Algorithmic Challenges: From Suffix Array to Suffix Tree

Michael Levin

Higher School of Economics

Algorithms on Strings
Algorithms and Data Structures at edX

Outline

Construct suffix Tree

Input: String S

Output: Suffix tree of S

- You already know how to construct suffix tree
- But $O(|S|^2)$ will only work for short strings
- You will learn to build it in $O(|S| \log |S|)$ which enables very long texts!

General Plan

- Construct suffix array in $O(|S| \log |S|)$
- Compute additional information in $O(|S|)$
- Construct suffix tree from suffix array and additional information in $O(|S|)$

Suffix array and suffix tree

Question

What is the upper bound for the sum of lengths of all the edge labels of the suffix tree of a string S in the worst case?

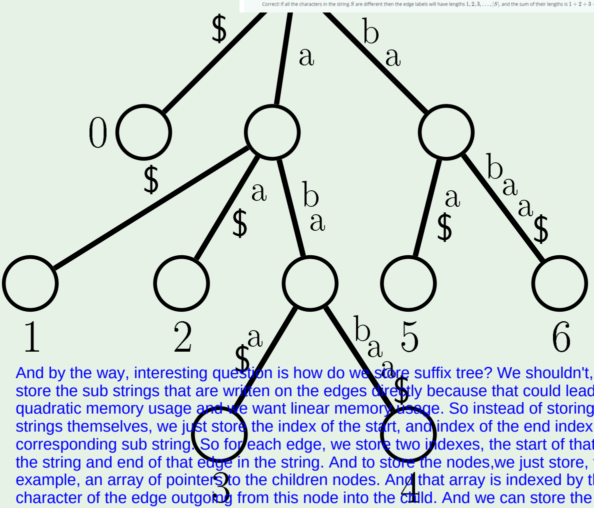
- (selected) $O(|S|^2)$
- $O(|S|)$
- $O(|S| \log |S|)$

Correct

Correct! If all the characters in the string S are different then the edge labels will have lengths $1, 2, 3, \ldots, |S|$, and the sum of their lengths is $1 + 2 + 3 + \cdots + |S| = \frac{|S|(|S|+1)}{2} = O(|S|^2)$.

$S = ababaa\$$

0	\$
1	a\$
2	aa\$
3	abaa\$
4	ababaa\$
5	baa\$
6	babaa\$



And by the way, interesting question is how do we store suffix tree? We shouldn't, of course, store the sub strings that are written on the edges directly because that could lead to quadratic memory usage and we want linear memory usage. So instead of storing the sub strings themselves, we just store the index of the start, and index of the end index of the corresponding sub string. So for each edge, we store two indexes, the start of that edge in the string and end of that edge in the string. And to store the nodes,we just store, for example, an array of pointers to the children nodes. And that array is indexed by the first character of the edge outgoing from this node into the child. And we can store the information about the edge itself in the node for which this edge is going from its parent. This is one of the ways to store everything but you may organize everything in another way. The important thing is that you shouldn't store edges as sub strings.

Suffix array and suffix tree

S = ababaa$

0 $

1 a$

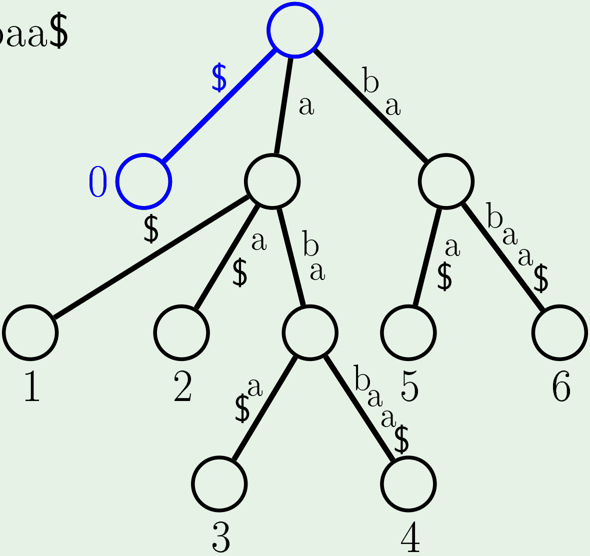
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array and suffix tree

S = ababaa$

0 $

1 a$

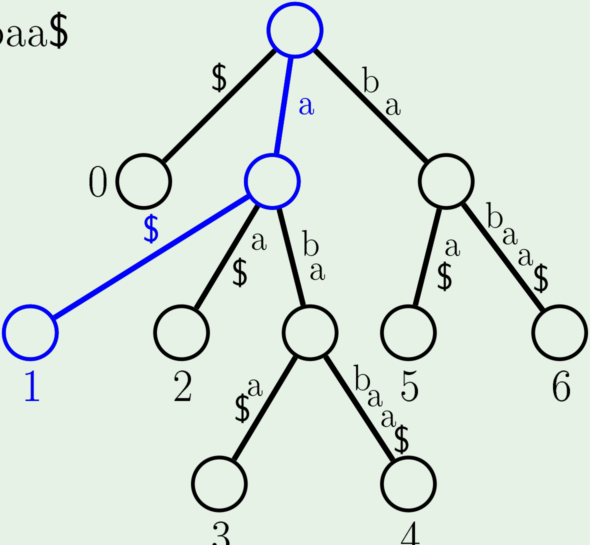
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array and suffix tree

S = ababaa$

0 $

1 a$

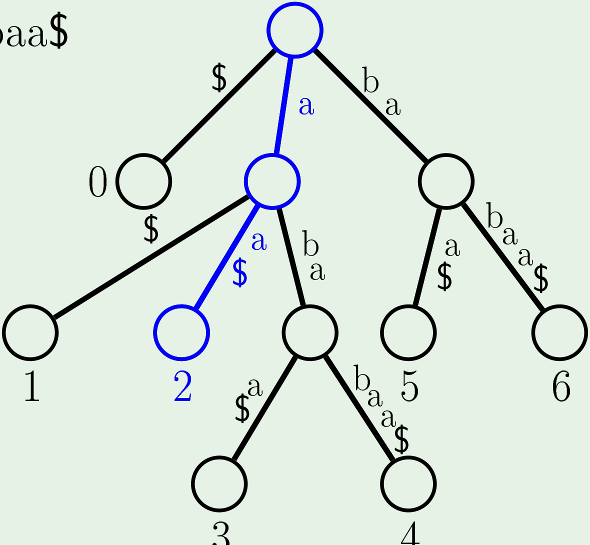
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array and suffix tree

$S = \text{ababaa}\$$

0 \$

1 a\$

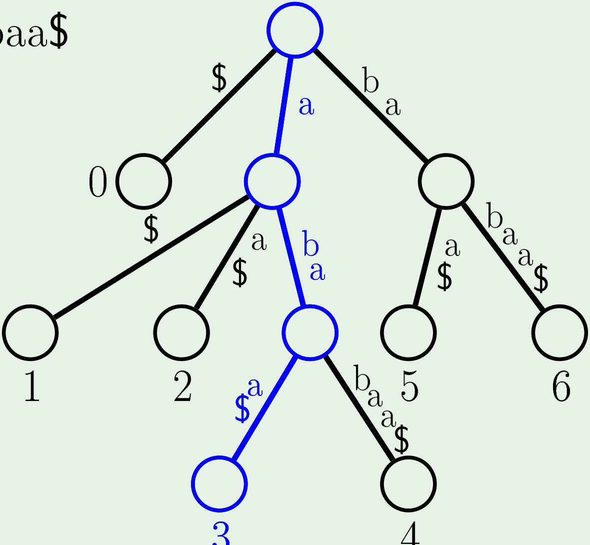
2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$



Suffix array and suffix tree

S = ababaa$

0 $

1 a$

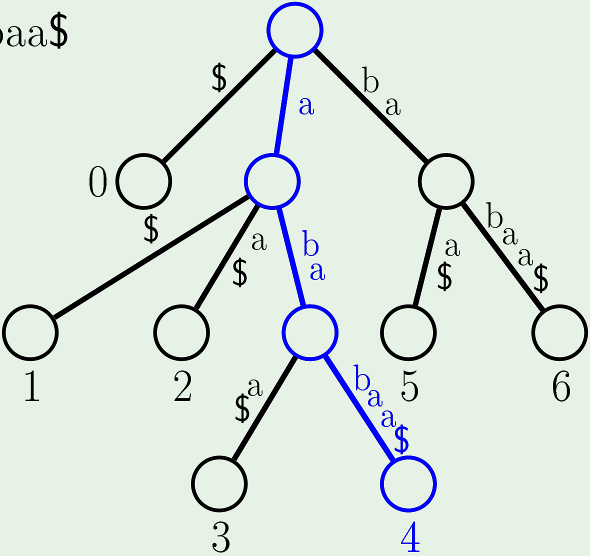
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array and suffix tree

S = ababaa$

0 $

1 a$

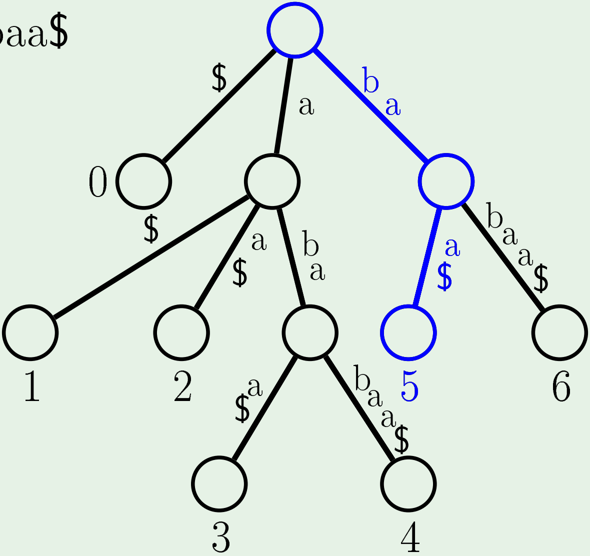
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array and suffix tree

S = ababaa$

0 $

1 a$

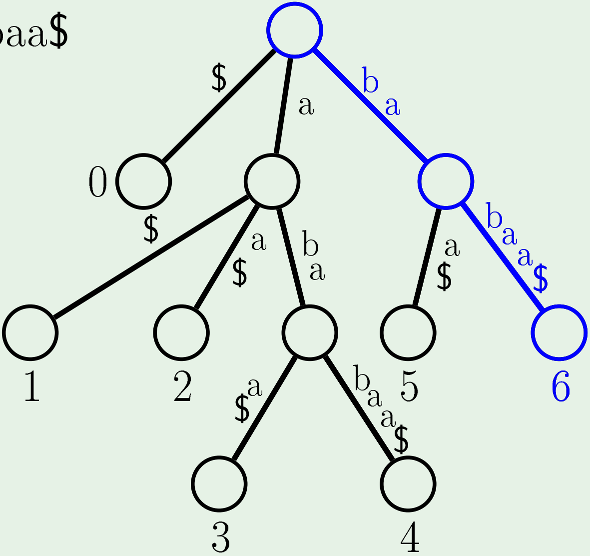
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Definition

The longest common prefix (or just "lcp") of two strings S and T is the longest such string u that u is both a prefix of S and T. We denote by LCP(S, T) the length of the "lcp" of S and T.

Example

$\mathrm{LCP}(\text{“ababc”}, \text{“abc”}) = 2$
$\mathrm{LCP}(\text{“a”}, \text{“b”}) = 0$

Suffix array, suffix tree and lcp

$S = \text{ababaa\$}$

0 \$

1 a\$

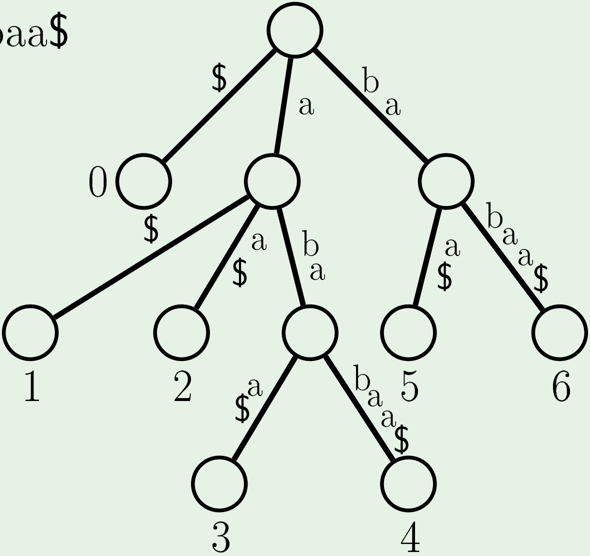
2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 a$

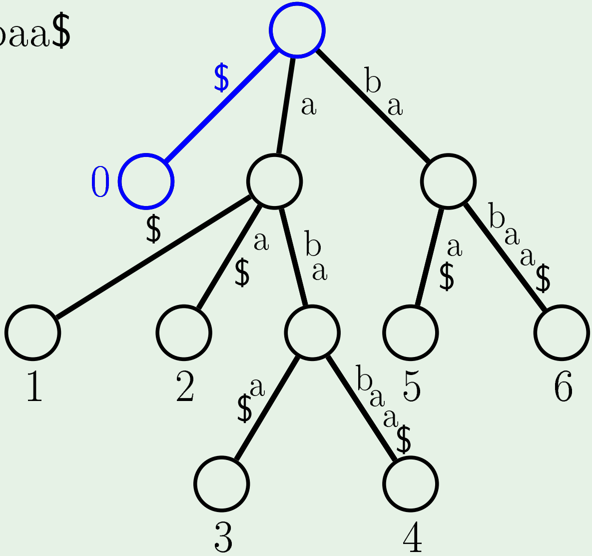
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

S = ababaa$

0 $

1 a$

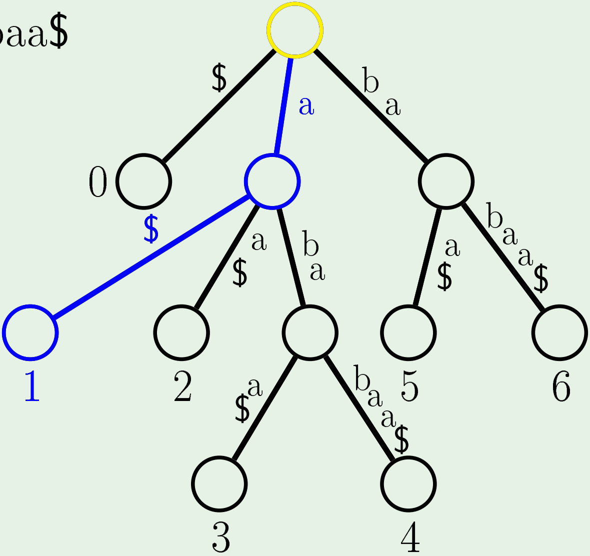
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

S = ababaa\$

0 \$

1 a\$

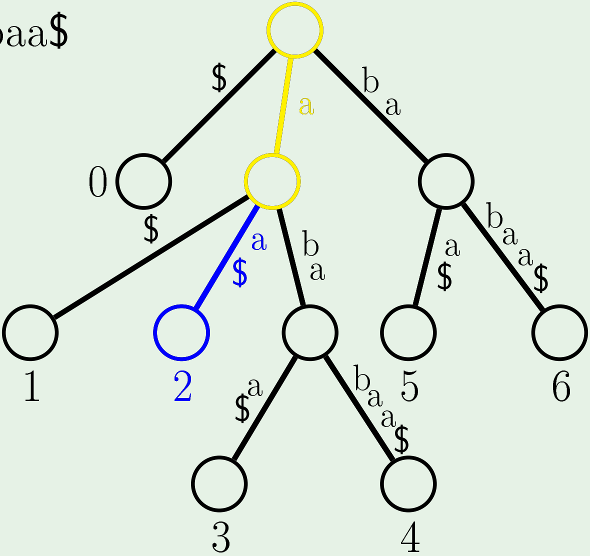
2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$



Suffix array, suffix tree and lcp

S = ababaa$

0 $

1 a$

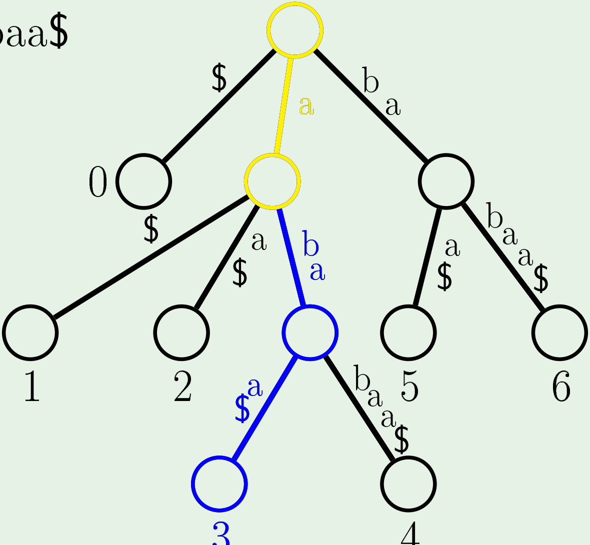
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

S = ababaa$

0 $

1 a$

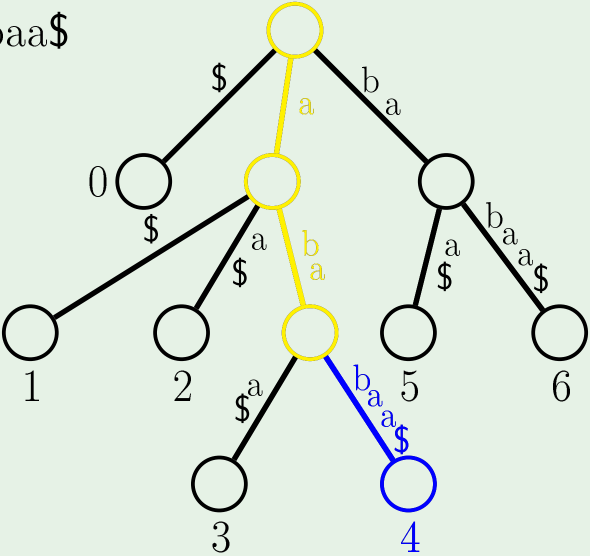
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

S = ababaa$

0 $

1 a$

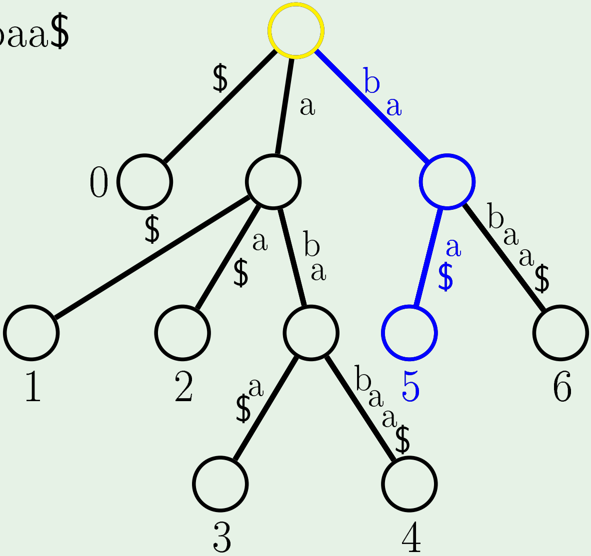
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 \$

1 a\$

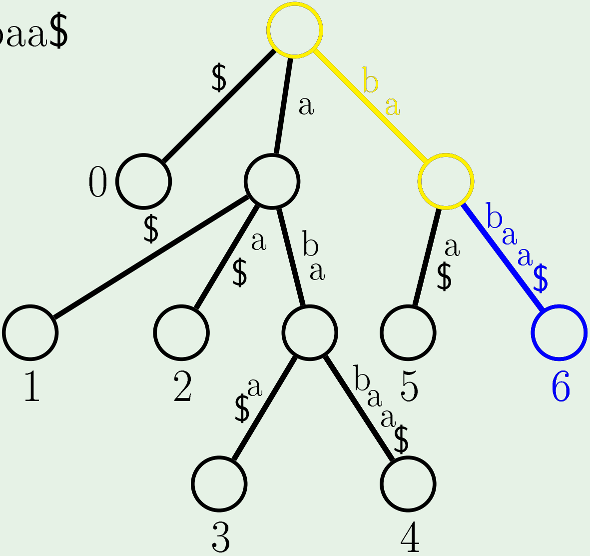
2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$



LCP array

Definition

Consider suffix array A of string S in the raw form, that is $A[0] < A[1] < A[2] < \cdots < A[|S| - 1]$ are all the suffixes of S in lexicographic order. LCP array of string S is the array lcp of size $|S| - 1$ such that for each i such that $0 \leq i \leq |S| - 2$,

$$\mathrm{lcp}[i] = \mathrm{LCP}(A[i], A[i+1])$$

LCP array

$S = \text{ababaa\$}$

0 \$

1 a\$

2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$

$\text{lcp} = [\ ,\ ,\ ,\ ,\ ,\ ]$

LCP array

$S = \text{ababaa\$}$

0 \$

1 a\$

2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$

$\text{lcp} = [\ ,\ ,\ ,\ ,\ ,\ ]$

LCP array

$S = ababaa\$$

0 $\$$

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [0, \ , \ , \ , \ , \ ]$

LCP array

$S = \text{ababaa\$}$

0 \$

1 a\$

2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$

$$\text{lcp} = [0, 1, \ , \ , \ , \ ]$$

LCP array

$S = ababaa\$$

0 $\$$

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

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

LCP array

$S = ababaa\$$

0 $\$$

1 a$\$$

2 aa$\$$

3 abaa$\$$

4 ababaa$\$$

5 baa$\$$

6 babaa$\$$

$$lcp = [0, 1, 1, 3, \ , \ ]$$

LCP array

$S = \text{ababaa\$}$

0 \$

1 a\$

2 aa\$

3 abaa\$

4 ababaa\$

5 baa\$

6 babaa\$

$\text{lcp} = [0, 1, 1, 3, 0, \quad]$

LCP array

$S = ababaa\$$

0 $\$$

1 a$\$$ $\quad\quad\quad\quad\quad$ $lcp = [0, 1, 1, 3, 0, 2]$

2 aa$\$$

3 abaa$\$$

4 ababaa$\$$

5 baa$\$$

6 babaa$\$$

LCP array property

Lemma

For any $i < j$, $LCP(A[i], A[j]) \le lcp[i]$ and $LCP(A[i], A[j]) \le lcp[j-1]$.

And the central LCP array property which will enable us to compute it fast is that for any end assist i and j In the suffix array, where i is less than j. The longest common prefix between A[i] andA[j] which are far from each other, is not bigger than the LCP of i, which is basically the longest common prefix of i and the next element. So what I'm saying with this Lemma is that the LCP of two neighboring elements is always at least as big as the LCP of the first one of them with any of the next elements. And the same goes the other way. The LCP of two neighboring elements is at least the same as LCP of the second of them with any of the previous ones.

Proof

	$\dots$
i	ababababa
i + 1	abababc
	$\dots$
j	abbcabab

Proof

	$\ldots$
i	ababababa
i + 1	abababc
	$\ldots$
j	abbcabab

Proof

	…
i	ababababa
i + 1	xxxxxxxxx
	…
j	abbcabab

If $\mathrm{LCP}(A[i], A[j]) > \mathrm{LCP}(A[i], A[i+1])$

Proof

$\dots$

i ababababa

$i+1$ xxxxxxxxx $k = 1$

$\dots$

j abbcabab

If $\mathrm{LCP}(A[i], A[j]) > \mathrm{LCP}(A[i], A[i+1])$

Consider $k = \mathrm{LCP}(A[i], A[i+1])$

Proof

	$\dots$
i	ababababa
$i+1$	a_ $k = 1$
	$\dots$
j	abbcabab

If $k = |A[i+1]|$, then $A[i+1] < A[i]$ – contradiction

Proof

$\ldots$

i	ababababa
$i+1$	axxxxxxxx $k = 1$
	$\ldots$
j	abbcabab

Otherwise $A[j][k] = A[i][k] \neq A[i+1][k]$

So maybe for some other situation with a suffix number i+1, it could be solved that the common prefix of i and j will be bigger than common prefix of i and i+1. So let's suppose that, and we don't know what is suffix i + 1,so we just replace it with many x. X is an unknown letter. We know that the LCP of i and j is equal to 2. So let's consider k which is the length of the longest common prefix of A[i] and A[i + 1]. And we suppose that it is smaller than 2 in this case. So how can that be? One variant is if A[i +1] is shorter than 2, and then A[i + 1] is actually a prefix of A[i]. But in this case, A[i+1] is smaller thanAi which contradicts the property of the suffix array. That the suffixes are sorted. And if suffix i+1 is sufficiently long then it follows that it's kth character is different from the kth character of both ith suffix and jth suffix. And in this case there are again two cases. In the first case is that this character in suffix i+1 is bigger than the corresponding one in strings i and j. But from this it immediately follows that suffix i+1 is bigger than suffix j which contradicts the suffix array properties, so it is impossible. And another case is that this character is less than the corresponding character in both strings i and j. But in this case it immediately follows that A[i] is bigger than A[i + 1] which again contradicts the suffix array property. So in all cases we found the contradiction. And so, it is not possible that the longest common prefix of i and j is bigger than the longest common prefix of i and i+1. And we proved the LCParray property because for this symmetric case,the proof is a null x.

Proof

$\dots$

i ababababa

i + 1 acxxxxxxx $k = 1$

$\dots \bigvee$

j abbcabab

If $A[j][k] = A[i][k] < A[i+1][k]$, then

$A[j] < A[i+1]$ — contradiction

Proof

$$\begin{array}{ll} & \dots \\ i & \text{abababababa} \\ & \quad\;\; \vee \\ i+1 & \text{aaxxxxxxx} \quad k = 1 \\ & \dots \\ j & \text{abbcabab} \end{array}$$

Which of the following is true?

- $LCP(A[i], A[j]) = \max_{k=i..j-1} LCP(A[k], A[k+1])$
- $LCP(A[i], A[j]) = \sum_{k=i..j-1} LCP(A[k], A[k+1])$
- $LCP(A[i], A[j]) = \prod_{k=i..j-1} LCP(A[k], A[k+1])$
- $LCP(A[i], A[j]) = \min_{k=i..j-1} LCP(A[k], A[k+1])$

Correct

Correct! If $m = \min_{k=i..j-1} LCP(A[k], A[k+1])$, then for all k between i and $j-1$ $LCP(A[k], A[k+1]) \geq m$, so there is a prefix of length m common for all suffixes $A[i], A[i+1], \dots, A[j]$, so $LCP(A[i], A[j]) \geq m$. However, if $LCP(A[k], A[k+1]) = m$ for some k between i and $j-1$, then character number $m+1$ has changed between k-th and $k+1$-th suffix and so it cannot be the same in $A[i]$ and $A[j]$, so $LCP(A[i], A[j]) \leq m$. Thus, $LCP(A[i], A[j]) = m$.

If $A[i][k] > A[i+1][k]$, then $A[i] > A[i+1]$

— contradiction $\square$

Computing LCP array

- For each i, compute $\mathrm{LCP}(A[i], A[i+1])$ via comparing $A[i]$ and $A[i+1]$ character-by-character
- $O(|S|)$ for each i, $O(|S|)$ different i — total time $O(|S|^2)$
- How to do this faster?

Outline

Idea

Lemma

Let h be the longest common prefix between S_{i-1} and its adjacent (next) suffix in the suffix array of string S. Then the longest common prefix between S_i and its adjacent (next) suffix in the suffix array is at least $h - 1$.

$$S = \text{abracadabra\$}$$

index	sorted suffix	LCP
$\ldots$	$\ldots$	$\ldots$
$i = 10$	a$	
7	abra$	
$\ldots$	$\ldots$	$\ldots$
$j = 3$	acadabra$	
$\ldots$	$\ldots$	$\ldots$
$i - 1 = 9$	ra$	
$j - 1 = 2$	racadabra$	

$$S = \text{abracadabra\$}$$

index	sorted suffix	LCP
$\dots$	$\dots$	$\dots$
$i = 10$	a$	
7	abra$	
$\dots$	$\dots$	$\dots$
$j = 3$	acadabra$	
$\dots$	$\dots$	$\dots$
$i - 1 = 9$	ra$	$h = 2$
$j - 1 = 2$	racadabra$	

$$S = \text{abracadabra\$}$$

index	sorted suffix	LCP
...	...	...
$i = 10$	a$	
7	abra$	
...	...	...
$j = 3$	acadabra$	
...	...	...
$i - 1 = 9$	ra$	$h = 2$
$j - 1 = 2$	racadabra$	

$$S = \mathrm{abracadabra\$}$$

index	sorted suffix	LCP
$\ldots$	$\ldots$	$\ldots$
$i = 10$	a$	$1 \geq h - 1$
7	abra$	
$\ldots$	$\ldots$	$\ldots$
$j = 3$	acadabra$	
$\ldots$	$\ldots$	$\ldots$
$i - 1 = 9$	ra$	$h = 2$
$j - 1 = 2$	racadabra$	

Idea

- Start by computing $\mathrm{LCP}(A[0], A[1])$ directly

Compare the smallest first two suffixes directly character by character

- Instead of computing to $\mathrm{LCP}(A[1], A[2])$, move $A[0]$ one position to the right in the string, get some $A[k]$ and compute $\mathrm{LCP}(A[k], A[k+1])$
- Repeat this until LCP array is fully computed

Length of the LCP never decreases by more than one each iteration

- Length of the LCP never decreases by more than one each iteration

Notation

- Let $A_{n(i)}$ be the suffix starting in the next position in the string after A[i]

Example

a	b	a	b	d	a	b	c

- $A[0] =$ "ababdabc", $A[1] =$ "abc"
- Compute $LCP(A[0], A[1]) = 2$ directly
- $LCP(A_{n(0)}, A_{n(1)}) \geq LCP(A[0], A[1]) - 1$
- $A[0] < A[1] \Rightarrow A_{n(0)} < A_{n(1)}$
- LCP of $A_{n(0)}$ with the next in order $A[j]$ is also at least $LCP(A[0], A[1]) - 1$

Example

a	b	a	b	d	a	b	c

- $A[0] = $ "ababdabc", $A[1] = $ "abc"
- Compute $LCP(A[0], A[1]) = 2$ directly
- $LCP(A_{n(0)}, A_{n(1)}) \geq LCP(A[0], A[1]) - 1$
- $A[0] < A[1] \Rightarrow A_{n(0)} < A_{n(1)}$
- LCP of $A_{n(0)}$ with the next in order $A[j]$ is also at least $LCP(A[0], A[1]) - 1$

Example

a	b	a	b	d	a	b	c

An(0) is babdabc and An(1) is bc

- $\mathrm{LCP}(A_{n(0)}, A_{n(1)}) \geq \mathrm{LCP}(A[0], A[1]) - 1$
- $A[0] < A[1] \Rightarrow A_{n(0)} < A_{n(1)}$
- LCP of $A_{n(0)}$ with the next in order $A[j]$ is also at least $\mathrm{LCP}(A[0], A[1]) - 1$

  Compute LCP(An(0), A[j]) directly,
  but don't compare first LCP(A[0], A[1]) - 1 characters: they are equal

- Compute $\mathrm{LCP}(A_{n(0)}, A[j])$ directly,

Algorithm

- Compute LCP(A[0], A[1]) directly, save as lcp
- First suffix goes to the next in the string
- Second suffix is the next in the order
- Con
  lcp -
- Rep

Question

What is the asymptotically fastest way to implement going from the current suffix to the next one in the suffix array on each iteration?

- ○ Go through the suffix array to find the current suffix, then take the next element of the suffix array. This takes in $O(|S|^2)$ for all iterations combined.
- ○ On each iteration if we are currently at suffix S_k, go to the suffix S_{k+1}.
- ○ Go through the suffix array to find the current suffix, then take the next element of the suffix array. This takes in $O(|S|$ for all iterations combined.
- ◉ Before doing iterations, precompute array pos - "inverted suffix array", such that for any i $pos[order[i]] = order[pos[i]] = i$. It can be precomputed in time $O(|S|)$. Then on each iteration to go from suffix S_k to the next one in the suffix array, take suffix with index $pos[k]+1$ in the suffix array (that is, $A[pos[k]+1]$). This takes $O(1)$ per iteration, so together with precomputation this is $O(|S|)$ for all iterations combined.

✓ **Correct**

Correct!

LCPOfSuffixes(S, i, j, equal)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

InvertSuffixArray(order)

pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos

Before doing iterations, pre compute array pos - "inverted suffix array", such that for any i pos[order[i]] = order[pos[i]] = i. It can be pre computed in time O(|S|). Then on each iteration to go from suffix Sk to the next one in the suffix array, take suffix with index pos[k] + 1 in the suffix array that is (A[pos[k] + 1]). This takes O(1) per iteration, so together with pre computation this is O(|S|) for all iterations combined.

ComputeLCPArray(S, order)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

Analysis

Lemma

This algorithm computes LCP array in $O(|S|)$

Proof

- Each comparison increases lcp
- lcp $\leq |S|$
- Each iteration lcp decreases by at most 1
- Number of comparisons is $O(|S|)$ $\square$

But note that during each iteration not just one character is compared but LCP (A[0], A[1]) - 1 characters are not compared. Hence it can be said that it works in linear time as there are only linear number of comparisons.

Outline

Building suffix tree

S = ababaa$

6 $

5 a$

4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$

Building suffix tree

S = ababaa$

6 $

5 a$

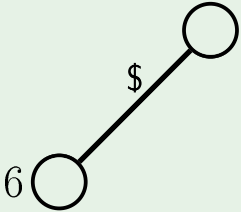
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

S = ababaa$

6 $

5 a$

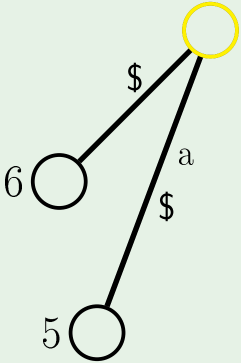
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

S = ababaa$

6 $

5 a$

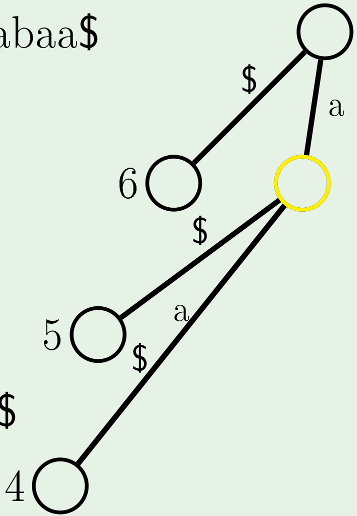
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

S = ababaa$

6 $

5 a$

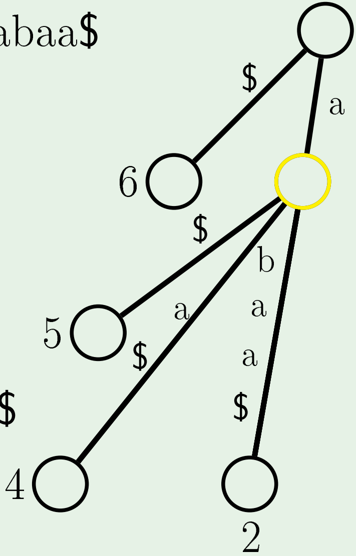
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

S = ababaa$

6 $

5 a$

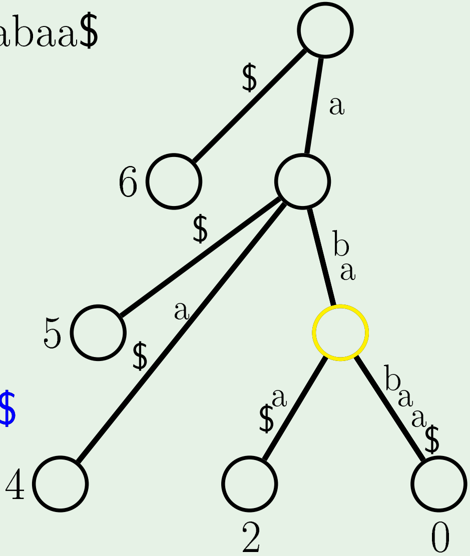
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 \$

5 a\$

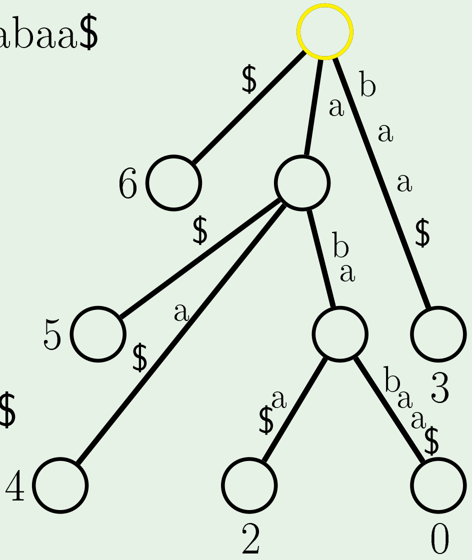
4 aa\$

2 abaa\$

0 ababaa\$

3 baa\$

1 babaa\$



Building suffix tree

S = ababaa$

6 $

5 a$

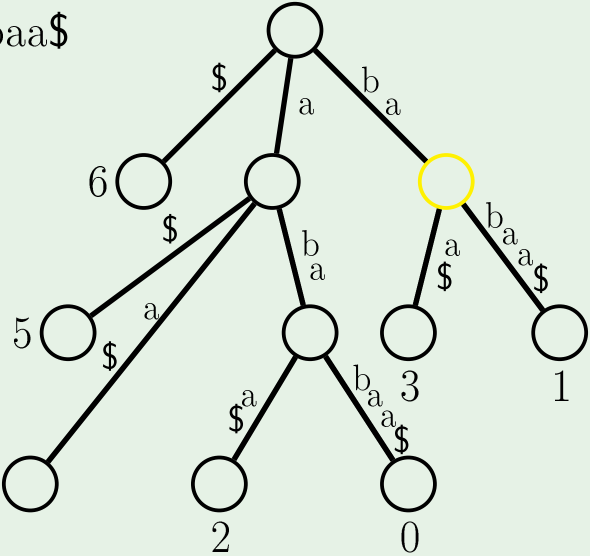
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Algorithm

- Build suffix array and LCP array
- Start from only root vertex
- Grow first edge for the first suffix
- For each next suffix, go up from the leaf until LCP with previous is below
- Build a new edge for the new suffix

class SuffixTreeNode:

SuffixTreeNode parent
Map<char, SuffixTreeNode> children
integer stringDepth
integer edgeStart
integer edgeEnd

STFromSA(S, order, lcpArray)

```
root ← new SuffixTreeNode(
   children = {}, parent = nil, stringDepth = 0,
   edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
   suffix ← order[i]
   while curNode.stringDepth > lcpPrev:
      curNode ← curNode.parent
   if curNode.stringDepth == lcpPrev:
      curNode ← CreateNewLeaf(curNode, S, suffix)
   else:
      edgeStart ← order[i − 1] + curNode.stringDepth
      offset ← lcpPrev − curNode.stringDepth
      midNode ← BreakEdge(curNode, S, edgeStart, offset)
      curNode ← CreateNewLeaf(midNode, S, suffix)
   if i < |S| − 1:
      lcpPrev ← lcpArray[i]
return root
```

$\mathrm{CreateNewLeaf(node, S, suffix)}$

```
leaf ← new SuffixTreeNode(
    children = {},
    parent = node,
    stringDepth = |S| − suffix,
    edgeStart = suffix + node.stringDepth,
    edgeEnd = |S| − 1)
node.children[S[leaf.edgeStart]] ← leaf
return leaf
```

BreakEdge(node, S, start, offset)

```
startChar ← S[start]
midChar ← S[start + offset]
midNode ← new SuffixTreeNode(
    children = {},
    parent = node,
    stringDepth = node.stringDepth + offset,
    edgeStart = start,
    edgeEnd = start + offset − 1)
midNode.children[midChar] ← node.children[startChar]
node.children[startChar].parent ← midNode
node.children[startChar].edgeStart+ = offset
node.children[startChar] ← midNode
return midNode
```

Analysis

Lemma

This algorithm runs in $O(|S|)$

Proof

- Total number of edges in suffix tree is O(|S|)
- For each edge, we go at most once down and at most once up
- Constant time to create a new edge and possibly a new node

Conclusion

- Can build suffix tree from suffix array in linear time
- Can build suffix tree from scratch in time $O(|S| \log |S|)$