Algorithmic Challenges: Suffix Array

Michael Levin

Higher School of Economics

Algorithms on Strings
Data Structures and Algorithms

Outline

1. Suffix Array
2. General Construction Strategy
3. Initialization
4. Sort Doubled Cyclic Shifts
5. Updating Classes and Full Algorithm

Construct Suffix Array

Input:	String S
Output:	All suffixes of S in lexicographic order

Alphabet

We assume the alphabet is ordered, that is, for any two different characters in the alphabet one of them is considered smaller than another. For example, in English

$$'a' < 'b' < 'c' < \cdots < 'z'$$

Definition

String S is lexicographically smaller than string T if $S \neq T$ and there exist such i that:

- $0 \leq i \leq |S|$
- $S[0..i-1] = T[0..i-1]$ (assume $S[0..-1]$ is an empty string)
- Either $i = |S|$ (then S is a prefix of T) or $S[i] < T[i]$

Examples

"ab" $<$ "bc" $(i = 0)$

"abc" $<$ "abd" $(i = 2)$

"abc" $<$ "abcd" $(i = 3)$

Suffix Array Example

$S = ababaa$

Suffixes in lexicographic order:

a

aa

$abaa$

$ababaa$

baa

$babaa$

Avoiding Prefix Rule

- Inconvenient rule: if S is a prefix of T, then $S < T$
- Append special character '\$' smaller than all other characters to the end of all strings
- If S is a prefix of T, then $S\$$ differs from $T\$$ in position $i = |S|$, and $\$ < T[|S|]$, so $S\$ < T\$$

Example

$S =$ "ababaa" $\Rightarrow S' =$ "ababaa\$"

Suffixes in lexicographic order:

\$

$a\$$

$aa\$$

$abaa\$$

$ababaa\$$

$baa\$$

$babaa\$$

Example

$S =$ "ababaa" $\Rightarrow S' =$ "ababaa\$"

Suffixes in lexicographic order:

a

aa

$abaa$

$ababaa$

baa

$babaa$

Storing Suffix Array

- Total length of all suffixes is $1 + 2 + \cdots + |S| = \Theta(|S|^2)$

Storing Suffix Array

- Total length of all suffixes is $1 + 2 + \cdots + |S| = \Theta(|S|^2)$
- Storing them all is too much memory

Storing Suffix Array

- Total length of all suffixes is $1 + 2 + \cdots + |S| = \Theta(|S|^2)$
- Storing them all is too much memory
- Store the order of suffixes $O(|S|)$

Storing Suffix Array

- Total length of all suffixes is $1 + 2 + \cdots + |S| = \Theta(|S|^2)$
- Storing them all is too much memory
- Store the order of suffixes $O(|S|)$
- Suffix array is this order

Example

$S = ababaa\$$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = []$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4, 2]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4, 2, 0]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4, 2, 0, 3]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4, 2, 0, 3, 1]$

Example

$S = ababaa\$$

Suffixes are numbered by their starting positions: $ababaa\$$ is 0, $abaa\$$ is 2

Suffix array: $order = [6, 5, 4, 2, 0, 3, 1]$

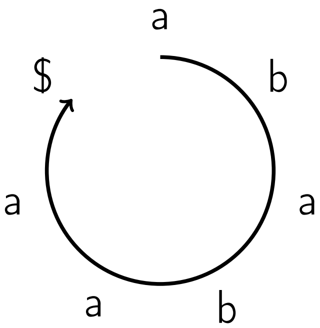
if we need to look at, for example, the third character of the second suffix in order we can first go into the area order, find out which suffix is number two, and that will be the first position of that suffix in the stream. And if we then add two to that we'll get character with position two in that suffix. So in theory we can look at any character of any suffix really efficiently although we don't store those suffixes directly.

- OK, you know how to store suffix array

- OK, you know how to store suffix array
- But how to construct it?

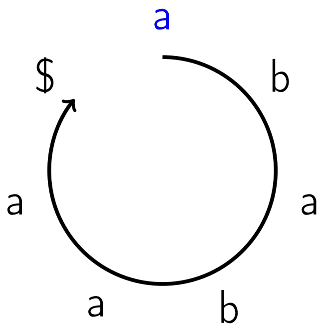
Outline

Sorting Cyclic Shifts

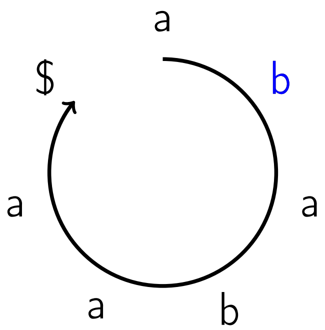


Sorting Cyclic Shifts

ababaa$



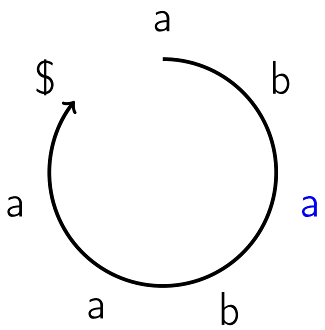
Sorting Cyclic Shifts



ababaa$

babaa$a

Sorting Cyclic Shifts

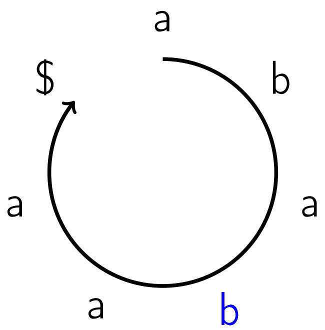


ababaa$

babaa$a

abaa$ab

Sorting Cyclic Shifts



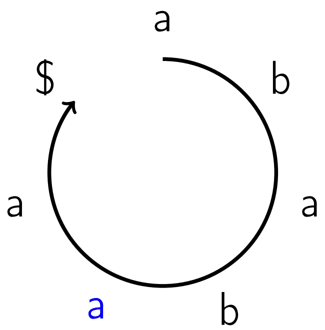
ababaa$

babaa$a

abaa$ab

baa$aba

Sorting Cyclic Shifts



ababaa\$

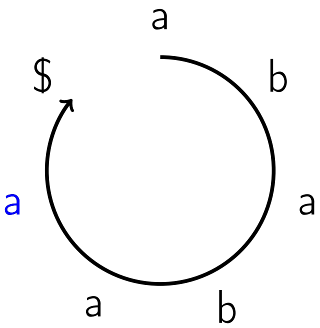
babaa\$a

abaa\$ab

baa\$aba

aa\$abab

Sorting Cyclic Shifts



ababaa$

babaa$a

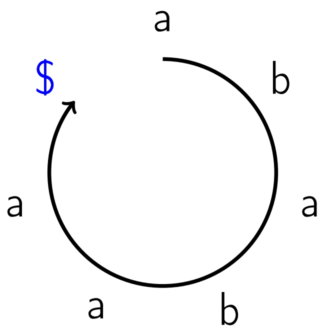
abaa$ab

baa$aba

aa$abab

a$ababa

Sorting Cyclic Shifts



ababaa$

babaa$a

abaa$ab

baa$aba

aa$abab

a$ababa

$ababaa

Sorting Cyclic Shifts

ababaa\$

babaa\$a

abaa\$ab

baa\$aba

aa\$abab

a\$ababa

\$ababaa

Sorting Cyclic Shifts

ababaa\$	\$ababaa
babaa\$a	a\$ababa
abaa\$ab	aa\$abab
baa\$aba	abaa\$ab
aa\$abab	ababaa\$
a\$ababa	baa\$aba
\$ababaa	babaa\$a

Sorting Cyclic Shifts

ababaa$	$ababaa
babaa$a	a$ababa
abaa$ab	aa$abab
baa$aba	abaa$ab
aa$abab	ababaa$
a$ababa	baa$aba
$ababaa	babaa$a

Sorting Cyclic Shifts

ababaa$	$ababaa	$
babaa$a	a$ababa	a$
abaa$ab	aa$abab	aa$
baa$aba	abaa$ab	abaa$
aa$abab	ababaa$	ababaa$
a$ababa	baa$aba	baa$
$ababaa	babaa$a	babaa$

Lemma

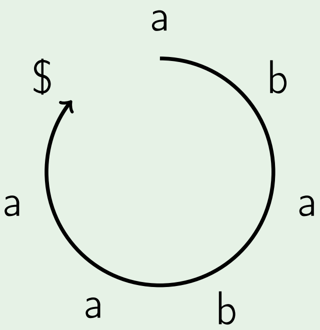
After adding to the end of string S character \$ which is smaller than all other characters, sorting cyclic shifts of S and suffixes of S is equivalent.

Partial Cyclic Shifts

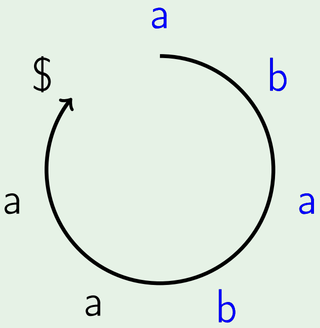
Definition

Substrings of cyclic string S are called partial cyclic shifts of S

Partial Cyclic Shifts Example



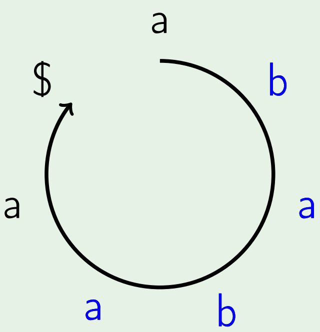
Partial Cyclic Shifts Example



Cyclic shifts of length 4:

abab

Partial Cyclic Shifts Example

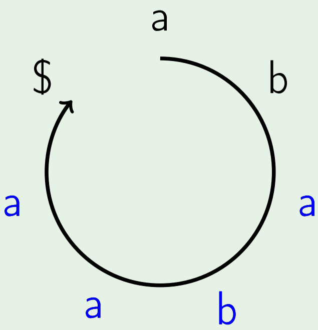


Cyclic shifts of length 4:

abab

baba

Partial Cyclic Shifts Example



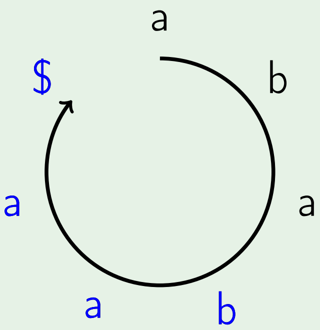
Cyclic shifts of length 4:

abab

baba

abaa

Partial Cyclic Shifts Example



Cyclic shifts of length 4:

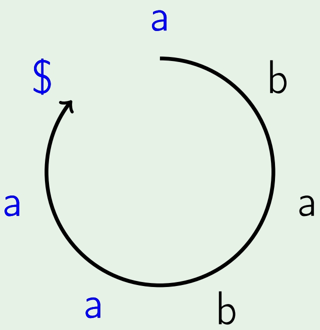
abab

baba

abaa

baa$

Partial Cyclic Shifts Example



Cyclic shifts of length 4:

abab

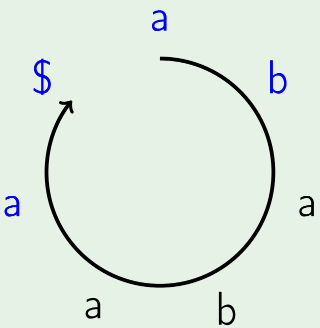
baba

abaa

baa$

aa$a

Partial Cyclic Shifts Example



Cyclic shifts of length 4:

abab

baba

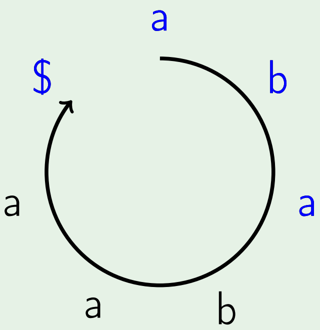
abaa

baa$

aa$a

a$ab

Partial Cyclic Shifts Example



Cyclic shifts of length 4:

abab

baba

abaa

baa$

aa$a

a$ab

$aba

General strategy

- Start with sorting single characters of S

General strategy

- Start with sorting single characters of S
- Cyclic shifts of length $L = 1$ sorted

General strategy

- Start with sorting single characters of S
- Cyclic shifts of length $L = 1$ sorted
- While $L < |S|$, sort shifts of length $2L$

General strategy

- Start with sorting single characters of S
- Cyclic shifts of length $L = 1$ sorted
- While $L < |S|$, sort shifts of length $2L$
- If $L \geq |S|$, cyclic shifts of length L sort the same way as cyclic shifts of length $|S|$

Example

$S = ababaa\$$

Example

$S = ababaa\$$

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

Example

$S = ababaa\$$

$order = [6, 0, 2, 4, 5, 1, 3]$

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

Example

$S = ababaa\$$

6	$\$a$
5	$a\$$
4	aa
0	ab
2	ab
1	ba
3	ba

Example

$S = ababaa\$$

$6 \quad \$a$ $\qquad order = [6, 5, 4, 0, 2, 1, 3]$

$5 \quad a\$$

$4 \quad aa$

$0 \quad ab$

$2 \quad ab$

$1 \quad ba$

$3 \quad ba$

Example

$S = ababaa\$$

6 $\$aba$

5 $a\$ab$

4 $aa\$a$

2 $abaa$

0 $abab$

3 $baa\$$

1 $baba$

Example

$S = ababaa\$$

$6\ \$aba$ $\qquad$ $order = [6, 5, 4, 2, 0, 3, 1]$

$5\ a\$ab$

$4\ aa\$a$

$2\ abaa$

$0\ abab$

$3\ baa\$$

$1\ baba$

Example

$S = ababaa\$$

6 $\$ababaa\$$

5 $a\$ababaa$

4 $aa\$ababa$

2 $abaa\$aba$

0 $ababaa\$a$

3 $baa\$abab$

1 $babaa\$ab$

Example

$S = ababaa\$$

6 $\$ababaa\$$ $order = [6, 5, 4, 2, 0, 3, 1]$

5 $a\$ababaa$

4 $aa\$ababa$

2 $abaa\$aba$

0 $ababaa\$a$

3 $baa\$abab$

1 $babaa\$ab$

Example

$S = ababaa\$$

6 $\$ababaa\$$ $\quad order = [6, 5, 4, 2, 0, 3, 1]$

5 $a\$ababaa$

4 $aa\$ababa$

2 $abaa\$aba$

0 $ababaa\$a$

3 $baa\$abab$

1 $babaa\$ab$

Example

$S = ababaa\$$

$6 \ \$$ $\qquad order = [6, 5, 4, 2, 0, 3, 1]$

$5 \ a\$$

$4 \ aa\$$

$2 \ abaa\$$

$0 \ ababaa\$$

$3 \ baa\$$

$1 \ babaa\$$

Outline

Sorting single characters

- Alphabet Σ has $|\Sigma|$ different characters

Sorting single characters

- Alphabet Σ has $|\Sigma|$ different characters
- Use counting sort to compute *order* of characters

SortCharacters(S)

```
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
   count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   c ← S[i]
   count[c] ← count[c] − 1
   order[count[c]] ← i
return order
```

SortCharacters(S)

```
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
   count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   c ← S[i]
   count[c] ← count[c] − 1
   order[count[c]] ← i
return order
```

SortCharacters(S)

```
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
    count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    c ← S[i]
    count[c] ← count[c] − 1
    order[count[c]] ← i
return order
```

```
SortCharacters(S)
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
   count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   c ← S[i]
   count[c] ← count[c] − 1
   order[count[c]] ← i
return order
```

SortCharacters(S)

```
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
   count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   c ← S[i]
   count[c] ← count[c] − 1
   order[count[c]] ← i
return order
```

```
SortCharacters(S)
order ← array of size |S|
count ← zero array of size |Σ|
for i from 0 to |S| − 1:
   count[S[i]] ← count[S[i]] + 1
for j from 1 to |Σ| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   c ← S[i]
   count[c] ← count[c] − 1
   order[count[c]] ← i
return order
```

Lemma

Running time of `SortCharacters` is $O(|S| + |\Sigma|)$.

Proof

We know this is the running time of the counting sort for $|S|$ items that can take $|\Sigma|$ different values. $\square$

Equivalence classes

- C_i — partial cyclic shift of length L starting in i
- C_i can be equal to C_j — then they are in one equivalence class
- Compute $class[i]$ — number of *different* cyclic shifts of length L that are strictly smaller than C_i
- $C_i == C_j \Leftrightarrow class[i] == class[j]$

Example

$S = ababaa\$$

6	\$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [\ ,\ ,\ ,\ ,\ ,\ ,\ ]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [\ ,\ ,\ ,\ ,\ ,\ ,\ ]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [\ ,\ ,\ ,\ ,\ ,\ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

$$order = [6, 0, 2, 4, 5, 1, 3]$$

$$class = [\ ,\ ,\ ,\ ,\ ,\ , 0]$$

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

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \ , \ , \ , \ , \ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

$order = [6, 0, 2, 4, 5, 1, 3]$

$class = [1, \ , \ , \ , \ , \ , 0]$

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

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \ , 1, \ , \ , \ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \ , 1, \ , \ , \ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \ \ , 1, \ \ , 1, \ \ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \ , 1, \ , 1, \ , 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, \quad, 1, \quad, 1, 1, 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

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

$order = [6, 0, 2, 4, 5, 1, 3]$

$class = [1, \quad, 1, \quad, 1, 1, 0]$

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, 2, 1, \quad , 1, 1, 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

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

$order = [6, 0, 2, 4, 5, 1, 3]$

$class = [1, 2, 1, \quad, 1, 1, 0]$

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, 2, 1, 2, 1, 1, 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

Example

$S = ababaa\$$

6	$\$$	$order = [6, 0, 2, 4, 5, 1, 3]$
0	a	$class = [1, 2, 1, 2, 1, 1, 0]$
2	a	
4	a	
5	a	
1	b	
3	b	

ComputeCharClasses(S, $order$)

Computes equivalence classes just for single character cyclic shifts of string s given their order

```
class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
    if S[order[i]] ≠ S[order[i − 1]]:
        class[order[i]] = class[order[i − 1]] + 1
    else:
        class[order[i]] = class[order[i − 1]]
return class
```

ComputeCharClasses(S, $order$)

$class \leftarrow$ array of size $|S|$

$class[order[0]] \leftarrow 0$

for i from 1 to $|S| - 1$:

  if $S[order[i]] \neq S[order[i-1]]$:

    $class[order[i]] = class[order[i-1]] + 1$

  else:

    $class[order[i]] = class[order[i-1]]$

return $class$

ComputeCharClasses(S, $order$)

```
class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
   if S[order[i]] ≠ S[order[i − 1]]:
      class[order[i]] = class[order[i − 1]] + 1
   else:
      class[order[i]] = class[order[i − 1]]
return class
```

```
ComputeCharClasses(S, order)
class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
    if S[order[i]] ≠ S[order[i − 1]]:
        class[order[i]] = class[order[i − 1]] + 1
    else:
        class[order[i]] = class[order[i − 1]]
return class
```

```
ComputeCharClasses(S, order)

class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
    if S[order[i]] ≠ S[order[i − 1]]:
        class[order[i]] = class[order[i − 1]] + 1
    else:
        class[order[i]] = class[order[i − 1]]
return class
```

```
ComputeCharClasses(S, order)
class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
    if S[order[i]] ≠ S[order[i − 1]]:
        class[order[i]] = class[order[i − 1]] + 1
    else:
        class[order[i]] = class[order[i − 1]]
return class
```

```
ComputeCharClasses(S, order)
class ← array of size |S|
class[order[0]] ← 0
for i from 1 to |S| − 1:
    if S[order[i]] ≠ S[order[i − 1]]:
        class[order[i]] = class[order[i − 1]] + 1
    else:
        class[order[i]] = class[order[i − 1]]
return class
```

Lemma

The running time of `ComputeCharClasses` is $O(|S|)$.

Proof

One for loop with $O(|S|)$ iterations. $\square$

Outline

Idea

- C_i — cyclic shift of length L starting in i

Idea

- C_i — cyclic shift of length L starting in i
- C'_i — doubled cyclic shift starting in i

Idea

- C_i — cyclic shift of length L starting in i
- C'_i — doubled cyclic shift starting in i
- $C'_i = C_i C_{i+L}$ — concatenation of strings

Idea

- C_i — cyclic shift of length L starting in i
- C'_i — doubled cyclic shift starting in i
- $C'_i = C_i C_{i+L}$ — concatenation of strings
- To compare C'_i with C'_j, it's sufficient to compare C_i with C_j and C_{i+L} with C_{j+L}

See Next Slide

Example

$S = ababaa\$$

$L = 2$

$i = 2$

$C_i = C_2 = ab$

$C_{i+L} = C_{2+2} = C_4 = aa$

$C'_i = C'_2 = abaa = C_2 C_4$

Sorting pairs

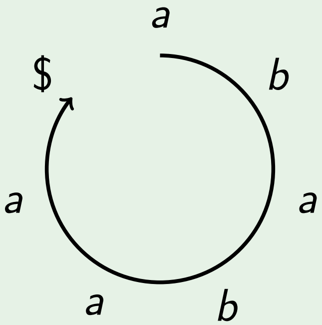
- First sort by second element of pair

Sorting pairs

- First sort by second element of pair
- Then **stable** sort by first element of pair

Example

$L = 2$



$C_6 = \$a$

$C_5 = a\$$

$C_4 = aa$

$C_0 = ab$

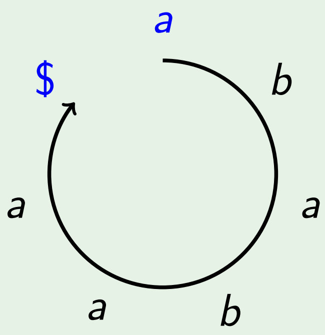
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$C_6 = \$a$

$C_5 = a\$$

$C_4 = aa$

$C_0 = ab$

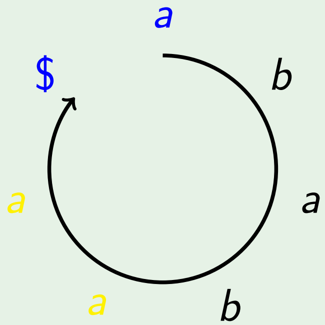
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$$C_4' = aa\$a$$

$$C_5 = a\$$$

$$C_4 = aa$$

$$C_0 = ab$$

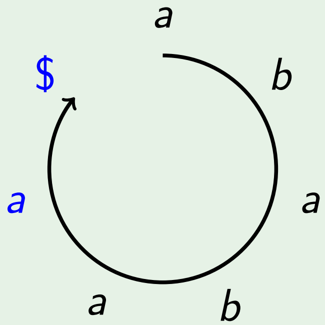
$$C_2 = ab$$

$$C_1 = ba$$

$$C_3 = ba$$

Example

$L = 2$



$C'_4 = aa\$a$

$C_5 = a\$$

$C_4 = aa$

$C_0 = ab$

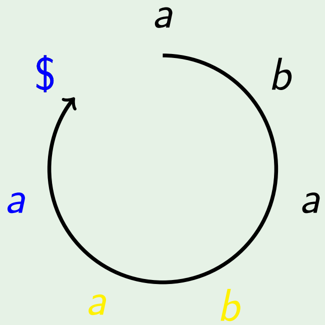
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C_4 = aa$

$C_0 = ab$

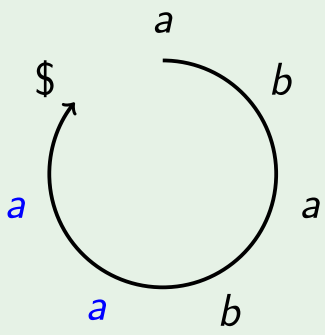
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$$L = 2$$



$$C'_4 = aa\$a$$

$$C'_3 = baa\$$$

$$C_4 = aa$$

$$C_0 = ab$$

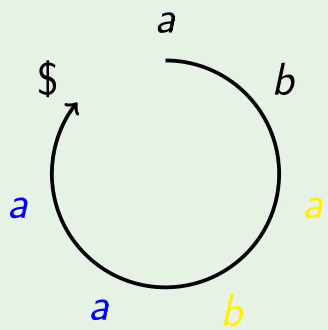
$$C_2 = ab$$

$$C_1 = ba$$

$$C_3 = ba$$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C_0 = ab$

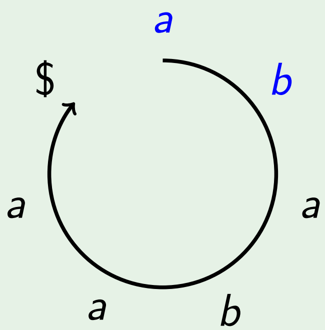
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$$L = 2$$



$$C'_4 = aa\$a$$

$$C'_3 = baa\$$$

$$C'_2 = abaa$$

$$C_0 = ab$$

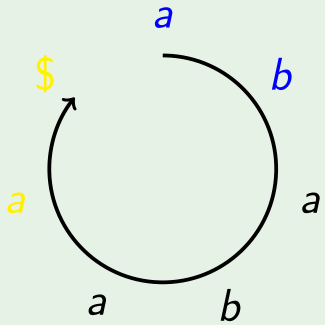
$$C_2 = ab$$

$$C_1 = ba$$

$$C_3 = ba$$

Example

$$L = 2$$



$$C'_4 = aa\$a$$

$$C'_3 = baa\$$$

$$C'_2 = abaa$$

$$C'_5 = a\$ab$$

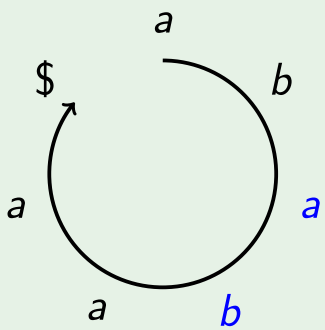
$$C_2 = ab$$

$$C_1 = ba$$

$$C_3 = ba$$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

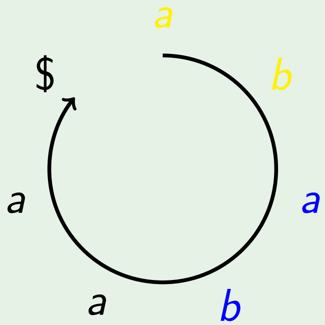
$C_2 = ab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

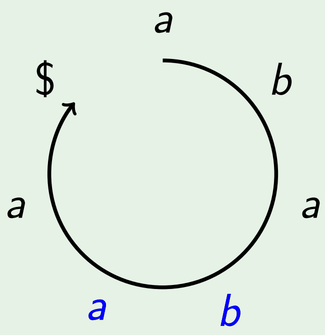
$C'_0 = abab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

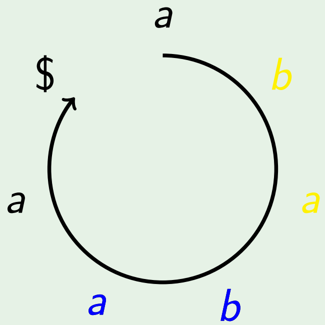
$C'_0 = abab$

$C_1 = ba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

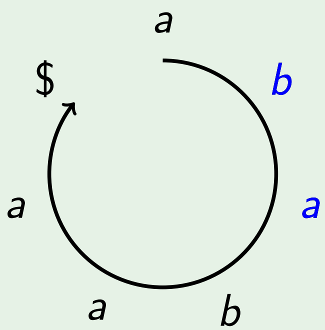
$C'_0 = abab$

$C'_6 = \$aba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

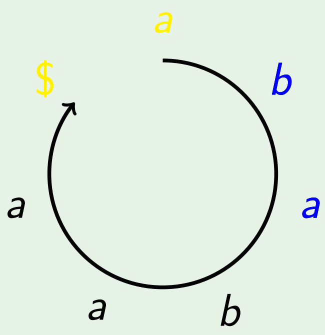
$C'_0 = abab$

$C'_6 = \$aba$

$C_3 = ba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

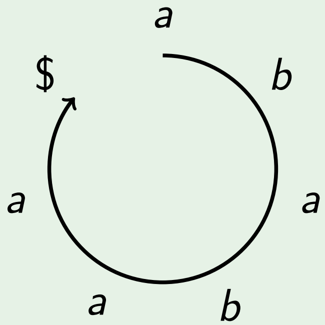
$C'_0 = abab$

$C'_6 = \$aba$

$C'_1 = baba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

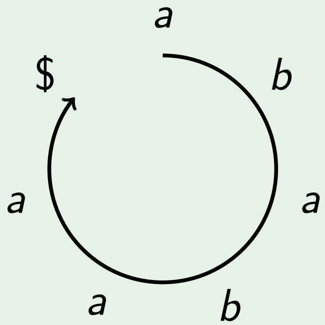
$C'_0 = abab$

$C'_6 = \$aba$

$C'_1 = baba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

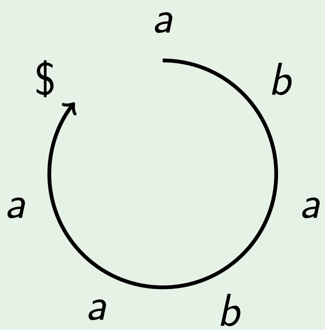
$C'_0 = abab$

$C'_6 = \$aba$

$C'_1 = baba$

Example

$L = 2$



$C'_4 = aa\$a$

$C'_3 = baa\$$

$C'_2 = abaa$

$C'_5 = a\$ab$

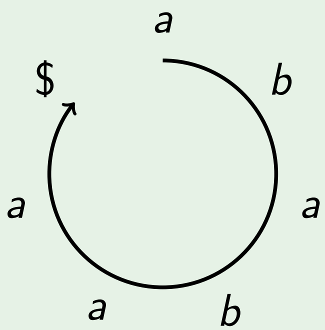
$C'_0 = abab$

$C'_6 = \$aba$

$C'_1 = baba$

Example

$L = 2$



$C'_6 = \$aba$

$C'_5 = a\$ab$

$C'_4 = aa\$a$

$C'_2 = abaa$

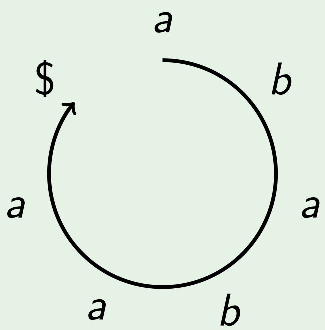
$C'_0 = abab$

$C'_3 = baa\$$

$C'_1 = baba$

Example

$L = 2$



$C'_6 = \$aba$

$C'_5 = a\$ab$

$C'_4 = aa\$a$

$C'_2 = abaa$

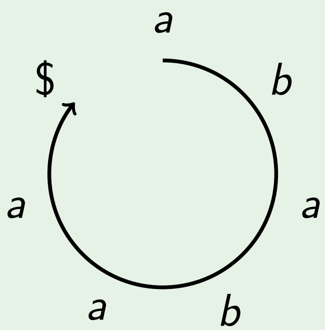
$C'_0 = abab$

$C'_3 = baa\$$

$C'_1 = baba$

Example

$L = 2$



$C'_6 = \$aba$

$C'_5 = a\$ab$

$C'_4 = aa\$a$

$C'_2 = abaa$

$C'_0 = abab$

$C'_3 = baa\$$

$C'_1 = baba$

Sorting doubled cyclic shifts

- C_i' — doubled cyclic shift starting in i

Sorting doubled cyclic shifts

- C'_i — doubled cyclic shift starting in i
- C'_i is a pair (C_i, C_{i+L})

Sorting doubled cyclic shifts

- C'_i — doubled cyclic shift starting in i
- C'_i is a pair (C_i, C_{i+L})
- $C_{order[0]}, C_{order[1]}, \ldots, C_{order[|S|-1]}$ are already sorted

Sorting doubled cyclic shifts

- C'_i — doubled cyclic shift starting in i
- C'_i is a pair (C_i, C_{i+L})
- $C_{order[0]}, C_{order[1]}, \ldots, C_{order[|S|-1]}$ are already sorted
- Take doubled cyclic shifts starting exactly L counter-clockwise ("to the left")

Sorting doubled cyclic shifts

- C'_i — doubled cyclic shift starting in i
- C'_i is a pair (C_i, C_{i+L})
- $C_{order[0]}, C_{order[1]}, \dots, C_{order[|S|-1]}$ are already sorted
- Take doubled cyclic shifts starting exactly L counter-clockwise ("to the left")
- $C'_{order[0]-L}, C'_{order[1]-L}, \dots, C_{order[|S|-1]-L}$ are sorted by second element of pair

Sorting doubled cyclic shifts

- $C'_{order[0]-L}, C'_{order[1]-L}, \cdots, C_{order[|S|-1]-L}$ are sorted by second element of pair
- Need a stable sort by first elements of pairs
- Counting sort is stable!
- We know equivalence classes of single shifts for counting sort

SortDoubled($S, L, order, class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

SortDoubled($S, L, order, class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

SortDoubled(S, L, $order$, $class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

SortDoubled(S, L, $order$, $class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
   count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   start ← (order[i] − L + |S|) mod |S|
   cl ← class[start]
   count[cl] ← count[cl] − 1
   newOrder[count[cl]] ← start
return newOrder
```

SortDoubled($S, L, order, class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

This is very important as we are sorting from the end to the beginning in a suffix

SortDoubled($S, L, order, class$)

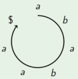
$count \leftarrow$ zero array of size $|S|$
$newOrder \leftarrow$ array of size $|S|$
for i from 0 to $|S| - 1$:
  $count[class[i]] \leftarrow count[class[i]] + 1$
for j from 1 to $|S| - 1$:
  $count[j] \leftarrow count[j] + count[j - 1]$
for i from $|S| - 1$ down to 0:
  $start \leftarrow (order[i] - L + |S|) \mod |$
  $cl \leftarrow class[start]$
  $count[cl] \leftarrow count[cl] - 1$
  $newOrder[count[cl]] \leftarrow start$
return $newOrder$

What will be the order of values assigned to the variable *start* in this example?

for i from $|S| - 1$ down to 0:
  $start \leftarrow (order[i] - L + |S|) \mod |S|$

Example

$L = 2$



$C'_4 = aa\$a$
$C'_3 = ba a\$$
$C'_2 = ab aa$
$C'_5 = a\$ ab$
$C'_0 = ab ab$
$C'_1 = ba ba$
$C'_6 = \$a ba$

- [6,5,4,3,2,1,0]
- [6,1,0,5,2,3,4]
- [1,3,0,2,4,5,6]
- [4,3,2,5,0,1,6]

✓ Correct
Correct! It will go through the starts of the doubled cyclic shifts C'_4, C'_3 and so on in the **reversed** order of their sorted second halves, and those are already sorted in the picture.

SortDoubled(S, L, $order$, $class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
   count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
   count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
   start ← (order[i] − L + |S|) mod |S|
   cl ← class[start]
   count[cl] ← count[cl] − 1
   newOrder[count[cl]] ← start
return newOrder
```

SortDoubled(S, L, $order$, $class$)

$count \leftarrow$ zero array of size $|S|$
$newOrder \leftarrow$ array of size $|S|$
for i from 0 to $|S| - 1$:
  $count[class[i]] \leftarrow count[class[i]] + 1$
for j from 1 to $|S| - 1$:
  $count[j] \leftarrow count[j] + count[j-1]$
for i from $|S| - 1$ down to 0:
  $start \leftarrow (order[i] - L + |S|) \mod |S|$
  $cl \leftarrow class[start]$
  $count[cl] \leftarrow count[cl] - 1$
  $newOrder[count[cl]] \leftarrow start$
return $newOrder$

SortDoubled($S, L, order, class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

SortDoubled($S, L, order, class$)

```
count ← zero array of size |S|
newOrder ← array of size |S|
for i from 0 to |S| − 1:
    count[class[i]] ← count[class[i]] + 1
for j from 1 to |S| − 1:
    count[j] ← count[j] + count[j − 1]
for i from |S| − 1 down to 0:
    start ← (order[i] − L + |S|) mod |S|
    cl ← class[start]
    count[cl] ← count[cl] − 1
    newOrder[count[cl]] ← start
return newOrder
```

Lemma

The running time of `SortDoubled` is $O(|S|)$.

Proof

Three for loops with $O(|S|)$ iterations each. $\square$

And the running time of this procedure is linear because this is basically the regular counting sort. Although it sorts very complex objects,in practice in the code, it just sorts integers,the equivalent classes of the single cyclic shifts and it does so in the running time of the counting sort which runs in the time number of items plus number of different values. Number of items is equal to length of the string and the number of different values of a classes is also to smallest length of the stream. So all in all, those three for loops run in linear time.

Outline

Updating classes

- Pairs are sorted — go through them in order, if a pair is different from previous, put it into a new class, otherwise put it into previous class
-
$$(P_1, P_2) == (Q_1, Q_2) \Leftrightarrow$$
$$(P_1 == Q_1) \text{ and } (P_2 == Q_2)$$
- We know equivalence classes of elements of pairs

Question

When we proceed from cyclic shifts of length L to cyclic shifts of length $2L$, what can happen with the number of equivalence classes?

- ○ It can only increase.
- ○ It can increase, decrease or stay the same.
- ○ It can only decrease.
- ○ It can either increase or decrease, but cannot stay the same.
- ◉ It can either increase or stay the same.
- ○ It can either decrease or stay the same.

✓ **Correct**

Correct! Doubled cyclic shifts can be equal only if their first halfs (which are single cyclic shifts) are equal, so the number of classes cannot decrease. However, it can stay the same. For example, if all the letters in the string are different, then the number of classes is equal to length of the string for any L.

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [\ ,\ ,\ ,\ ,\ ,\ ,\ ]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0, 1)$ $\leftarrow$ $class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [\ ,\ ,\ ,\ ,\ ,\ ,\ ]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0, 1) \leftarrow class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [\ ,\ ,\ ,\ ,\ ,\ , 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1)$ $\leftarrow$ $class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [\ ,\ ,\ ,\ ,\ ,\ ,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$$S = ababaa\$$$

C'_6 $\$a$ $(0, 1) \leftarrow class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [\ ,\ ,\ ,\ ,\ , 1, 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [\ ,\ ,\ ,\ ,\ ,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0, 1) \leftarrow class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [\ ,\ ,\ ,\ , 2, 1, 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [\ ,\ ,\ ,\ ,2,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$$S = ababaa\$$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [3, \ , \ , \ ,2,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [3, \ , \ , \ , 2, 1, 0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [3, \quad ,3, \quad ,2,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0, 1)$ $\leftarrow$ $class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [3, \quad, 3, \quad, 2, 1, 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$$S = ababaa\$$$

C'_6 $\$a$ $(0, 1) \leftarrow class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [3, 4, 3, \quad, 2, 1, 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [3,4,3,\ ,2,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0,1) \leftarrow class = [1,2,1,2,1,1,0]$

C'_5 $a\$$ $(1,0)$ $newOrder = [6,5,4,0,2,1,3]$

C'_4 aa $(1,1)$ $newClass = [3,4,3,4,2,1,0]$

C'_0 ab $(1,2)$

C'_2 ab $(1,2)$

C'_1 ba $(2,1)$

C'_3 ba $(2,1)$

Example

$S = ababaa\$$

C'_6 $\$a$ $(0, 1) \leftarrow class = [1, 2, 1, 2, 1, 1, 0]$

C'_5 $a\$$ $(1, 0)$ $newOrder = [6, 5, 4, 0, 2, 1, 3]$

C'_4 aa $(1, 1)$ $newClass = [3, 4, 3, 4, 2, 1, 0]$

C'_0 ab $(1, 2)$

C'_2 ab $(1, 2)$

C'_1 ba $(2, 1)$

C'_3 ba $(2, 1)$

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if  class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if  class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if  class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

`UpdateClasses`(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

UpdateClasses(*newOrder*, *class*, *L*)

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if  class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

`UpdateClasses(`*newOrder*, *class*, *L*`)`

```
n ← |newOrder|
newClass ← array of size n
newClass[newOrder[0]] ← 0
for i from 1 to n − 1:
   cur ← newOrder[i], prev ← newOrder[i − 1]
   mid ← (cur + L), midPrev ← (prev + L) (mod n)
   if class[cur] ≠ class[prev] or
         class[mid] ≠ class[midPrev]:
      newClass[cur] ← newClass[prev] + 1
   else:
      newClass[cur] ← newClass[prev]
return newClass
```

Lemma

The running time of `UpdateClasses` is $O(|S|)$.

Proof

One for loop with $O(|S|)$ iterations. $\square$

Just one for loop and constant time iterations happening inside

```
BuildSuffixArray(S)

order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

So procedure BuildSuffixArraytakes in only string S and returns the order of the cyclic shifts orof the suffixes of this string.

```
BuildSuffixArray(S)

order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

```
BuildSuffixArray(S)

order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

BuildSuffixArray(S)

```
order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

BuildSuffixArray(S)

```
order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

BuildSuffixArray(S)

```
order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

```
BuildSuffixArray(S)

order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

BuildSuffixArray(S)

```
order ← SortCharacters(S)
class ← ComputeCharClasses(S, order)
L ← 1
while L < |S|:
    order ← SortDoubled(S, L, order, class)
    class ← UpdateClasses(order, class, L)
    L ← 2L
return order
```

BuildSuffixArray(S)

$order \leftarrow$ SortCharacters(S)
$class \leftarrow$ ComputeCharClasses(S, $order$)
$L \leftarrow 1$
while $L < |S|$:
    $order \leftarrow$ SortDoubled(S, L, $order$, $class$)
    $class \leftarrow$ UpdateClasses($order$, $class$, L)
    $L \leftarrow 2L$

And by the time array order will contain the correct order of all the full cyclic shifts of the string S,which is the same as the correct order of all the suffixes of the string S if it has a $ on the

return $order$

Lemma

The running time of `BuildSuffixArray` is $O(|S|\log|S| + |\Sigma|)$.

Proof

- Initialization: `SortCharacters` in $O(|S| + |\Sigma|)$ and `ComputeCharClasses` in $O(|S|)$

Lemma

The running time of `BuildSuffixArray` is $O(|S|\log|S| + |\Sigma|)$.

Proof

- Initialization: `SortCharacters` in $O(|S| + |\Sigma|)$ and `ComputeCharClasses` in $O(|S|)$
- While loop iteration: `SortDoubled` and `UpdateClasses` run in $O(|S|)$

Lemma

The running time of `BuildSuffixArray` is $O(|S| \log |S| + |\Sigma|)$.

Proof

hence actual running time is initialization plus while loop hence O(|S| + |alphabet|) + O(|S|) + O(|S|log|S|) which is generalized as O(|S|log|S|)

- Initialization: `SortCharacters` in $O(|S| + |\Sigma|)$ and `ComputeCharClasses` in $O(|S|)$
- While loop iteration: `SortDoubled` and `UpdateClasses` run in $O(|S|)$
- $O(\log |S|)$ iterations while $L < |S|$ $\square$

Conclusion

TIme remains same as before but memory drastically reduced from O(| O(|n|^2)|^2) to O(|c|)

- Can build suffix array of a string S in $O(|S| \log |S|)$ using $O(|S|)$ memory
- Can also sort all cyclic shifts of a string S in $O(|S| \log |S|)$
- Suffix array enables many fast operations with the string
- Next lesson you will learn to construct suffix tree from suffix array in $O(|S|)$ time, so you will be able to build suffix tree in total $O(|S| \log |S|)$ time!