Suffix Arrays

02-714

Slides by Carl Kingsford

Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

index of suffix　　suffix of s

sort the suffixes alphabetically

→

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

s = attcatg$

index of suffix	suffix of s
1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

sort the suffixes alphabetically

$\longrightarrow$

the indices just "come along for the ride"

8
5
1
4
7
3
6
2

Another Example Suffix Array

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

s = cattcat$

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

sort the suffixes alphabetically

→

the indices just "come along for the ride"

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

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

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

sort the suffixes alphabetically

$\longrightarrow$

the indices just "come along for the ride"

8
6
2
5
1
7
4
3

Search via Suffix Arrays

s = cattcat$

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

- Does string "at" occur in s?
- Binary search to find "at".

- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

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

- How many times does "at" occur in the string?
- All the suffixes that start with "at" will be next to each other in the array.
- Find one suffix that starts with "at" (using binary search).
- Then count the neighboring sequences that start with at.

K-mer counting

Problem: Given a string s, an integer k, output all pairs (b, i) such that b is a length-k substring of s that occurs exactly i times.

k = 2

		CurrentCount	
8	$	1	
6	at$	1	
2	attcat$	2	
5	cat$	1	(at,2)
1	cattcat$	2	
7	t$	1	(ca,2)
4	tcat$	1	(t$,1)
3	ttcat$	1	(tc,1)
		1	(tt,1)

1. Build a suffix array.

2. Walk down the suffix array, keeping a CurrentCount count
   - If the current suffix has length < k, skip it
   - If the current suffix starts with the same length-k string as the previous suffix:
     - increment CurrentCount
   - else
     - output CurrentCount and previous length-k suffix
     - CurrentCount := 1

Output CurrentCount & length-k suffix.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

  sort the n suffixes, which takes $O(n \log n)$ comparisons, where each comparison takes $O(n)$.

- There are several direct $O(n)$ algorithms for constructing suffix arrays that use very little space.
- The Skew Algorithm is one that is based on divide-and-conquer.

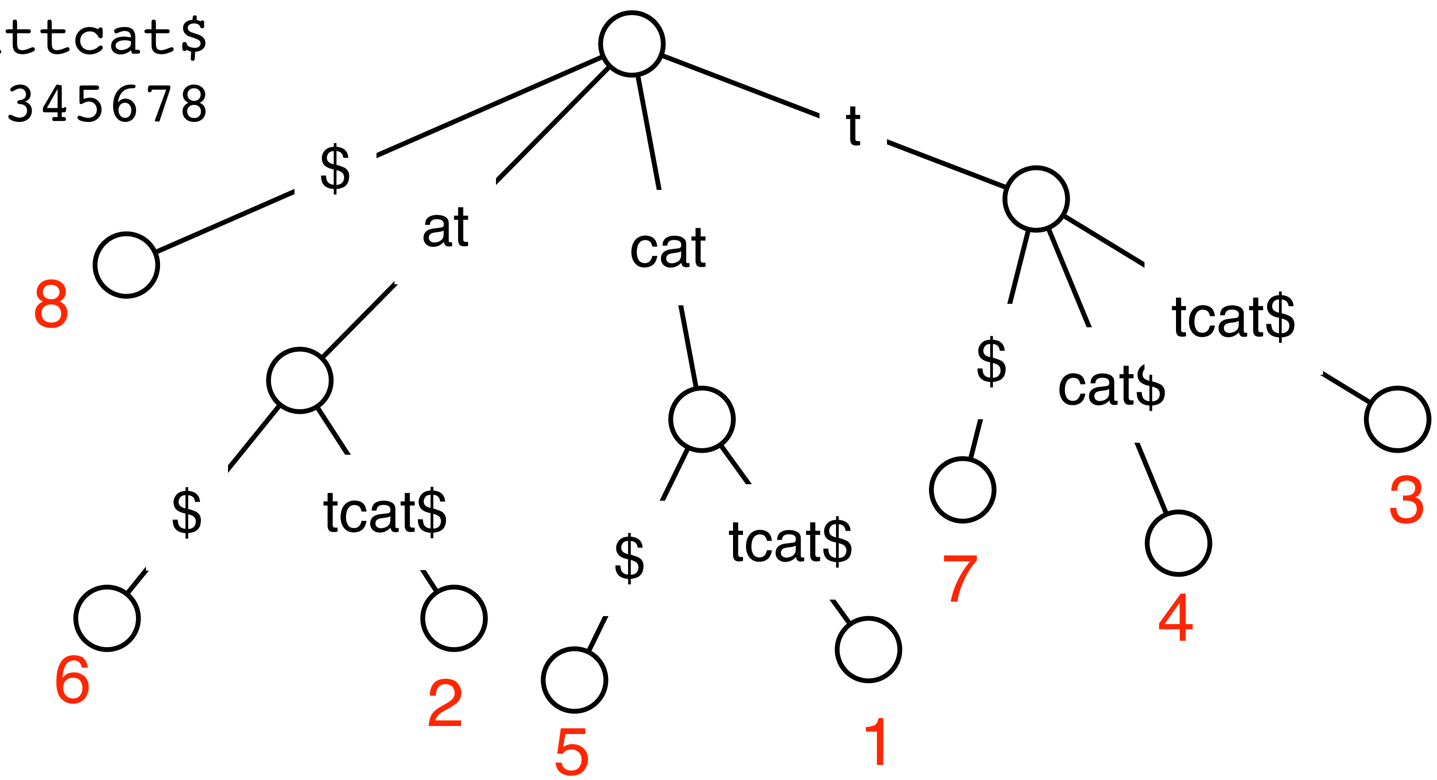
- An simple $O(n)$ algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$, a, c, t\}$

s = `cattcat$`

`12345678`



Red #s = starting position of the suffix ending at that leaf

Leaf labels left to right: 86251743

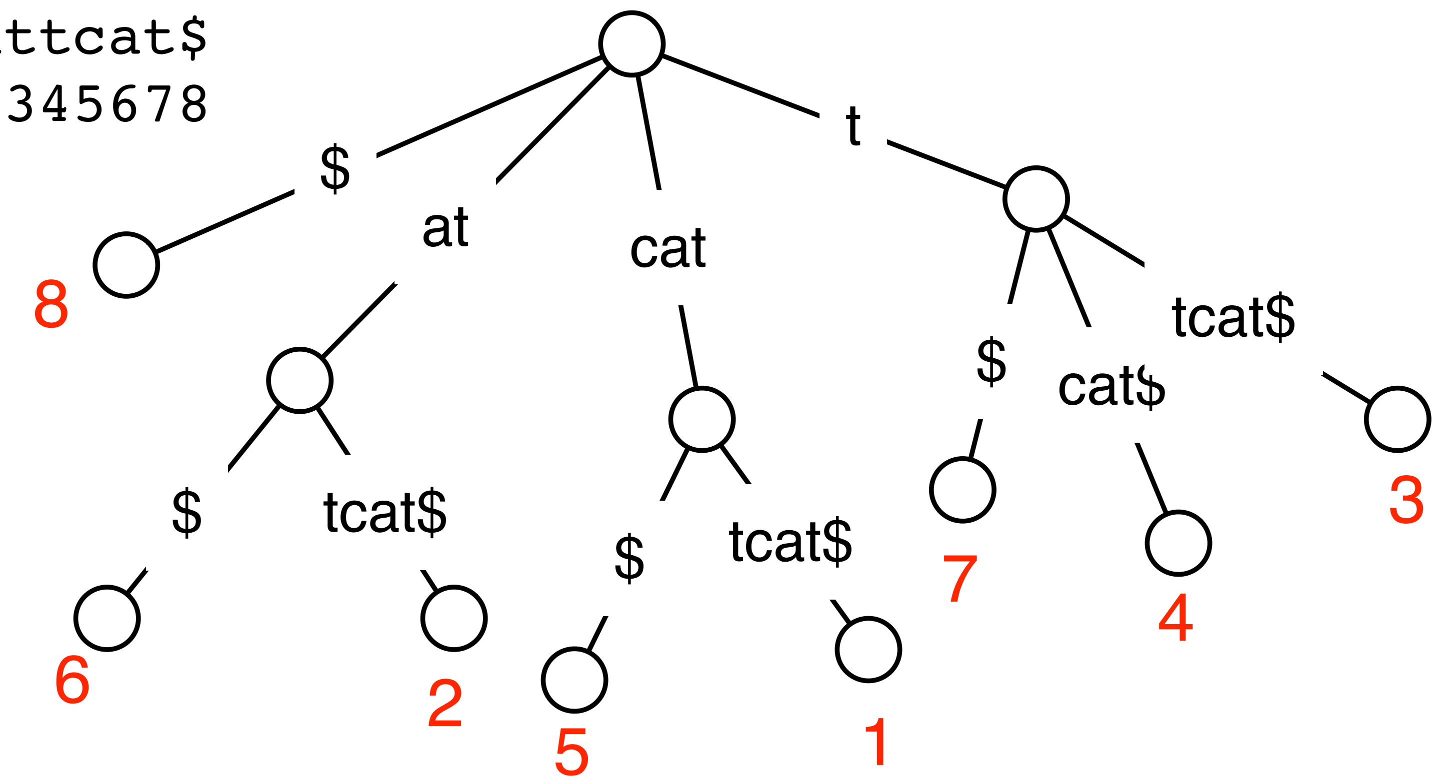
Edges leaving each node are sorted by label (left-to-right).

Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$, a, c, t\}$

s = cattcat$

12345678



s = cattcat$

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

Red #s = starting position of the suffix ending at that leaf

Edges leaving each node are sorted by label (left-to-right).

Leaf labels left to right: 86251743

Table I. Performance Summary of the Construction Algorithms

Algorithm	Worst Case	Time	Memory
Prefix-Doubling			
MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive			
KA [Ko and Aluru 2003]	$O(n)$	2.5	7–$10n$
KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	10–$13n$
KSPP [Kim et al. 2003]	$O(n)$	—	—
HSS [Hon et al. 2003]	$O(n)$	—	—
KJP [Kim et al. 2004]	$O(n \log \log n)$	3.5	13–$16n$
N [Na 2005]	$O(n)$	—	—
Induced Copying			
IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	5–$6n$
MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	9–$10n$
BB [Baron and Bresler 2005]	$O(n\sqrt{\log n})$	2.1	$18n$
M [Maniscalco and Puglisi 2007]	$O(n^2 \log n)$	1.3	5–$6n$
MP [Maniscalco and Puglisi 2006]	$O(n^2 \log n)$	1	5–$6n$
Hybrid			
IT+KA	$O(n^2 \log n)$	4.8	$5n$
BK+IT+KA	$O(n \log n)$	2.3	5–$6n$
BK+S	$O(n \log n)$	2.8	5–$6n$
Suffix Tree			
K [Kurtz 1999]	$O(n \log \sigma)$	6.3	13–$15n$

Time is relative to MP, the fastest in our experiments. Memory is given in bytes including space required for the suffix array and input string and is the average space required in our experiments. Algorithms HSS and N are included, even though to our knowledge they have not been implemented. The time for algorithm MM is estimated from experiments in Larsson and Sadakane [1999].

Puglisi, Smyth, Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

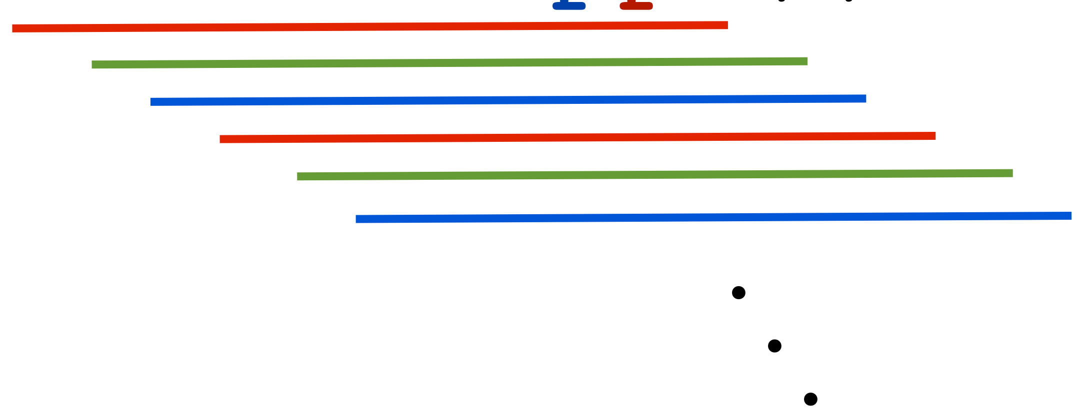
The Skew Algorithm

Kärkkäinen & Sanders, 2003

- **Main idea: Divide suffixes into 3 groups:**
  - Those starting at positions i=0,3,6,9,.... (i mod 3 = 0)
  - Those starting at positions 1,4,7,10,... (i mod 3 = 1)
  - Those starting at positions 2,5,8,11,... (i mod 3 = 2)

- For simplicity, assume text length is a multiple of 3 after padding with a special character.

mississippi$$



Basic Outline:

- Recursively handle suffixes from the i mod 3 = 1 and i mod 3 = 2 groups.
- Merge the i mod 3 = 0 group at the end.

Handing the 1 and 2 groups

`s = mississippi$$`

iss	iss	ipp	i$$	ssi	ssi	ppi

triples for groups 1 and 2 groups

`t` = C C B A E E D

assign each triple a token in lexicographical order

AEED 4
BAEED 3
CBAEED 2
CCBAEED 1
D 7
ED 6
EED 5

recursively compute the suffix array for tokenized string

4321765

Every suffix of t corresponds to a suffix of s (maybe with some cruft at the end of it).

Relationship Between t and s

s = mississippi$$

iss	iss	ipp	i$$	ssi	ssi	ppi
C	C	B	A	E	E	D

$t = \mathtt{CCBA}\underbrace{\mathtt{EED}}_{t_4}$

4321765

Key Point #1: The lexicographical order of the suffixes of t is the same as the order of the group 1 & 2 suffixes of s.

Why?

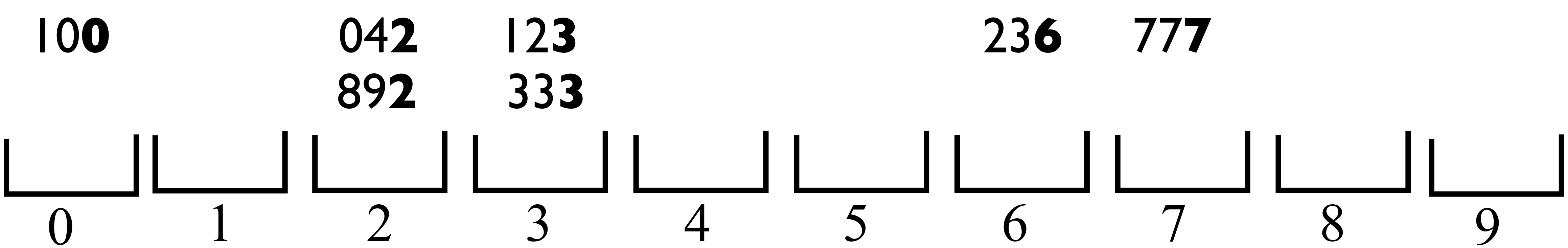
Every suffix of t corresponds to some suffix of s (perhaps with some extra letters at the end of it --- in this case EED)

Because the tokens are sorted in the same order as the triples, the sort order of the suffix of t matches that of s.

So: The recursive computational of the suffix array for t gives you the ordering of the group 1 and group 2 suffixes.

Radix Sort

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



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

Handling 0 Suffixes

- First: sort the group 0 suffixes, using the representation ($s[i]$, S_{i+1})
  - Since the S_{i+1} suffixes are already in the array sorted, we can just *stably* sort them with respect to $s[i]$, using radix sort.

1,2-array:

ipp	iss	iss	i$$	ppi	ssi	ssi

0-array:

mis	pi$	sip	sis

- We have to merge the group 0 suffixes into the suffix array for group 1 and 2.
- Given suffix S_i and S_j, need to decide which should come first.
  - If S_i and S_j are both either group 1 or group 2, then the recursively computed suffix array gives the order.
  - If one of i or j is 0 (mod 3), see next slide.

Comparing 0 suffix S_j with 1 or 2 suffix S_i

Represent S_i and S_j using subsequent suffixes:

$\underline{i \pmod 3 = 1:}$

$$(s[i], S_{i+1}) \overset{?}{<} (s[j], S_{j+1})$$

S_{i+1}: $\equiv 2 \pmod 3$ $\quad$ S_{j+1}: $\equiv 1 \pmod 3$

$\underline{i \pmod 3 = 2:}$

$$(s[i], s[i+1], S_{i+2}) \overset{?}{<} (s[j], s[j+1], S_{j+2})$$

S_{i+2}: $\equiv 1 \pmod 3$ $\quad$ S_{j+2}: $\equiv 2 \pmod 3$

⇒ the suffixes can be compared quickly because the relative order of S_{i+1}, S_{j+1} or S_{i+2}, S_{j+2} is known from the 1,2-array we already computed.

Running Time

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

time to sort and merge

array in recursive calls is 2/3rds the size of starting array

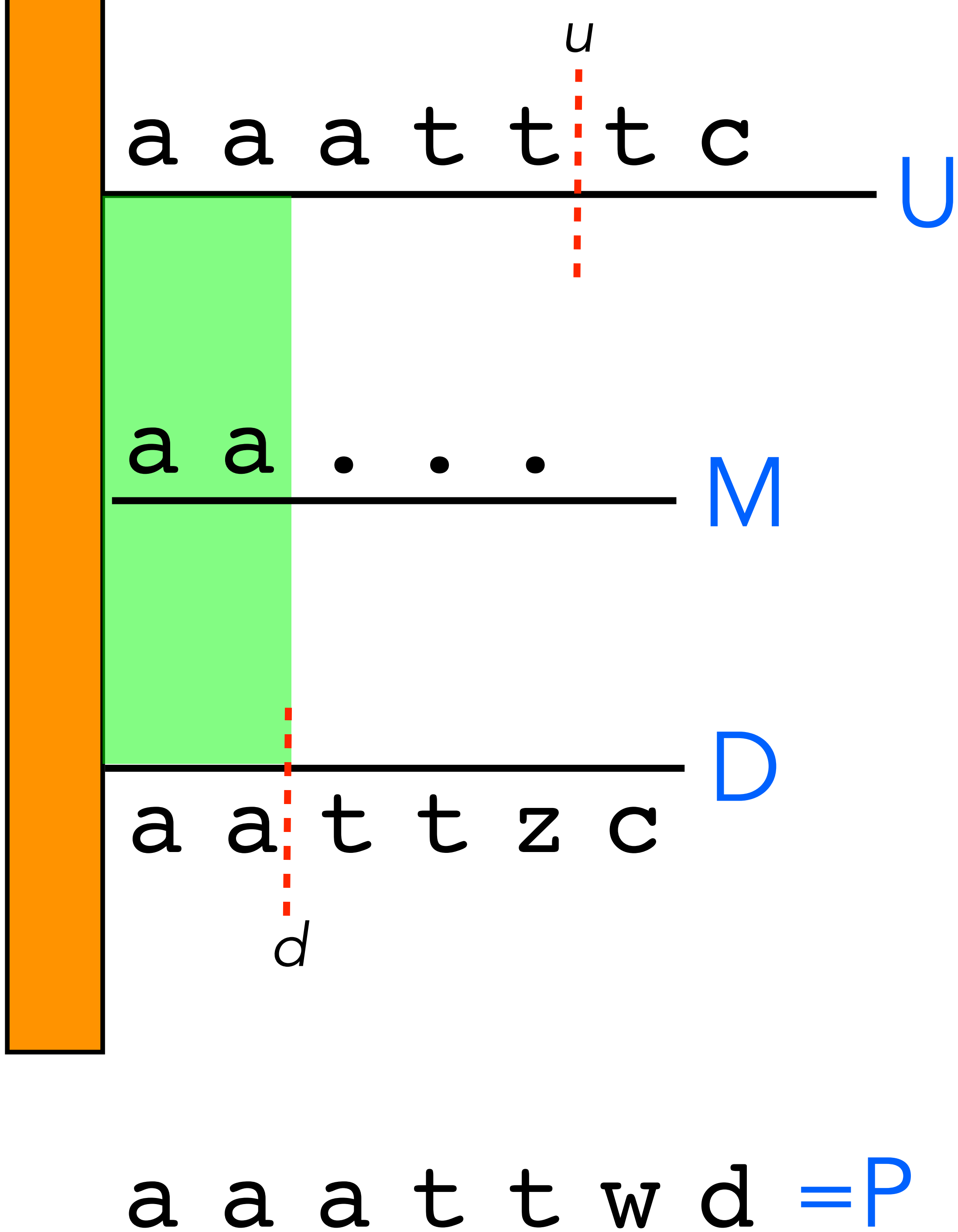
Solves to $T(n) = O(n)$:

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

Faster Suffix Array Search

- The basic binary search takes O(|P| log |T|):
  - it takes O(log |T|) iterations
  - each comparison taking O(|P|) time
- We can do it faster by avoiding the O(|P|) time for comparison if we are willing to keep some extra values associated with the array.
- Follows Gusfield, section 7.14

Speedup #1



u := length of longest prefix of U that matches a prefix of P

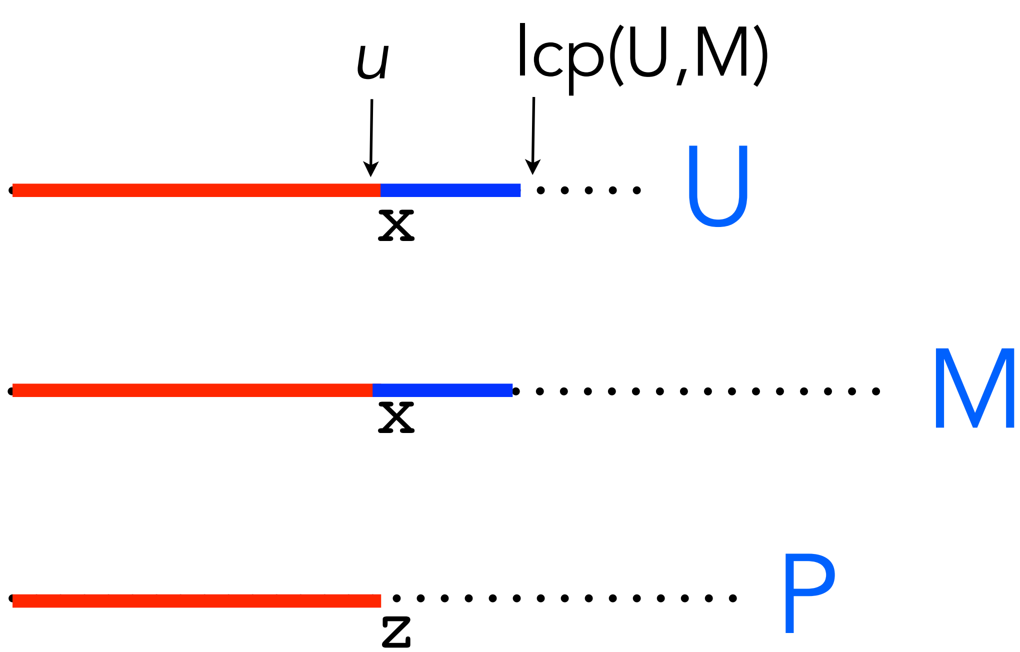
d := length of longest prefix of D that matches a prefix of P

Speedup: maintain *u*,*d* throughout algorithm. Begin comparison of P with M at position min(*u*,*d*)

Speedup #2: Lcp(i,j) array

Def. *lcp(X,Y) is the length of the longest common prefix of strings X and Y.*

Case #1: lcp(U,M) > u = lcp(U,P)

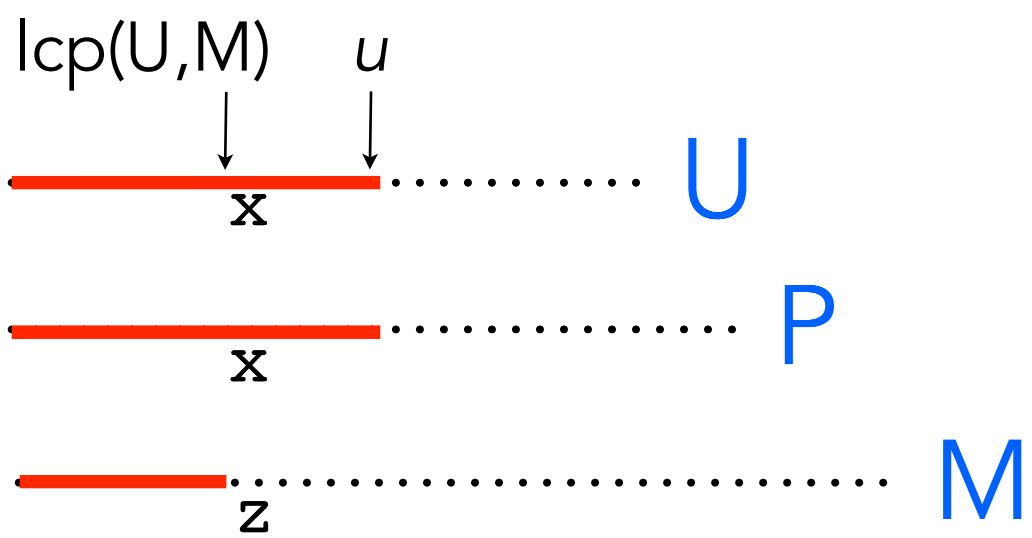


x < z because U < P (since P ∈ [U,D])

⟹ M < P and therefore the new range should be [M,D]

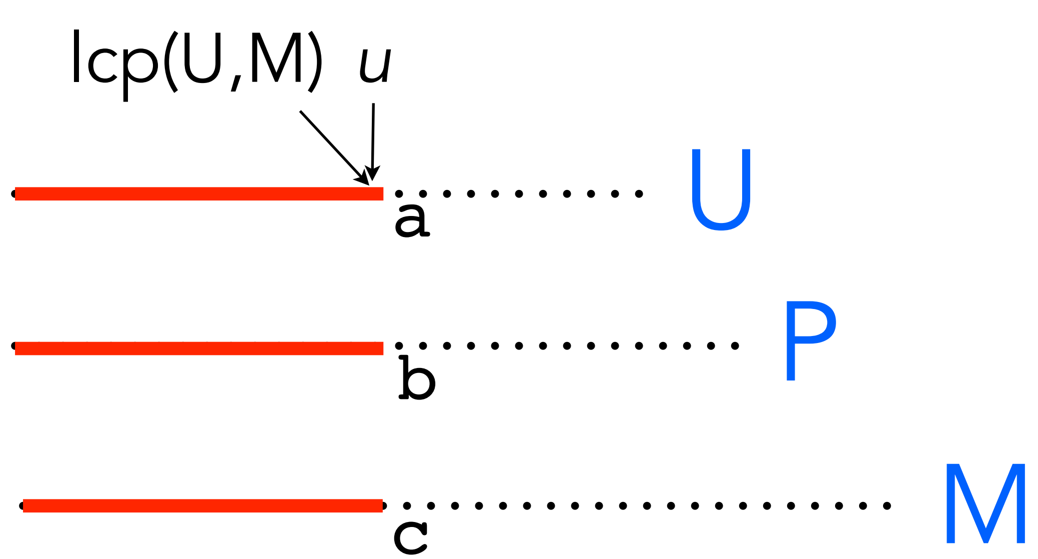
Speedup #2: Case #2 & #3

Case #2: lcp(U,M) < u = lcp(U,P)



x < z because U < M (by definition)
$\Longrightarrow$ P < M and the new range should be [U,M]

Case #3: lcp(U,M) = u = lcp(U,P)



We have no information about a,b,c but we can start comparing at position *u*.

Algorithm

Case #0: u = d: start comparing from position u+1 = d+1.

Algorithm: If u = d, apply case 0.

If u > d, apply case 1, 2 or 3 as appropriate

If u < d, apply cases 1′, 2′, or 3′ that are the symmetric version of cases 1,2,3 swapping D for U.

Running Time

Thm. *Given the lcp(X,Y) values, searching for a string P in a suffix array of length m now takes O(|P| + log m) time.*

Proof. Only cases 0 and 3 (and 3') actually compare any characters.

They always start comparing at max(u,d) [for case 0 this is trivial, for case 3 this is true b/c we assume $u > d$].

If they match k characters of P, then one of u or d will be incremented by k, and those characters will never be compared again, so there are at most O(|P|) such comparisons.

The mismatch character may be compared more than once.

But there can be only 1 mismatch / iteration. There are O(log m) iterations, so there are at most O(log m) mismatches.

$\therefore$Total # of comparisons = O(|P| + log m). $\square$

Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

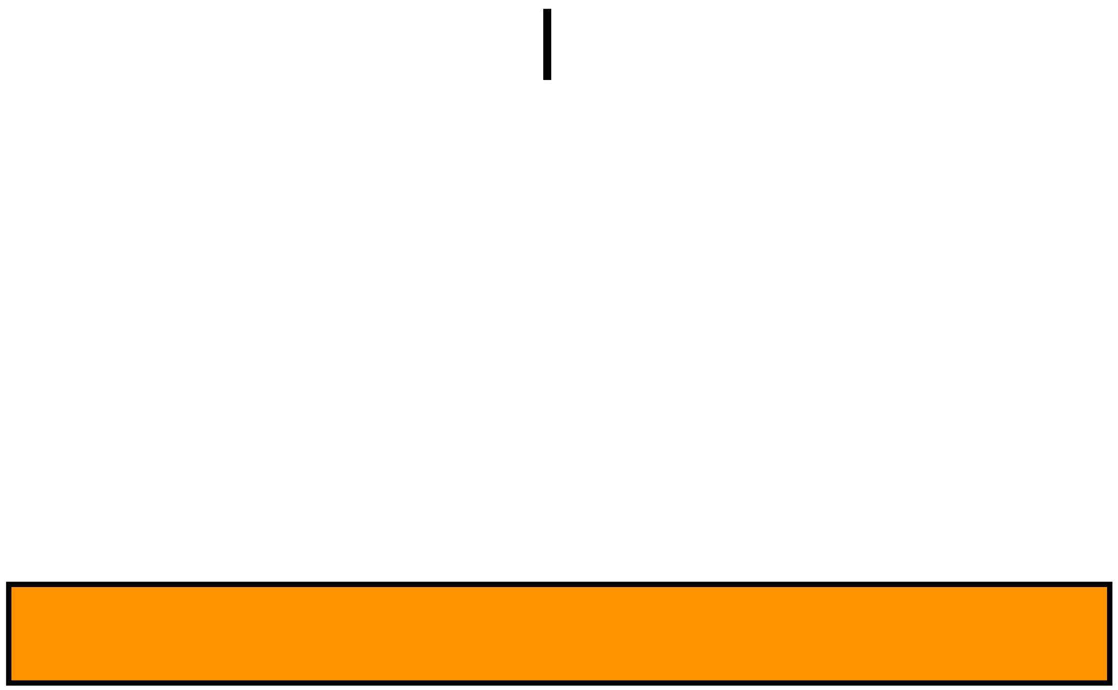
While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

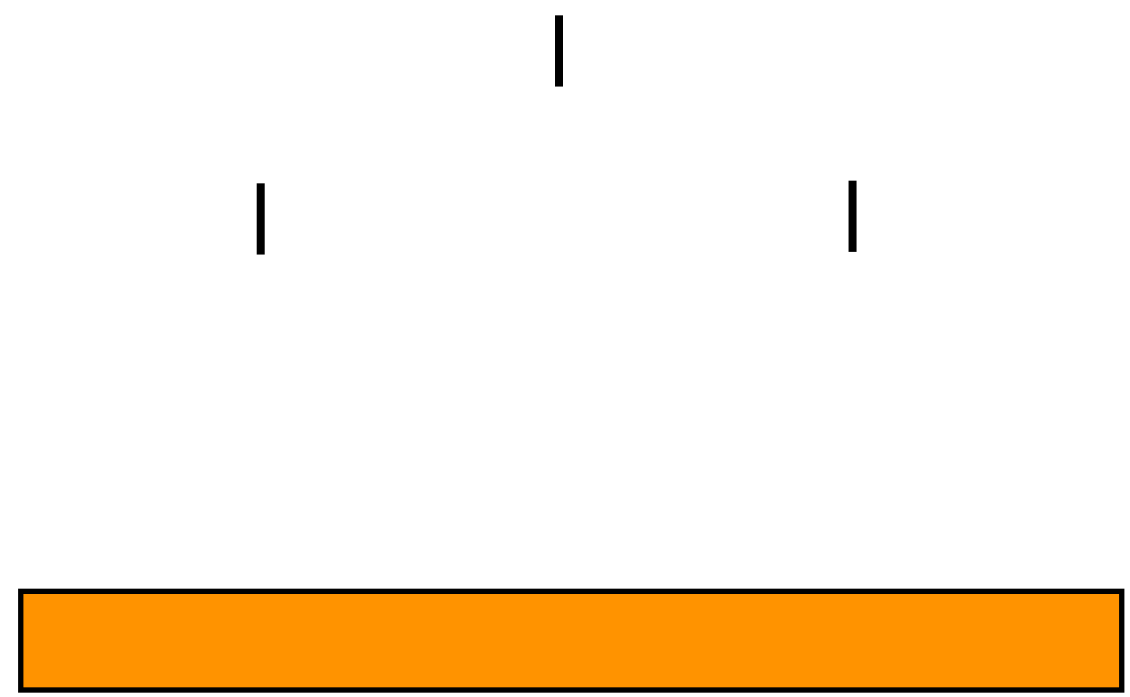


Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

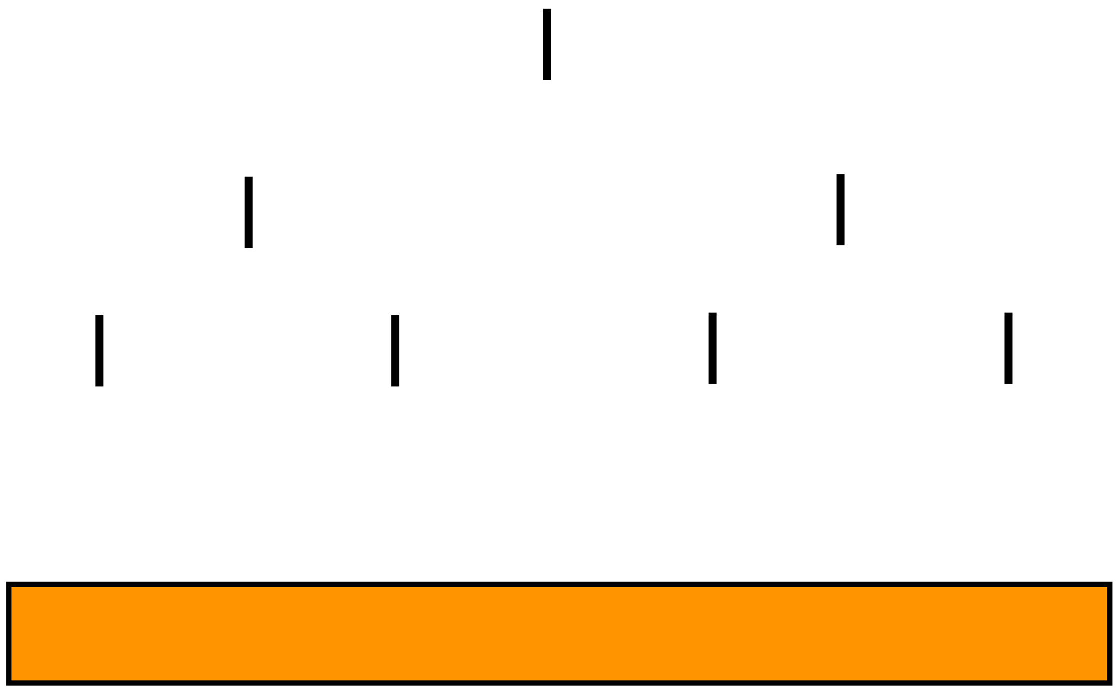


Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

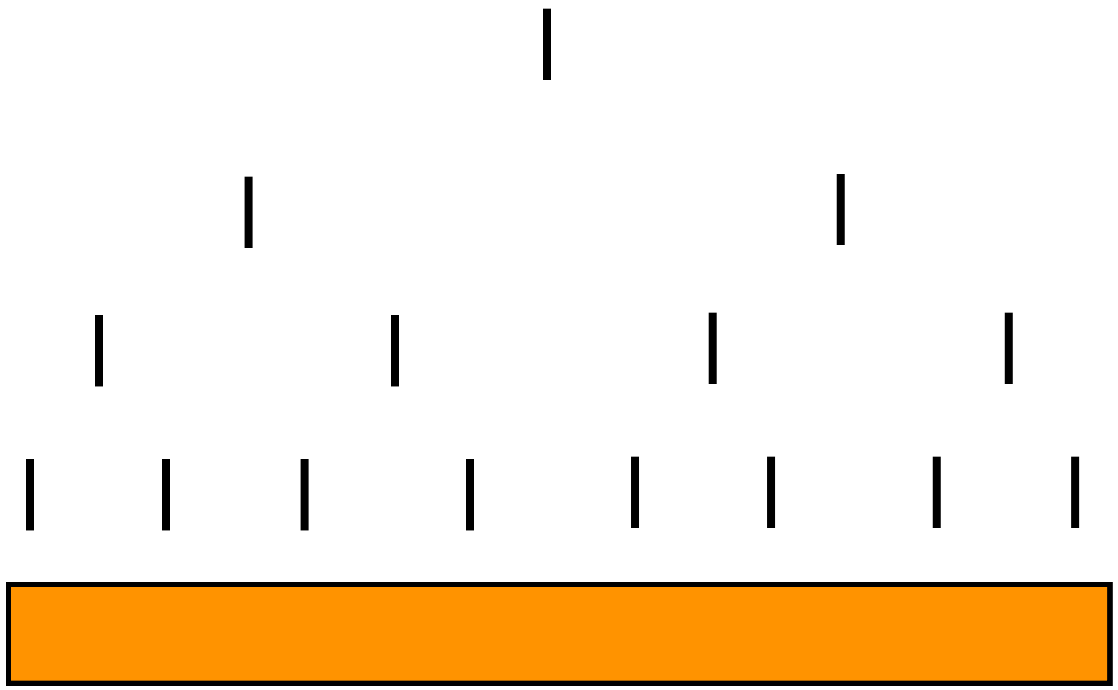


Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

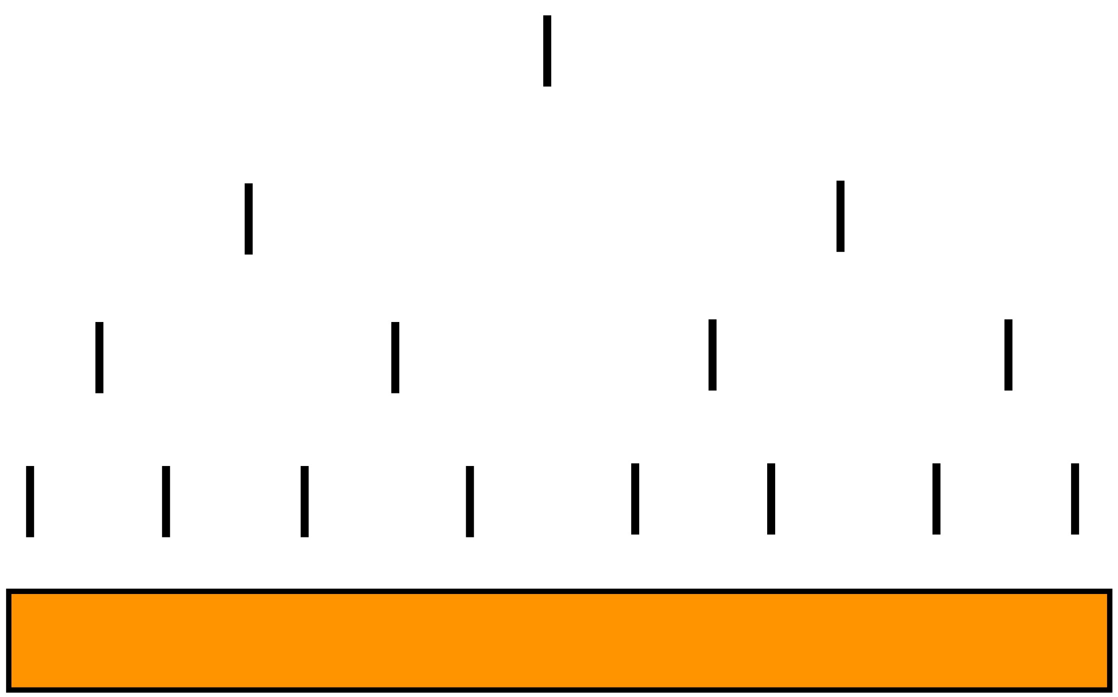


Pre-computing the Lcp(i,j) values

Notation. Lcp(i,j) := longest common prefix between A(i) and A(j), where A(i) is the suffix in position i of the suffix array A.

Lcp(i,j) values depend only on the suffix array.

While there are O(m^2) possible values, only O(m) of them will ever be accessed in *any* suffix array search. Why?

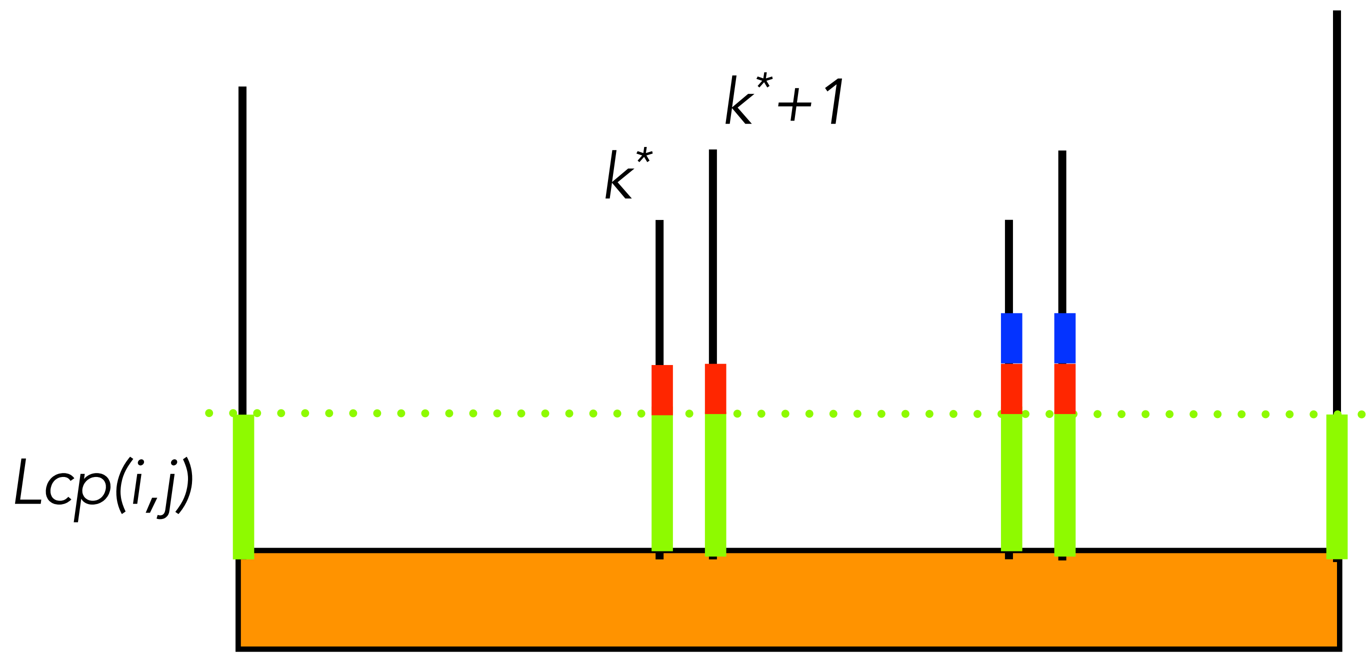


= complete binary tree with m leaves.

Has O(m) nodes entirely, so at most O(m) ranges are considered.

Computing Lcp(i, j)

Thm. *Lcp(i,j) = min Lcp(k, k+1), where k = i,...,j-1.*



Lcp(k,k+1) ≥ *Lcp(i,j)* for all *k* because everything in this range shares the same *Lcp*(i,j) prefix at least.

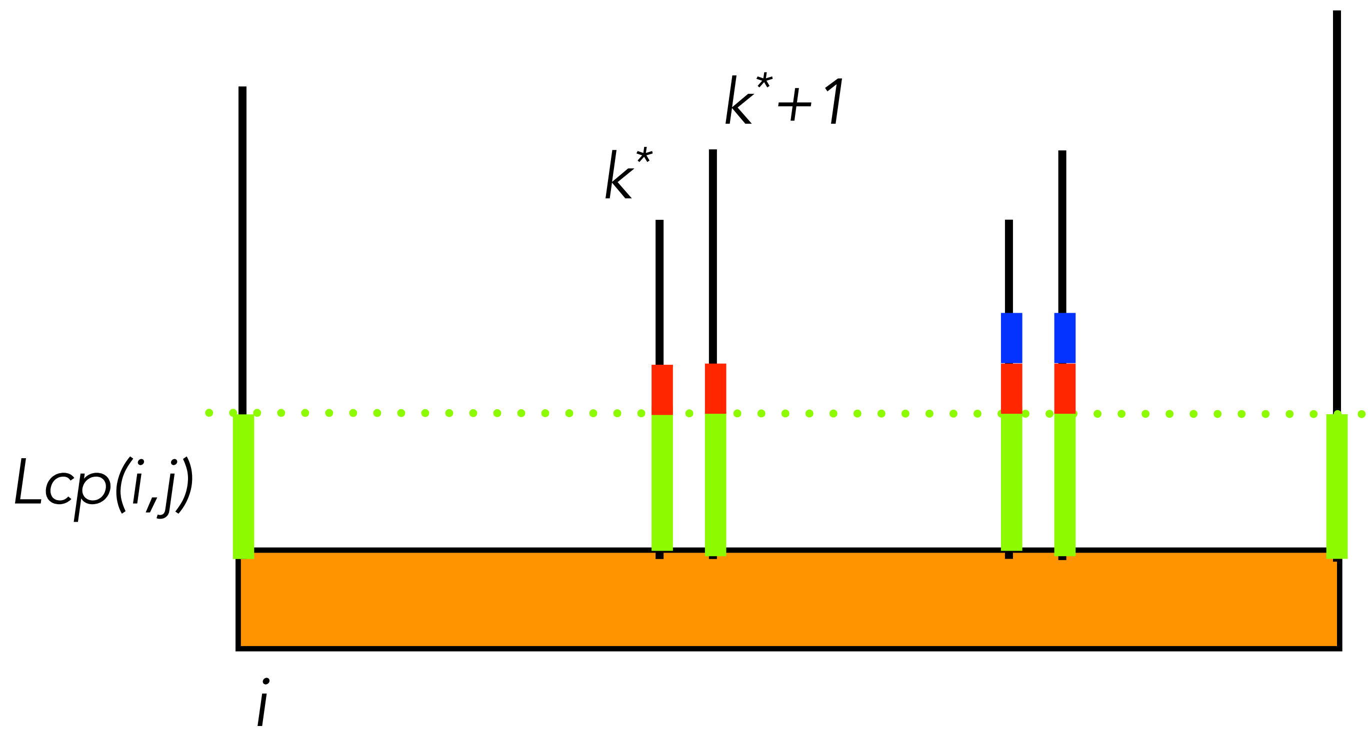
Other consecutive pairs can have larger Lcp, but not smaller than k^* by the minimality of Lcp(k^*, k^* +1).

⇒*By transitivity, Lcp(i,j) ≥ Lcp(k^*, k^* +1).*

Lcp(k,k+1) can be computed in O(m) time by traversing a suffix tree to find the depth of the lca of k and k+1.

Computing Lcp(i, j)

Thm. *Lcp(i,j) = min Lcp(k, k+1), where k = i,...,j-1.*



Lcp(k,k+1) ≥*Lcp(i,j)* for all *k* because everything in this range shares the same *Lcp*(i,j) prefix at least.

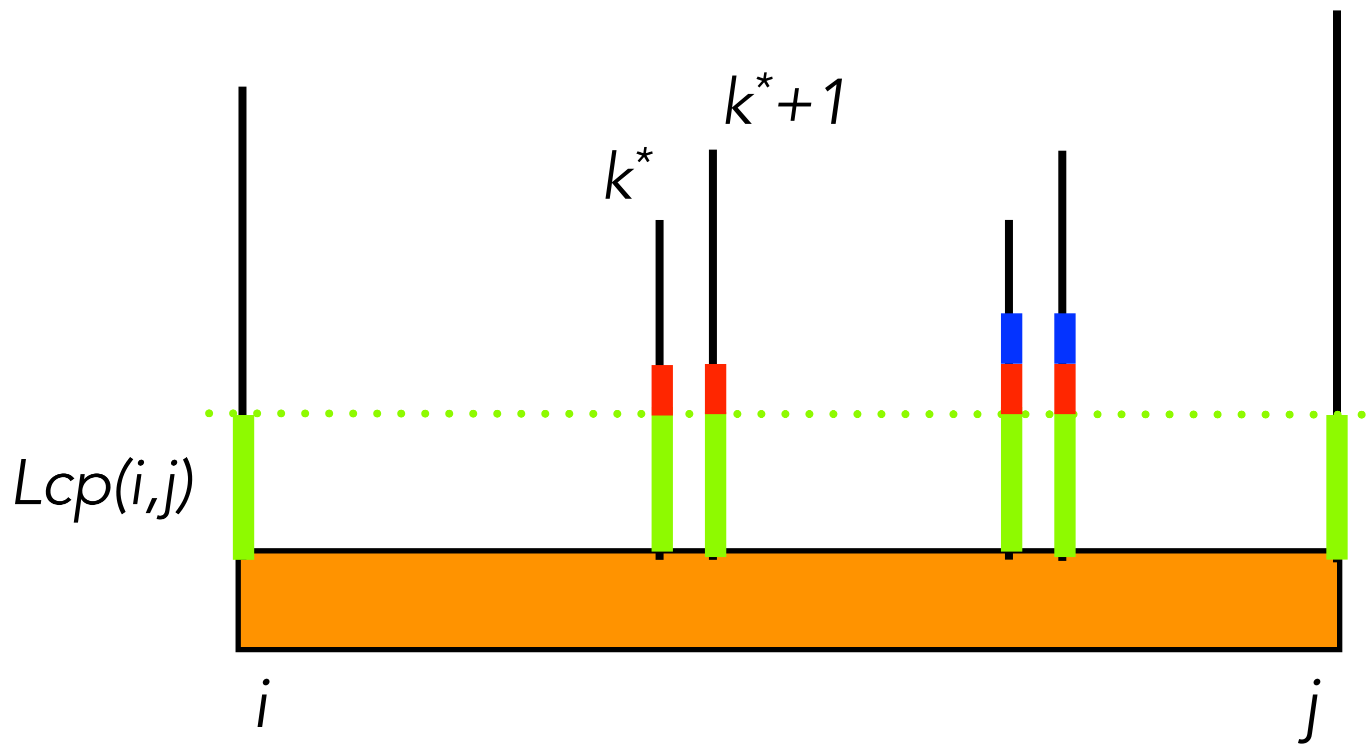
Other consecutive pairs can have larger Lcp, but not smaller than k^* *by the minimality of* $Lcp(k^*, k^*+1)$.

⇒*By transitivity,* $Lcp(i,j) \geq Lcp(k^*, k^*+1)$.

Lcp(k,k+1) can be computed in O(m) time by traversing a suffix tree to find the depth of the lca of k and k+1.

Computing Lcp(i, j)

Thm. *Lcp(i,j) = min Lcp(k, k+1), where k = i,…,j-1.*



Lcp(k,k+1) ≥*Lcp(i,j)* for all *k* because everything in this range shares the same *Lcp*(i,j) prefix at least.

Other consecutive pairs can have larger Lcp, but not smaller than k^* by the minimality of Lcp(k^*, k^* +1).

⇒*By transitivity, Lcp(i,j) ≥ Lcp(k^*, k^* +1).*

Lcp(k,k+1) can be computed in O(m) time by traversing a suffix tree to find the depth of the lca of k and k+1.

Recap

- Suffix arrays can be used to search and count substrings.
- Construction:
  - Easily constructed in $O(n^2 \log n)$
  - Simple algorithms to construct them in $O(n)$ time.
  - More complicated algorithms to construct them in $O(n)$ time using even less space.
- More space efficient than suffix trees: just storing the original string + a list of integers.