KS Algorithm

(slides by Tomasz Waleń)

MIM UW

$\mathrm{RANK} = \mathrm{SA}^{-1}$

$$\mathrm{RANK}[p] = i \text{ iff } \mathrm{SA}[i] = p$$

i	$SA[i]$	$y[SA[i]\ldots]$
0	10	i
1	7	ippi
2	4	issippi
3	1	ississippi
4	0	mississippi
5	9	pi
6	8	ppi
7	6	sippi
8	3	sissippi
9	5	ssippi
10	2	ssissippi

$O(n)$-time Algorithm Computing SA

- We aim at time complexity $O(n)$
- We assume integer alphabet
- The algorithm will be recursive
- Its running time $T(n)$ for a string of length n will satisfy the formula:

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

- Hence, $T(n) = O(n)$.

KS Algorithm

Juha Kärkkäinen, Peter Sanders,
"Simple linear work suffix array construction"
ICALP 2003
~530 citations + ~350 (journal version: J. ACM 2006)

Tit bit:
the paper contains an implementation of the algorithm (~50 lines of code in C)

KS Algorithm: Outline

Algorithm 1: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$
2 generate a string y' that represents the suffixes from S_1 and S_2
  ▷ $|y'| \approx 2n/3$
3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$
4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2
5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0
6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`
7 **return** SA

KS Algorithm: Time Complexity

Lemma
KS algorithm works in $O(n)$ time.

Proof.
Assuming that we can compute the string y' and the arrays $\mathrm{SA}_{1,2}$ and SA_0, and merge them in $O(n)$ time, we obtain the following recursive formula:

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

which indeed yields $T(n) = O(n)$. □

KS Algorithm: Partitioning of Suffixes

Algorithm 2: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
▷ $|S_i| \approx n/3$

2 generate a string y' that represents the suffixes from S_1 and S_2
▷ $|y'| \approx 2n/3$

3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$

4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2

5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0

6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`

7 **return** SA

Partitioning of Suffixes into S_0, S_1 and S_2

Technical detail

Depending on $n \bmod 3$, y is padded with 1, 2 or 3 characters \$ (we assume that $\$ < a$ for every $a \in \Sigma$) so that 3 divides $|y|$.

Partitioning of Suffixes into S_0, S_1 and S_2

Technical detail

Depending on $n \bmod 3$, y is padded with 1, 2 or 3 characters \$ (we assume that $\$ < a$ for every $a \in \Sigma$) so that 3 divides $|y|$.

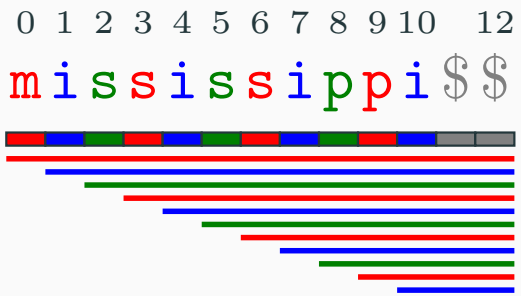
We define:

$$S_j = \{i : i < n \text{ and } i \bmod 3 = j\}$$

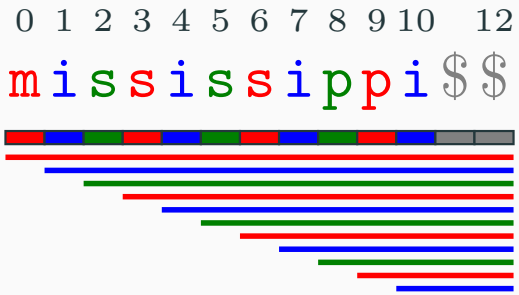
$$S_0 = \{0, 3, 6, 9\}$$

$$S_1 = \{1, 4, 7, 10\}$$

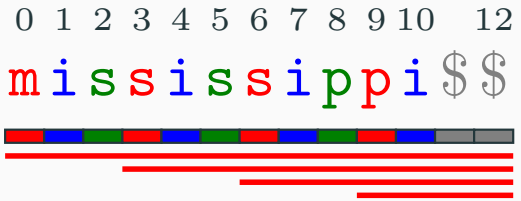
$$S_2 = \{2, 5, 8\}$$



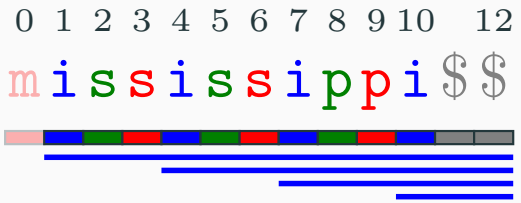
Partitioning of suffixes into S_0, S_1 and S_2



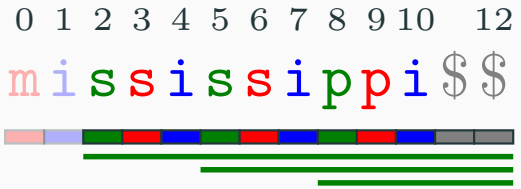
$S_0 = \{0, 3, 6, 9\}$



$S_1 = \{1, 4, 7, 10\}$



$S_2 = \{2, 5, 8\}$



KS Algorithm: Auxiliary String y'

Algorithm 3: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$

2 generate a string y' that represents the suffixes from S_1 and S_2
  ▷ $|y'| \approx 2n/3$

3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$

4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2

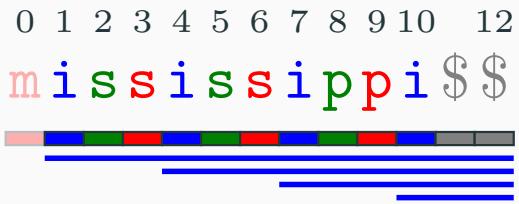
5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0

6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`

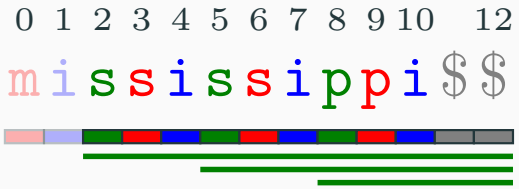
7 **return** SA

String y' that represents the suffixes in S_1 and S_2

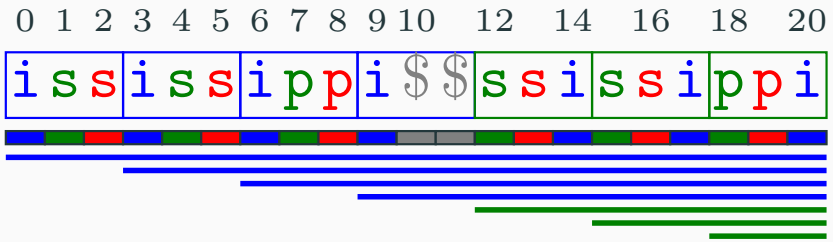
$S_1 = \{1, 4, 7, 10\}$



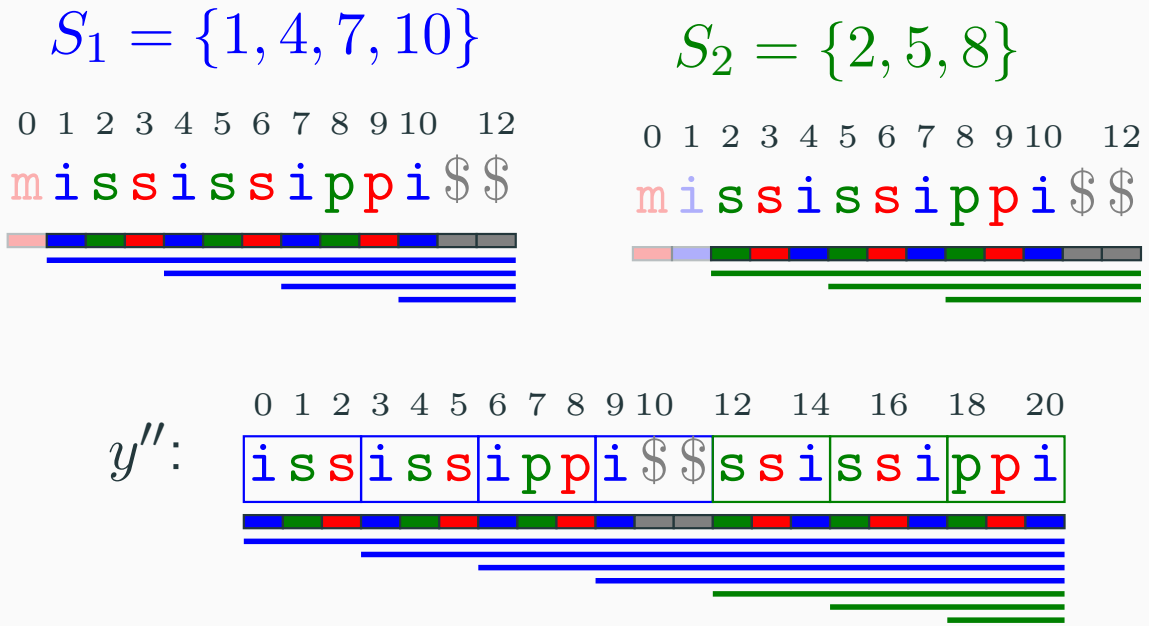
$S_2 = \{2, 5, 8\}$



y'':



String y' that represents the suffixes in S_1 and S_2



Unfortunately, the string y'' is too long (its length is $\approx 2n$ rather than $2n/3$).

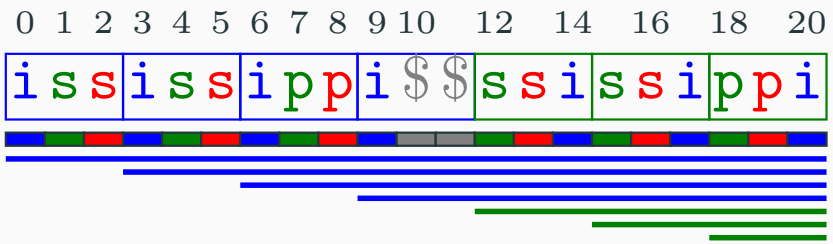
String y' that represents the suffixes in S_1 and S_2

Why not compress y''?

String y' that represents the suffixes in S_1 and S_2

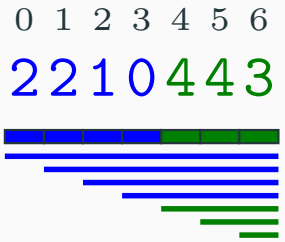
Why not compress y''?

y'':



i$$ →0 ipp →1 iss →2 ppi →3 ssi →4

y':



$|y'| \approx 2n/3$. **Yay!**

KS Algorithm: $\mathrm{SA}_{1,2}$

Algorithm 4: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$
2 generate a string y' that represents the suffixes from S_1 and S_2
  ▷ $|y'| \approx 2n/3$
3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$
4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2
5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0
6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`
7 **return** SA

$\mathrm{SA}_{1,2}$ Array

Let us notice that the construction of y' from y is "reversible", i.e., for any suffix from S_1 and S_2 we can find a corresponding suffix in y.

y': 0 1 2 3 4 5 6
2 2 1 0 4 4 3

y: 0 1 2 3 4 5 6 7 8 9 10 12
m i s s i s s i p p i $ $

i	$\mathrm{SA}_{y'}[i]$	suf. in y'	suf. in y	$\mathrm{SA}_{1,2}[i]$
0	3	0443	i$$	10
1	2	10443	ippi$$	7
2	1	210443	issippi$$	4
3	0	2210443	ississippi$$	1
4	6	3	ppi$$	8
5	5	43	ssippi$$	5
6	4	443	ssissippi$$	2

KS Algorithm: SA_0

Algorithm 5: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$
2 generate a string y' that represents the suffixes from S_1 and S_2
  ▷ $|y'| \approx 2n/3$
3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$
4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2
5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0
6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`
7 **return** SA

SA_0 Array

Unfortunately, here a recursive call is not possible. We need to use $\mathrm{SA}_{1,2}$ to compute SA_0.
Note that every suffix i from SA_0 can be written as:

$$(y[i], \quad y[i+1\ldots])$$

SA_0 Array

Unfortunately, here a recursive call is not possible. We need to use $\mathrm{SA}_{1,2}$ to compute SA_0.
Note that every suffix i from SA_0 can be written as:

$$(y[i], \quad y[i+1\ldots])$$

Since $i+1 \in S_1$, the rank of this suffix is known:

$$(y[i], \quad \mathrm{RANK}_{1,2}[i+1])$$

Such pairs are small and can be radix sorted in linear time!
An ordering of the pairs gives SA_0.

KS Algorithm: Merging SA_0 and $\mathrm{SA}_{1,2}$

Algorithm 6: ComputeSA

Input: string y, $|y| = n$

Output: suffix array SA of y

1 partition the suffixes of y into three categories S_0, S_1, S_2
  ▷ $|S_i| \approx n/3$
2 generate a string y' that represents the suffixes from S_1 and S_2
  ▷ $|y'| \approx 2n/3$
3 $\mathrm{SA}_{y'} = \mathrm{ComputeSA}(y')$
4 $\mathrm{SA}_{1,2}$ = compute from $\mathrm{SA}_{y'}$ the lex order of suffixes from S_1 and S_2
5 SA_0 = compute from $\mathrm{SA}_{y'}$ and y the lex order of suffixes from S_0
6 SA = merge $\mathrm{SA}_{1,2}$ and SA_0 ▷ `tricky`
7 **return** SA

Merging SA_0 and $\mathrm{SA}_{1,2}$

i	$\mathrm{SA}_0[i]$	suf. in y
0	0	`mississippi$$`
1	9	`pi$$`
2	6	`sippi$$`
3	3	`sissippi$$`

i	$\mathrm{SA}_{1,2}[i]$	suf. in y
0	10	`i$$`
1	7	`ippi$$`
2	4	`issippi$$`
3	1	`ississippi$$`
4	8	`ppi$$`
5	5	`ssippi$$`
6	2	`ssissippi$$`

Merging SA_0 and $\mathrm{SA}_{1,2}$

Both subsets of suffixes are sorted. It suffices to show that one can compare in $O(1)$ time suffixes $i \in \mathrm{SA}_0$ and $j \in \mathrm{SA}_{1,2}$.

Merging SA_0 and $\mathrm{SA}_{1,2}$

Both subsets of suffixes are sorted. It suffices to show that one can compare in $O(1)$ time suffixes $i \in \mathrm{SA}_0$ and $j \in \mathrm{SA}_{1,2}$.

Informally, the problem is that the suffixes come from different worlds:

$$\mathrm{cmp}(\mathsf{apple}, \mathsf{poem})$$

Merging SA_0 and $\mathrm{SA}_{1,2}$

$$\mathrm{cmp}(i \in S_0,\ j \in S_1 \cup S_2)$$

There are two cases:

- $j \in S_1$: compare
  $(y[i],\ \ \mathrm{RANK}_{1,2}[i+1])$ and $(y[j],\ \ \mathrm{RANK}_{1,2}[j+1])$
  $(i+1 \in S_1,\ \ j+1 \in S_2)$

Merging SA_0 and $\mathrm{SA}_{1,2}$

$$\mathrm{cmp}(i \in S_0,\ j \in S_1 \cup S_2)$$

There are two cases:

- $j \in S_1$: compare
  $(y[i],\ \ \mathrm{RANK}_{1,2}[i+1])$ and $(y[j],\ \ \mathrm{RANK}_{1,2}[j+1])$
  $(i+1 \in S_1,\ \ j+1 \in S_2)$

- $j \in S_2$: compare
  $(y[i],\ \ y[i+1],\ \ \mathrm{RANK}_{1,2}[i+2])$ and
  $(y[j],\ \ y[j+1],\ \ \mathrm{RANK}_{1,2}[j+2])$
  $(i+2 \in S_2,\ \ j+2 \in S_1)$

Comparison – example

$$\mathrm{cmp}(\texttt{sippi\$\$} \in S_0, \texttt{ssippi\$\$} \in S_2)$$

reduces to:

$$\mathrm{cmp}\left((\texttt{s},\ \texttt{i},\ \mathrm{RANK}_{1,2}[\texttt{ppi\$\$}]),\quad (\texttt{s},\ \texttt{s},\ \mathrm{RANK}_{1,2}[\texttt{ippi\$\$}])\right)$$

i	$\mathrm{SA}_{1,2}[i]$	suf. in y
0	10	i$$
1	7	ippi$$
2	4	issippi$$
3	1	ississippi$$
4	8	ppi$$
5	5	ssippi$$
6	2	ssissippi$$