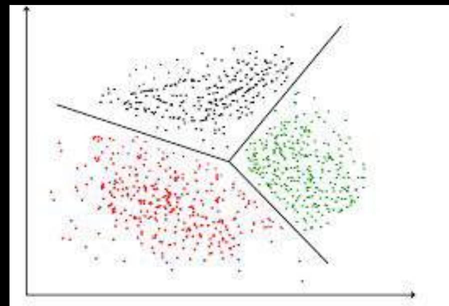
motivating applications

application 1: bioinformatics
application 2: text processing
application 3: data compression
application 4: clustering
application 6: oh and coding interviews :)

thisisthe

Lempel-Ziv-Welch
Compression
Algorithm -
Tutorial

CSLEARNING101



2. a primer on suffix arrays

2. a primer on suffix arrays

(side note: these are different from suffix trees)

a motivating example

lets consider the suffixes of our favorite word:

a motivating example

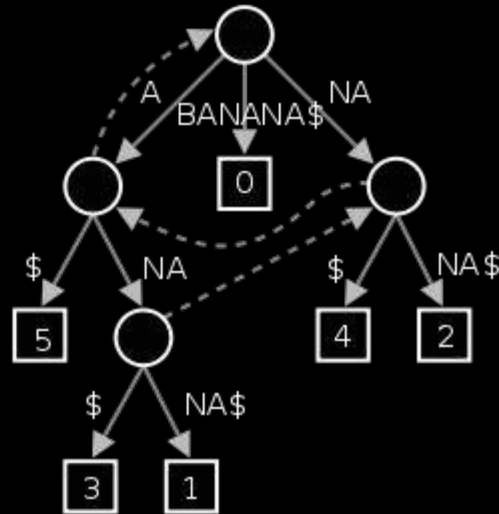
lets consider the suffixes of our favorite word:

M I S S I S S I P P I

(transition to live work on the ipad)

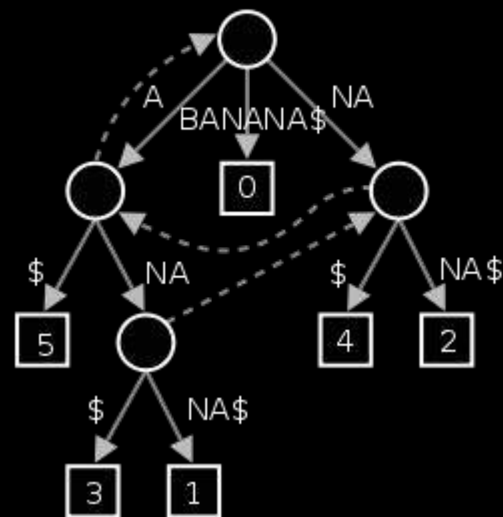
a simple algorithm

- Just do a depth-first traversal of the suffix tree:



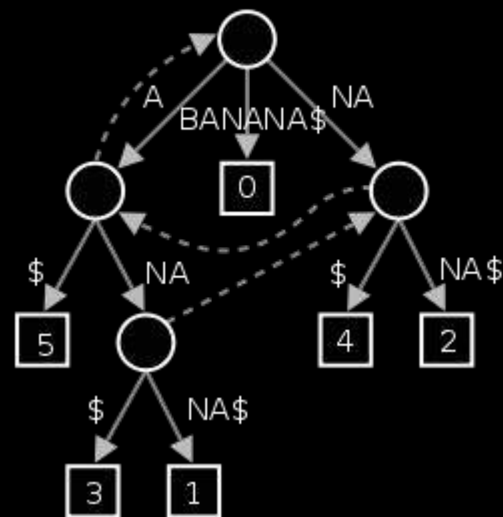
a simple algorithm

- Just do a depth-first traversal of the suffix tree:
    - this is linear work!
      (with a fast suffix tree algorithm)



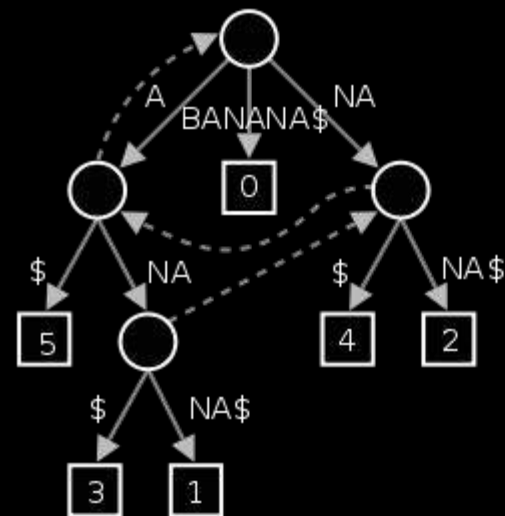
a simple algorithm

- Just do a depth-first traversal of the suffix tree:
    - this is linear work!
      (with a fast suffix tree algorithm)
- cons:
    - memory :(
        - we have to construct a full tree
    - tricky parallelism



a longer look at the simple algorithm

- linear work suffix tree construction:

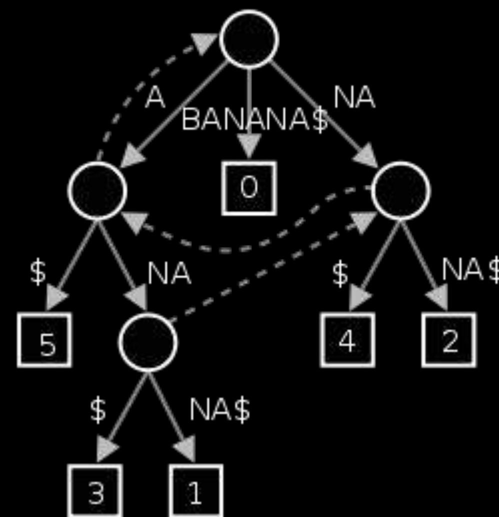


a longer look at the simple algorithm

- linear work suffix tree construction:

Algorithm 1: Farach's linear-time suffix tree construction:

1. Construct the suffix tree of the suffixes starting at odd positions. This is done by reduction to the suffix tree construction of a string of half the length, which is solved recursively.
2. Construct the suffix tree of the remaining suffixes using the result of the first step.
3. Merge the two suffix trees into one.

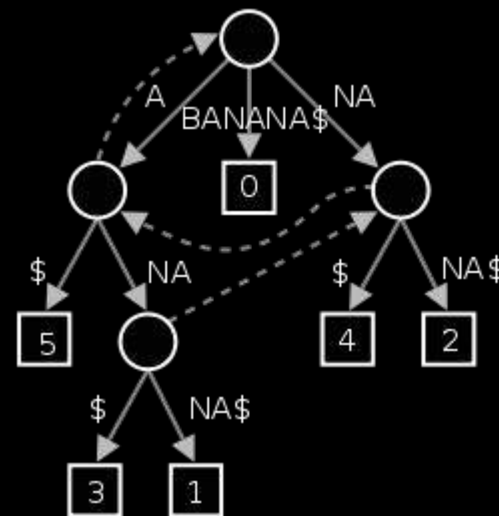


a longer look at the simple algorithm

- linear work suffix tree construction:

Algorithm 1: Farach's linear-time suffix tree construction:

1. Construct the suffix tree of the suffixes starting at odd positions. This is done by reduction to the suffix tree construction of a string of half the length, which is solved recursively.
2. Construct the suffix tree of the remaining suffixes using the result of the first step.
3. Merge the two suffix trees into one.



IDEA: use a similar merging based approach with 3 buckets

3. the *skew algorithm*

goal: a linear time algorithm

goal: accomplished :)

4. the *skew algorithm* in other models

Important interesting models

Important interesting models

model 1: external memory

model of computation	complexity	alphabet
External Memory [38] D disks, block size B, fast memory of size M	$\mathcal{O}\left(\frac{n}{DB}\log_{\frac{M}{B}}\frac{n}{B}\right)$ I/Os $\mathcal{O}\left(n\log_{\frac{M}{B}}\frac{n}{B}\right)$ internal work	integer

Important interesting models

model 1: external memory
model 2: cache model

model of computation	complexity	alphabet
Cache Oblivious [15]	$\mathcal{O}\left(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B}\right)$ cache faults	general

Important interesting models

model 1: external memory
model 2: cache model
model 3: BSP

model of computation	complexity	alphabet
BSP [37] P processors h-relation in time $L + gh$	$\mathcal{O}\left(\frac{n \log n}{P} + L \log^2 P + \frac{gn \log n}{P \log(n/P)}\right)$ time	general
$P = \mathcal{O}(n^{1-\epsilon})$ processors	$\mathcal{O}(n/P + L \log^2 P + gn/P)$ time	integer

Important interesting models

model 1: external memory
model 2: cache model
model 3: BSP
model 4: EREW-PRAM

model of computation	complexity	alphabet
EREW-PRAM [25]	$\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n \log n)$ work	general

5. questions

bonus. future of suffix arrays and fast implementations

bonus. future of suffix arrays and fast implementations

things that happen after this paper

1. Li, Li & Huo gave an in-place linear work algorithm

things that happen after this paper

1. Li, Li & Huo gave an in-place linear work algorithm
2. first fast in-practice algorithm: Nong, Zhang & Chin (2009)
    a. <100 Lines of Code
    b. can be used to construct LC array
    c. fast careful implementation by Yuta Mori
    https://sites.google.com/site/yuta256/sais

things that happen after this paper

1. Li, Li & Huo gave an in-place linear work algorithm
2. first fast in-practice algorithm: Nong, Zhang & Chin (2009)
    a. <100 Lines of Code
    b. can be used to construct LC array
    c. fast careful implementation by Yuta Mori
       https://sites.google.com/site/yuta256/sais
3. the work has been extended to Generalized Suffix Arrays
4. Other approaches: Prefix Doubling, Induced Copying

6. open discussion

a motivating example

MISSISSIPPI

a motivating example

MISSISSIPPI

suffixes:

a motivating example

MISSISSIPPI$

suffixes:

- mississippi$
- ississippi$
- ssissippi$
- sissippi$
- issippi$
- ssippi$
- sippi$
- ippi$
- ppi$
- pi$
- i$
- $

a motivating example

MISSISSIPPI$

String S

suffixes $S_i = S[i:n)$

suffixes:

- mississippi$
- ississippi$
- ssissippi$
- sissippi$
- issippi$
- ssippi$
- sippi$
- ippi$
- ppi$
- pi$
- i$
- $

a motivating example

MISSISSIPPI$

String S

suffixes $S_i = S[i:n)$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

a motivating example

MISSISSIPPI$ String S

suffixes $S_i = S[i:n)$

suffixes:

	i
- mississippi$	0
- ississippi$	1
- ssissippi$	2
- sissippi$	3
- issippi$	4
- ssippi$	5
- sippi$	6
- ippi$	7
- ppi$	8
- pi$	9
- i$	10
- $	11

sort by $S_i[0]$

a motivating example

MISSISSIPPI$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

suffix	i
$	11

a motivating example

MISSISSIPPI$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

suffix	i
$	11
i$	
ippi$	
issippi$	
ississippi$	

a motivating example

MISSISSIPPI$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

sorted suffix	i
$	11
i$	
ippi$	
issippi$	
ississippi$	
mississippi$	
pi$	
ppi$	

a motivating example

MISSISSIPPI$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

suffix	i
$	11
i$	10
ippi$	7
issippi$	4
ississippi$	1
mississippi$	0
pi$	9
ppi$	8
sippi$	6
sissippi$	3
ssippi$	5
ssissippi$	2

a motivating example

MISSISSIPPI$

suffixes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

suffix	i
$	11
i$	10
ippi$	7
issippi$	4
ississippi$	1
mississippi$	0
pi$	9
ppi$	8
sippi$	6
sissippi$	3
ssippi$	5
ssissippi$	2

a motivating example

MISSISSIPPI$ $\Rightarrow$ SA = [10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]

s xes:

suffix	i
mississippi$	0
ississippi$	1
ssissippi$	2
sissippi$	3
issippi$	4
ssippi$	5
sippi$	6
ippi$	7
ppi$	8
pi$	9
i$	10
$	11

sort by $S_i[0]$

suffix	i
$	11
i$	10
ippi$	7
issippi$	4
ississippi$	1
mississippi$	0
pi$	9
ppi$	8
sippi$	6
sissippi$	3
ssippi$	5
ssissippi$	2

The skew algorithm

The skew algorithm

Notation:

String s

$[a,b] = \{a, a+1, \ldots, b\}$

$s[a,b] = \{s[a], s[a+1], \ldots, s[b-1], s[b]\}$

$s[a,b) = \{s[a], s[a+1], \ldots, s[b-1]\}$

The skew algorithm

Notation:

String s ← (s numbered from 0)

$[a,b] = \{a, a+1, \dots, b\}$

$s[a,b] = \{s[a], s[a+1], \dots, s[b-1], s[b]\}$

$s[a,b) = \{s[a], s[a+1], \dots, s[b-1]\}$

Alphabet $\Sigma = [1,n]$

$s[i] \in \Sigma$ for $i \in [0,n)$

Goal: construct suffix array SA.

The skew algorithm

Notation:

String s ← (s numbered from 0)

$[a,b] = \{a, a+1, \ldots b\}$

$s[a,b] = \{s[a], s[a+1], \ldots, s[b-1], s[b]\}$

$s[a,b) = \{s[a], s[a+1], \ldots s[b-1]\}$

Alphabet $\Sigma = [1,n]$

$s[i] \in \Sigma$ for $i \in [0,n)$

(assume $n \equiv 0 \mod 3$ for simplicity)

The skew algorithm

Plan: Fix the problems with Farach's argument

1)

2)

3)

The skew algorithm

Plan: Fix the problems with Farach's argument

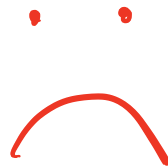
1) Construct SA_1 for indices $i \not\equiv 0 \mod 2$
2) Use step 1 to construct SA_0 for $i \equiv 0 \mod 2$
3) Merge : $SA = \text{merge}(SA_0, SA_1)$

The skew algorithm

Plan: Fix the problems with Farach's argument

1) Construct SA_1 for indices $i \not\equiv 0 \mod 2$
2) Use step 1 to construct SA_0 for $i \equiv 0 \mod 2$
3) Merge : $SA = \text{merge}(SA_0, SA_1)$

sadly this doesn't work

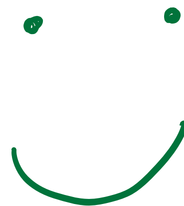


The skew algorithm

Plan: Fix the problems with Farach's argument

1) Construct SA_{12} for indices $i \not\equiv 0 \mod \cancel{2}\ 3$
2) Use step 1 to construct SA_0 for $i \equiv 0 \mod \cancel{2}\ 3$
3) Merge : $SA = \text{merge}(SA_0, SA_{12})$

This works.



The skew algorithm

Plan: Fix the problems with Farach's argument

1) Construct SA_{12} for indices $i \not\equiv 0 \mod \cancel{2}\ 3$
2) Use step 1 to construct SA_0 for $i \equiv 0 \mod \cancel{2}\ 3$
3) Merge : $SA = \text{merge}(SA_0, SA_{12})$

Intuition: just enough space for colliding first characters to be quickly verified in merge!

This works

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_{12})$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

The skew algorithm

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \mod 3$
2) construct SA_0 for indices $i \equiv 0 \mod 3$
3) $SA = \text{merge}(SA_0, SA_{12})$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

Example: mississippi$$ $n = 12$

$$I_{12} = \{1, 2, 4, 5, 7, 8, 10\}$$

$$T = \{ \text{iss}, \text{iss}, \text{ipp}, \text{i\$\$}, \text{ssi}, \text{ssi}, \text{ppi}$$

mississippi

mississippi

The skew algorithm

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Example: mississippi\$\$ $\quad n = 12$

$$I_{12} = \{1, 2, 4, 5, 7, 8, 10\}$$

$$T = \{ \text{iss, iss, ipp, i\$\$,} \quad \text{ssi, ssi, ppi} \}$$

mississippi

mississippi

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

* Radix sort on T to sort triples.
  Assign each triple a unique name.

The skew algorithm

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \mod 3$
2) construct SA_0 for indices $i \equiv 0 \mod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples
  $T = \{ s[i : i+2] \mid i \in I_{12} \}$
* Radix sort on T to sort triples.
  Assign each triple a unique name.

sorted(T) = { iss, iss, ipp, i$$ ssi, ssi, ppi } =

i$$
ipp
iss
iss
ppi
ssi
ssi

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \mod 3$
2) construct SA_0 for indices $i \equiv 0 \mod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

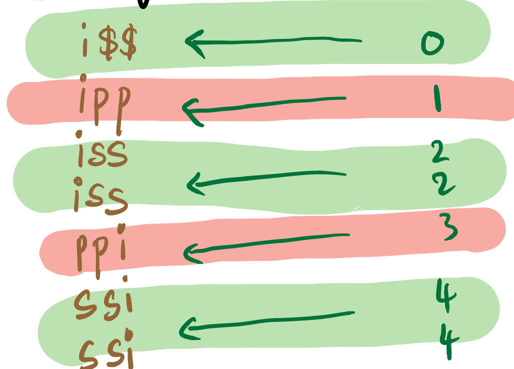
* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples
  $T = \{ s[i : i+2] \mid i \in I_{12} \}$
* Radix sort on T to sort triples.
  Assign each triple a unique name.

sorted(T) = { iss, iss, ipp, i$$
ssi, ssi, ppi }

⇓

$s^{12} = [3, 3, 2, 1, 5, 5, 4]$

=



The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \dots\}$
* Consider triples
  $T = \{ s[i : i+2] \mid i \in I_{12} \}$
* Radix sort on T to sort triples. Assign each triple a unique name.

* if sorted(T) is not unique, recurse on s_{12} to get SA_{12}

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

* Radix sort on T to sort triples. Assign each triple a unique name.

* if sorted(T) is not unique, recurse on s_{12} to get SA_{12}

$$s_{12} = [3,3,2,1,5,5,4] \implies SA_{12} = [3,2,1,0,6,5,4]$$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

* Radix sort on T to sort triples. Assign each triple a unique name.

* if sorted(T) is not unique, recurse on s_{12} to get SA_{12}

$$s_{12} = [3, 3, 2, 1, 5, 5, 4] \implies SA_{12} = [3, 2, 1, 0, 6, 5, 4]$$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 1: Construct SA_{12} for $i \not\equiv_3 0$

* Let $I_{12} = \{1, 2, 4, 5, \ldots\}$
* Consider triples

$$T = \{ s[i:i+2] \mid i \in I_{12} \}$$

* Radix sort on T to sort triples. Assign each triple a unique name.
* if sorted(T) is not unique, recurse on s_{12} to get SA_{12}
* fix up SA_{12} from triples → original

$SA_{12} = [3, 2, 1, 0, 6, 5, 4] \longrightarrow [10, 7, 4, 1, 8, 5, 2]$

The skew algorithm

Step 2: Construct SA_0 for $i \equiv_3 0$

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_{12})$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 2: Construct SA_0 for $i \equiv_3 0$

Easy:

* We want to sort $\{S_0, S_3, S_6 \ldots\}$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 2: Construct SA_0 for $i \equiv_3 0$

Easy:

* We want to sort $\{S_0, S_3, S_6 \ldots\}$
* equivalently sort $\{(S[0], S_1), (S[3], S_4), \ldots\}$

The skew algorithm

Plan:
1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 2: Construct SA_0 for $i \equiv_3 0$

Easy:

* We want to sort $\{S_0, S_3, S_6 \dots\}$
* equivalently sort $\{(S[0], S_1), (S[3], S_4), \dots\}$

example: mississippi

$$\text{sort}\left\{ \begin{array}{l} (S[0]=m,\ S_1) \\ (S[3]=s,\ S_4) \\ (S[6]=s,\ S_7) \\ (S[9]=p,\ S_{10}) \end{array} \right\} = \{S_0, S_9, S_6, S_3\}$$

because $S_4 < S_7$ ($4 = S_4$, $8 = S_7$)

The skew algorithm

Step 3: Merge SA_0, SA_{12}

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 3: Merge SA_0, SA_{12}

Also not bad:

* Do a merge step (from say merge sort)
* Need to compare S_i, S_j

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 3: Merge SA_0, SA_{12}

Also not bad:

* Do a merge step (from say merge sort)
* Need to compare S_i, S_j

cool: $i, j \equiv 0 \Rightarrow$ we know from SA_0

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \mod 3$
2) construct SA_0 for indices $i \equiv 0 \mod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Step 3: Merge SA_0, SA_{12}

Also not bad:

* Do a merge step (from say merge sort)
* Need to compare S_i, S_j

cool

- $i, j \equiv 0 \implies$ we know from SA_0
- $i, j \not\equiv 0 \implies$ we know from SA_{12}

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_{12})$

Step 3: Merge SA_0, SA_{12}

Also not bad:

* Do a merge step (from say merge sort)
* Need to compare S_i, S_j

cool:

- $i, j \equiv 0 \implies$ we know from SA_0
- $i, j \not\equiv 0 \implies$ we know from SA_{12}
- $i \equiv 1, j \equiv 0 \implies S_i = (s[i], S_{i+1})$, $S_j = (s[j], S_{j+1})$ → default to above
- $i \equiv 2, j \equiv 0 \implies S_i = (s[i], s[i+1], S_{i+2})$ → default to above

The skew algorithm

Plan:
1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_{1})$

Step 3: Merge SA_0, SA_{12}

Also not bad:

* Do a merge step (from say merge sort)
* Need to compare S_i, S_j

$$\text{Merge}\begin{pmatrix} SA_{12} = [10, 7, 4, 1, 8, 5, 2] \\ SA_0 = [0, 9, 6, 3] \end{pmatrix}$$

$$\parallel$$

$$SA = [10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2]$$

:)

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \mod 3$
2) construct SA_0 for indices $i \equiv 0 \mod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Running Time:

$$T(n) = 2T\left(\left\lceil \frac{2n}{3} \right\rceil\right) + O(n)$$

The skew algorithm

Plan:

1) Construct SA_{12} for indices $i \not\equiv 0 \bmod 3$
2) construct SA_0 for indices $i \equiv 0 \bmod 3$
3) $SA = \text{merge}(SA_0, SA_1)$

Running Time:

$$T(n) = 2T\left(\left\lceil \frac{2n}{3} \right\rceil\right) + O(n)$$

$$T(n) = O(n)$$