

Full-text indexing : The Suffix Array

Suffix array

$T\$ = \text{abaaba}\$$

Maintain T as part of the index

$SA(T) =$
(SA = "Suffix Array")

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

$m + 1$
integers

Suffix array of T is an array of integers in $[0, m]$ specifying the lexicographic order of $T\$$'s suffixes

Another Example Suffix Array

$s = \text{cattcat\$}$

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

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

sort the suffixes
alphabetically



the indices just
“come along for
the ride”

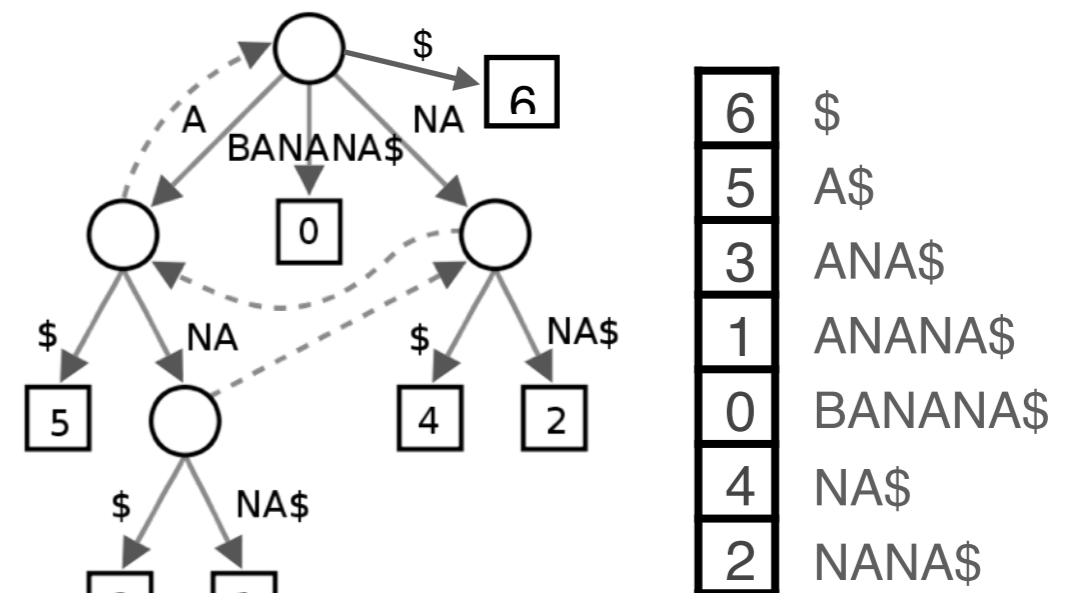
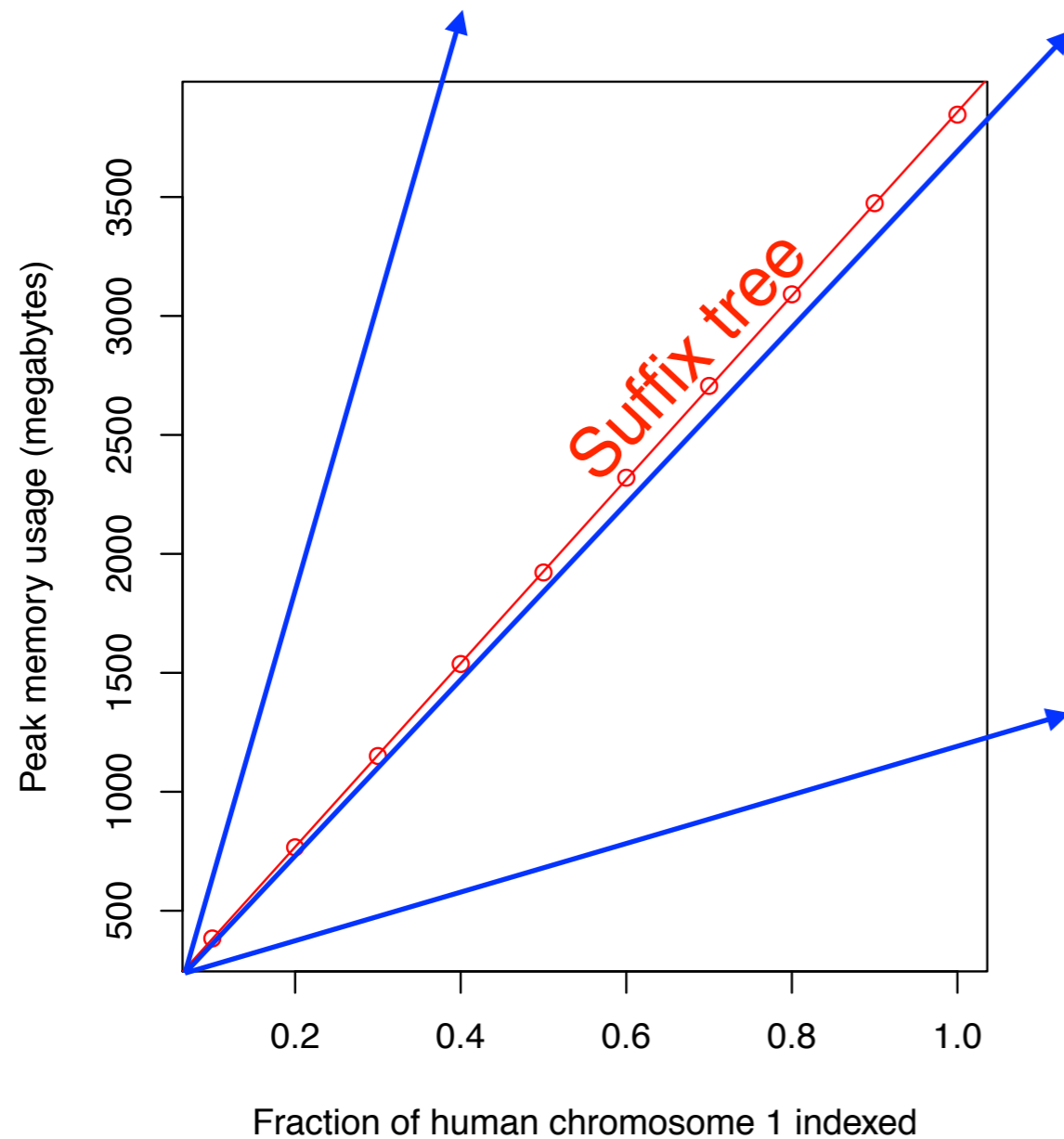
7	\$
5	at\$
1	attcat\$
4	cat\$
0	cattcat\$
6	t\$
3	tcat\$
2	ttcat\$

index of suffix

suffix of s

Suffix array

$O(m)$ space, like suffix tree Is “constant factor” worse, better, same?



Suffix array

32-bit integers sufficient for human genome, so fits in
 ~ 4 bytes/base \times 3 billion bases \approx 12 GB. Suffix tree is >45 GB.

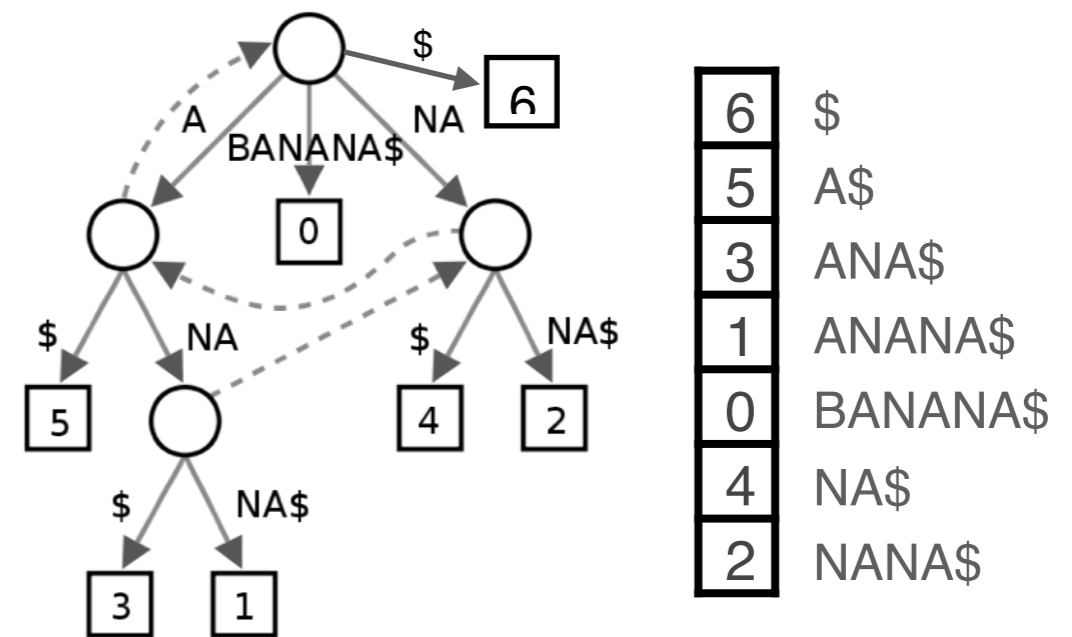
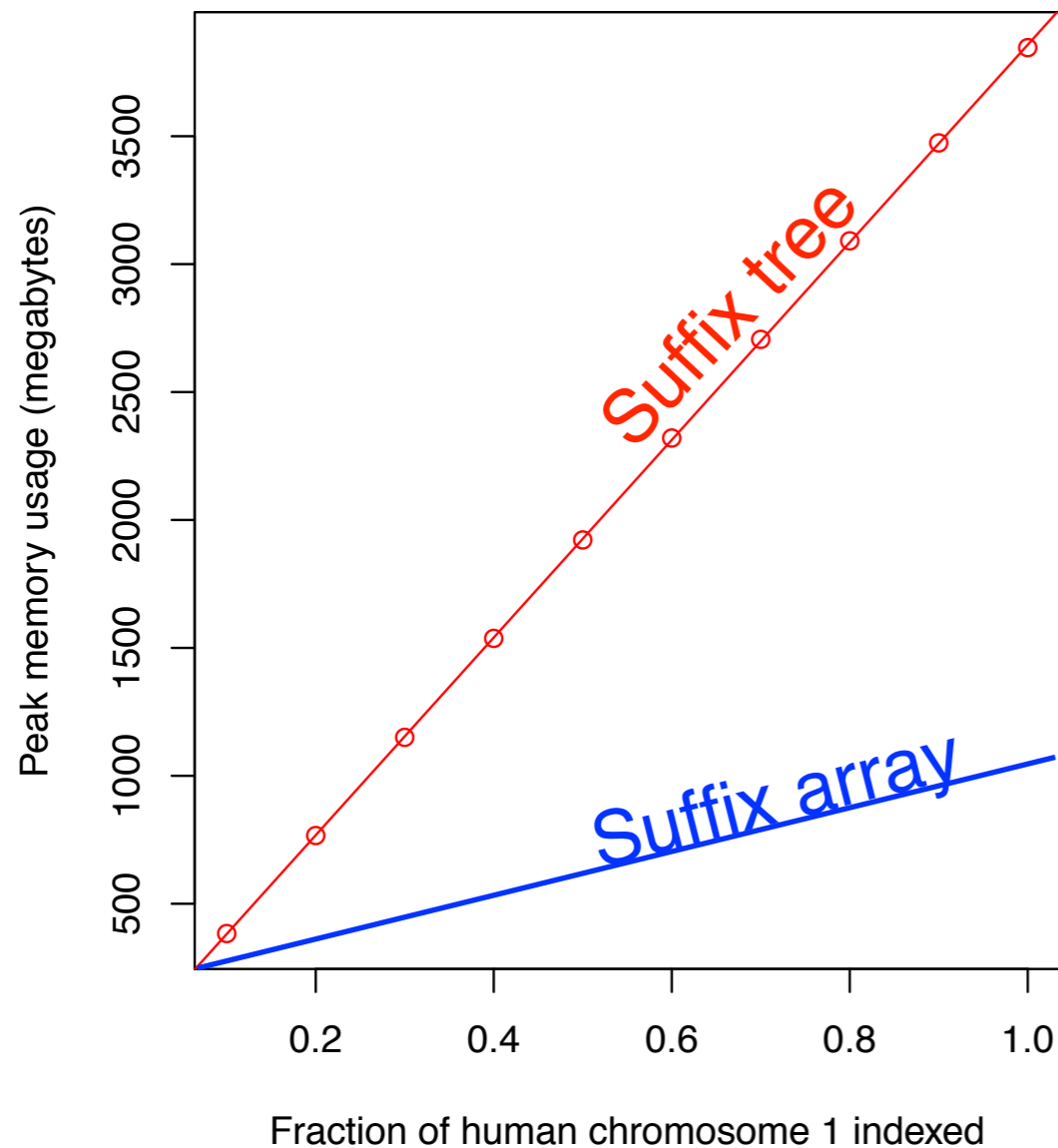


Table I. Performance Summary of the Construction Algorithms

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].

Suffix array: querying

Is P a substring of T ?

1. For P to be a substring, it must be a prefix of ≥ 1 of T 's suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

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

Suffix array: querying

Is P a substring of T ?

1. For P to be a substring, it must be a prefix of ≥ 1 of T 's suffixes
2. Suffixes sharing a prefix are consecutive in the suffix array

Use binary search

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

Suffix array: querying

Is P a substring of T ?

1. For P to be a substring, it must be a prefix of ≥ 1 of T 's suffixes

6 \$

Pay attention to this observation!

Almost all full-text indices (definitely the ones we will learn about), work based on the observation that **every substring of T is a prefix of some suffix of T** . These indices then focus on how to *organize* suffixes of T in a manner amenable to efficient search.

Suffix array: querying

Is P a substring of T ?

Do binary search, check whether P is a prefix of the suffix there

How many times does P occur in T ?

Two binary searches yield the range of suffixes with P as prefix; size of range equals # times P occurs in T

Worst-case time bound?

$O(\log_2 m)$ bisections, $O(n)$ comparisons per bisection, so $O(n \log m)$

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

Suffix array: querying

Consider further: binary search for suffixes with P as a prefix

Assume there's no $\$$ in P . So P can't be equal to a suffix.

Initialize $l = 0$, $c = \text{floor}(m/2)$ and $r = m$ (just past last elt of SA)

\uparrow \uparrow \uparrow
"left" "center" "right"

Notation: We'll use $\text{SA}[l]$ to refer to the suffix corresponding to suffix-array element l . We could write $\pi[\text{SA}[l]:]$, but that's too verbose.

Throughout the search, invariant is maintained:

$$\text{SA}[l] < P < \text{SA}[r]$$

Suffix array: querying

Throughout search, invariant is maintained:

$$SA[l] < P < SA[r]$$

What do we do at each iteration?

Let $c = \text{floor}((r + l) / 2)$

If $P < SA[c]$, either stop or let $r = c$ and iterate

If $P > SA[c]$, either stop or let $l = c$ and iterate

When to stop?

$P < SA[c]$ and $c = l + 1$ - answer is c

$P > SA[c]$ and $c = r - 1$ - answer is r

Longest Common Prefix

The longest common prefix of two strings s, t is simply the length of the prefix they share prior to the first difference (or the termination of either string).

S	ACTTACAGACCGAGAC
T	ACTTACAGACGGAGCTAGC

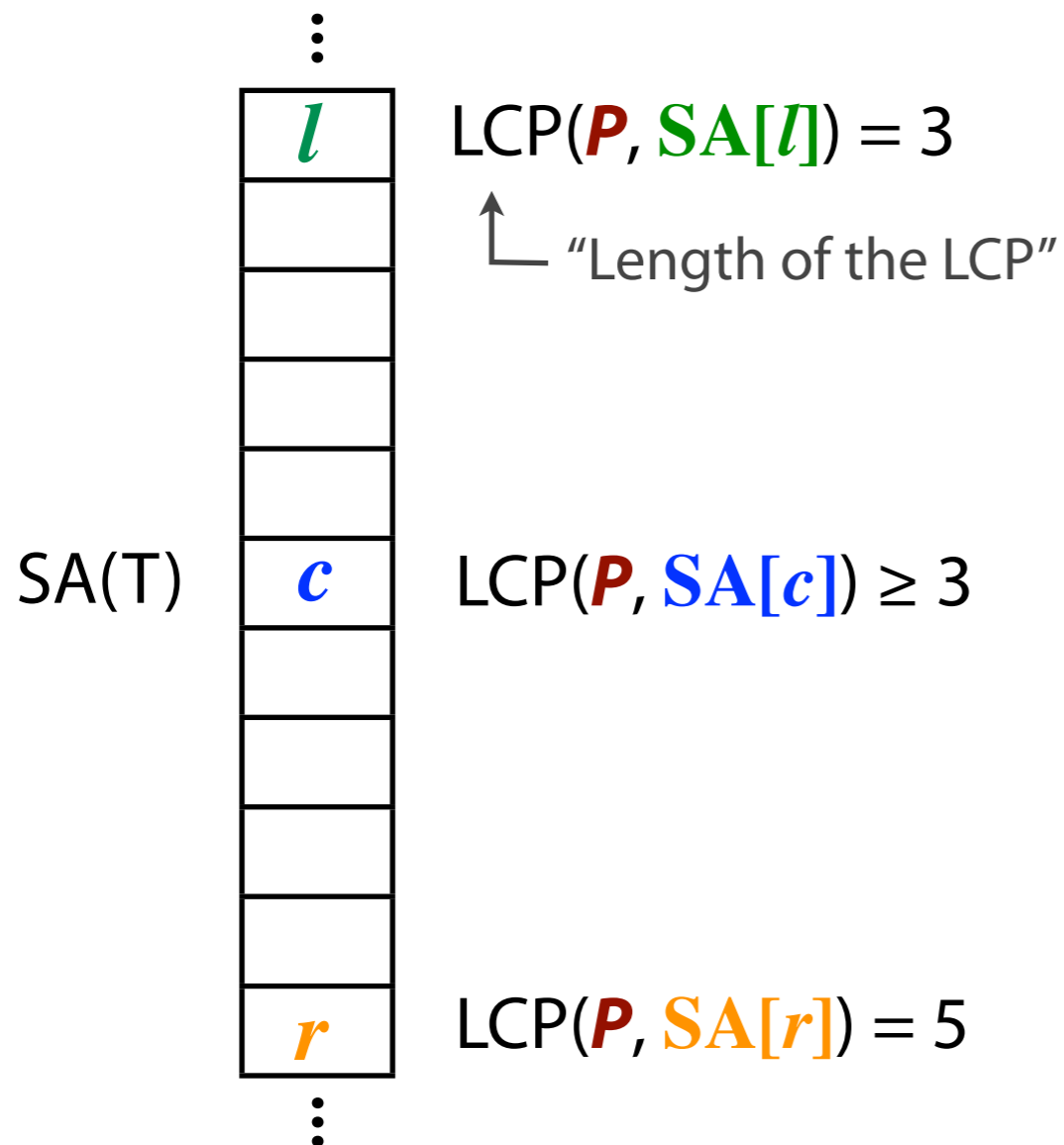
LCP(S,T) = ACTTACAGACGAC

|LCP(S,T)| = 13

Below, to avoid extra notation, we will use LCP(S,T) as shorthand for |LCP(S,T)|

Suffix array: querying

Say we're comparing P to $SA[c]$ and we've already compared P to $SA[l]$ and $SA[r]$ in previous iterations.



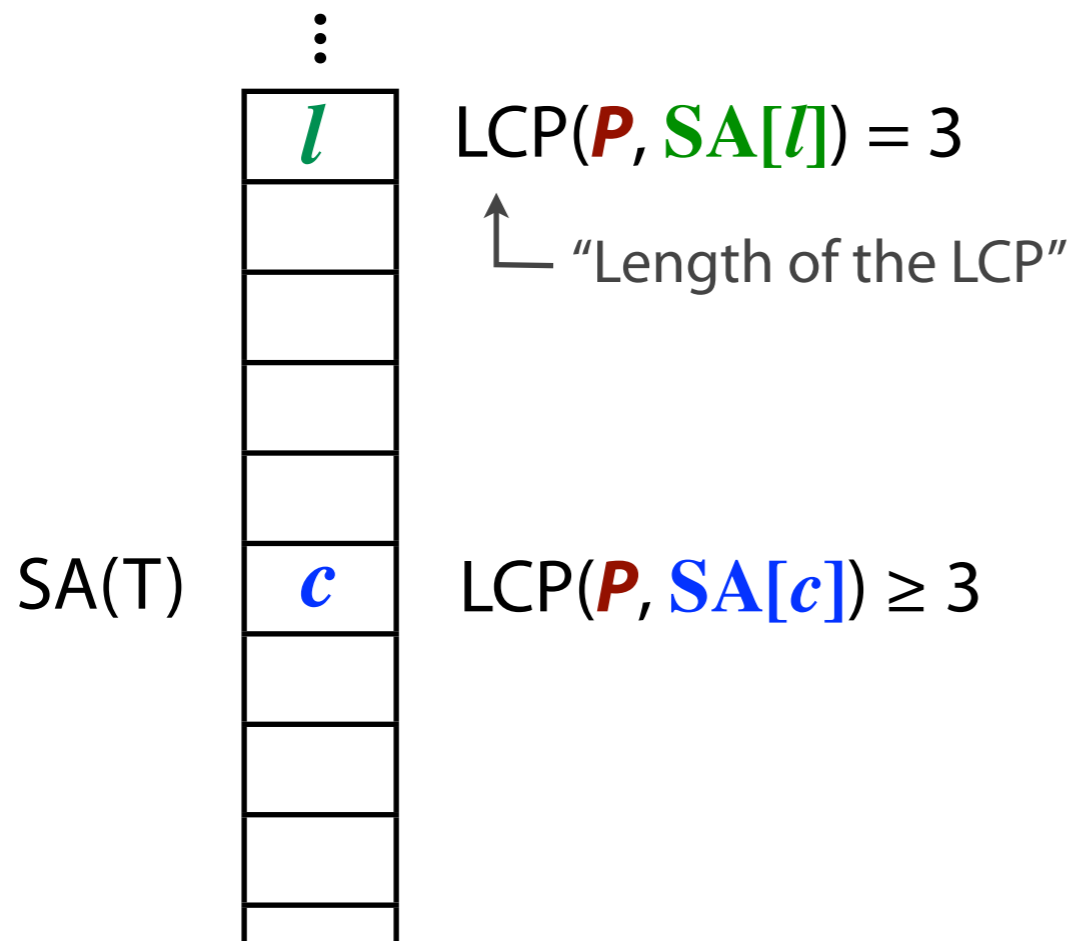
More generally:

$$LCP(P, SA[c]) \geq \min(LCP(P, SA[l]), LCP(P, SA[r]))$$

We can skip character comparisons

Suffix array: querying

Say we're comparing P to $SA[c]$ and we've already compared P to $SA[l]$ and $SA[r]$ in previous iterations.



More generally:

$$LCP(P, SA[c]) \geq \min(LCP(P, SA[l]), LCP(P, SA[r]))$$

We can skip character comparisons

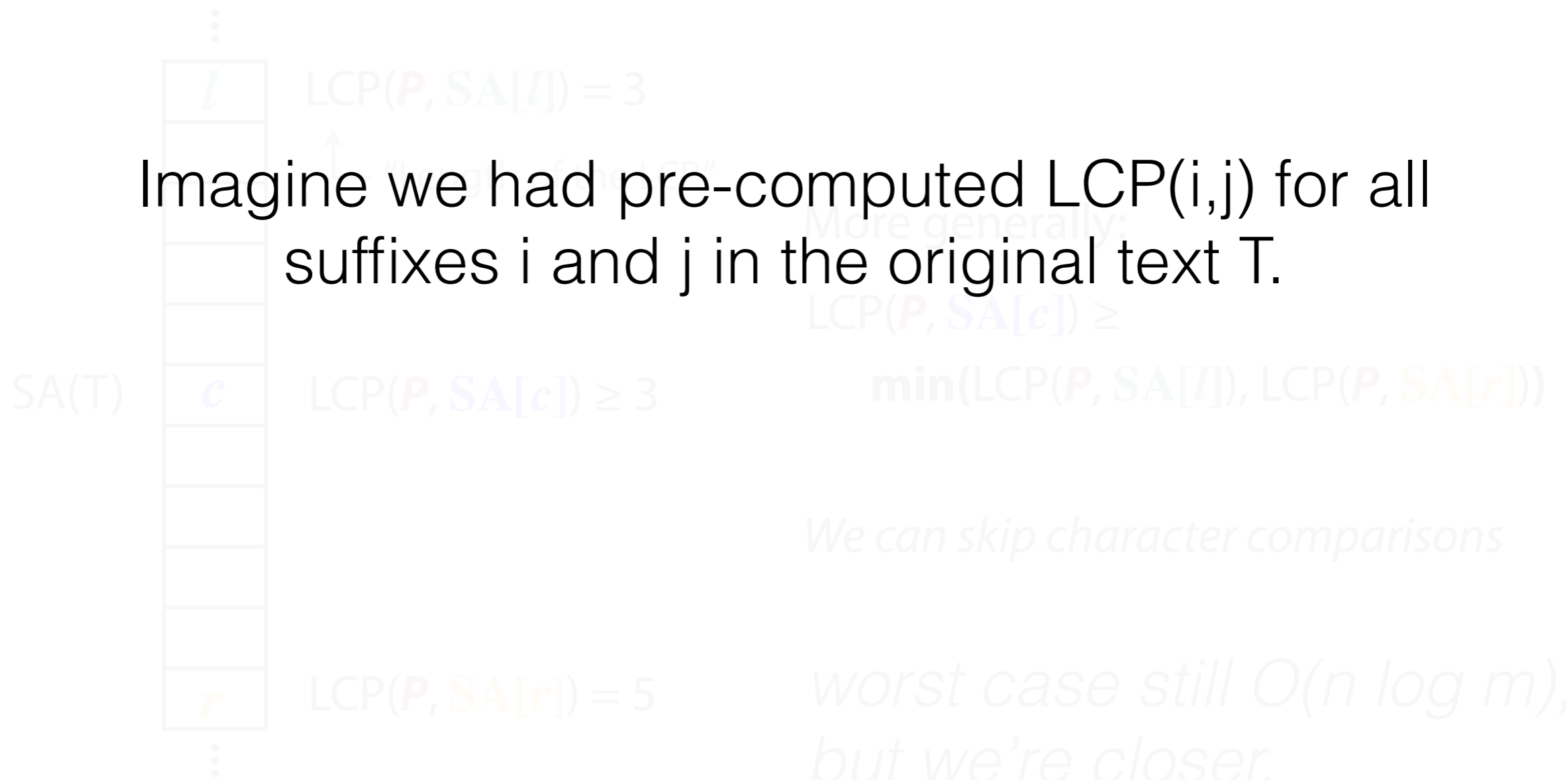
*worst case still $O(n \log m)$,
but we're closer.*

worst case example
 $S = ac_{M-2}b, P = c$

Suffix array: querying

Say we're comparing P to $SA[c]$ and we've already compared P to $SA[l]$ and $SA[r]$ in previous iterations.

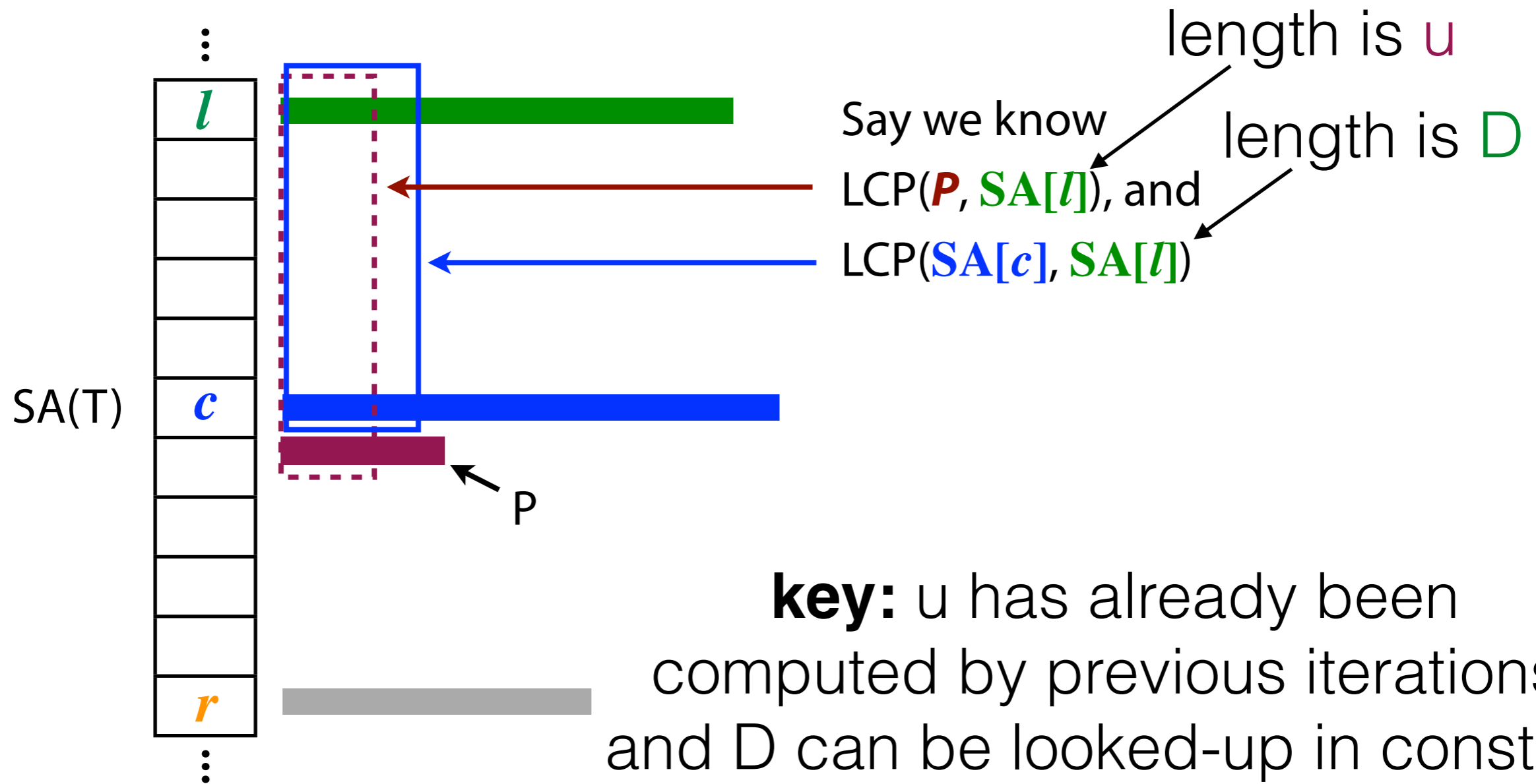
Imagine we had pre-computed $LCP(i,j)$ for all suffixes i and j in the original text T .



Suffix array: querying

Take an iteration of binary search:

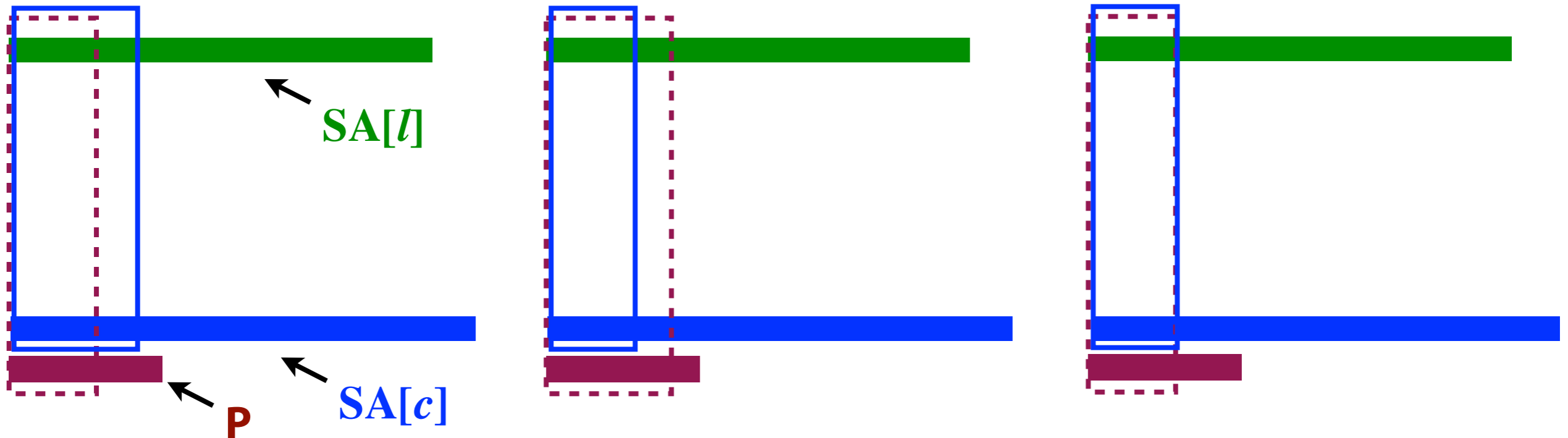
Assume, wlog, that $D = \text{LCP}(\text{SA}[l], \text{SA}[c]) \geq D' = \text{LCP}(\text{SA}[c], \text{SA}[r])$ otherwise there are symmetric cases.



key: u has already been computed by previous iterations, and D can be looked-up in constant time

Suffix array: querying

Three cases: or, if $D' = \text{LCP}(P, SA[r])$ is larger, 3 symmetric cases.



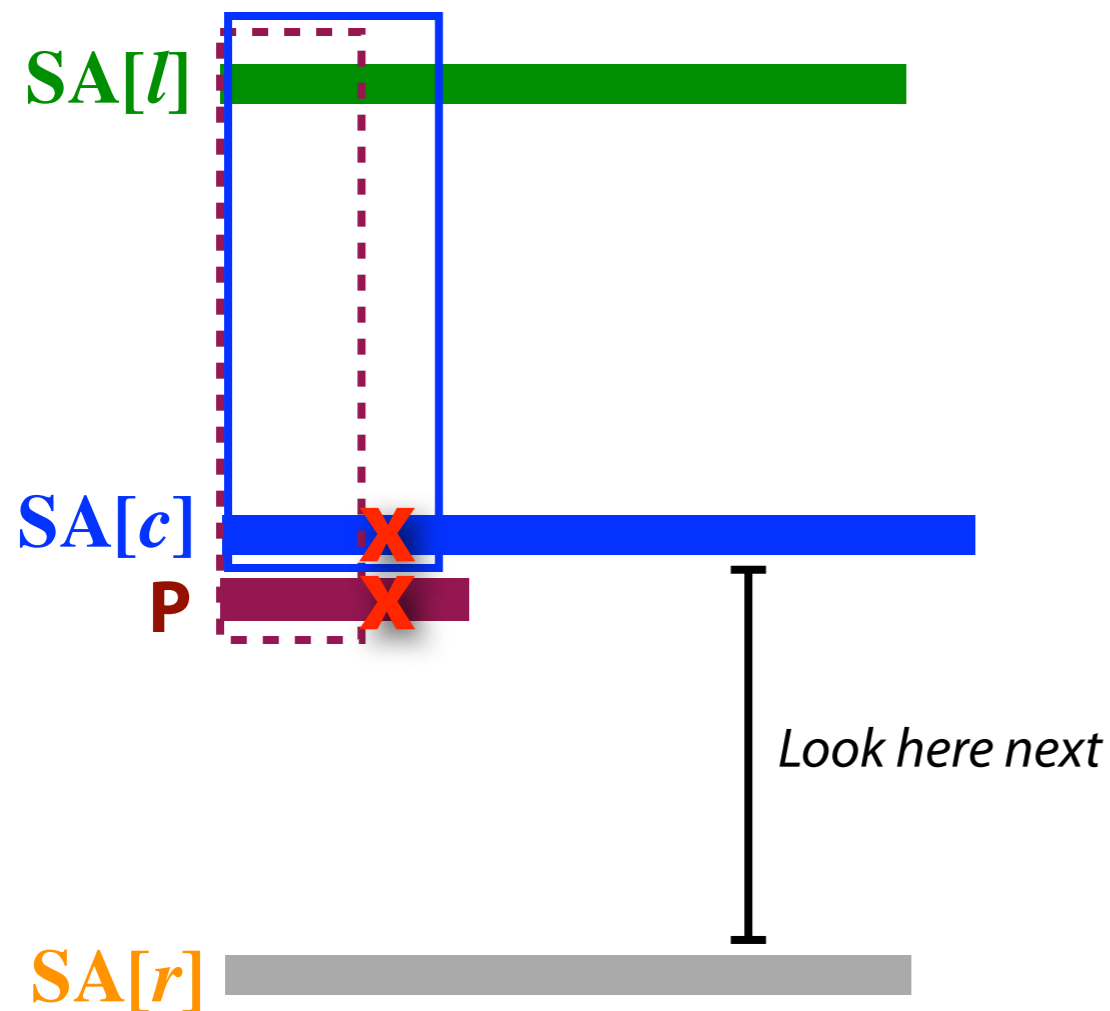
$$\text{LCP}(SA[c], SA[l]) > \text{LCP}(P, SA[l])$$

$$\text{LCP}(SA[c], SA[l]) < \text{LCP}(P, SA[l])$$

$$\text{LCP}(SA[c], SA[l]) = \text{LCP}(P, SA[l])$$

Suffix array: querying

Case 1: $\text{LCP}(\text{SA}[c], \text{SA}[l]) >$
 $\text{LCP}(P, \text{SA}[l])$



Next char of P after the $\text{LCP}(P, \text{SA}[l])$ must be *greater than* corresponding char of $\text{SA}[c]$

$$P > \text{SA}[c]$$

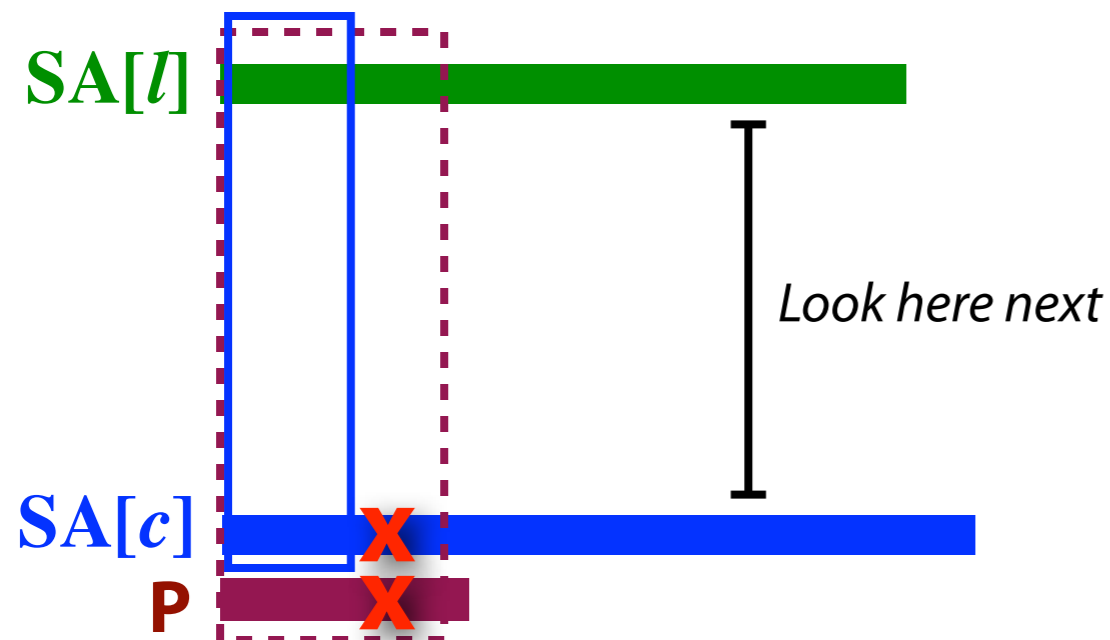
In this case, we compute $\text{LCP}(P[u:], \text{SA}[c][u:])$.
 c becomes our new l ,
and now we know the new $\text{LCP}(P, \text{SA}[l])$, b/c we just computed it!

$$\text{LCP}(\text{SA}[c], \text{SA}[l]) >$$

$$\text{LCP}(P, \text{SA}[l])$$

Suffix array: querying

Case 2: $\text{LCP}(\text{SA}[c], \text{SA}[l]) < \text{LCP}(P, \text{SA}[l])$



Next char of $\text{SA}[c]$ after $\text{LCP}(\text{SA}[c], \text{SA}[l])$ must be *greater than* corresponding char of P

$$P < \text{SA}[c]$$

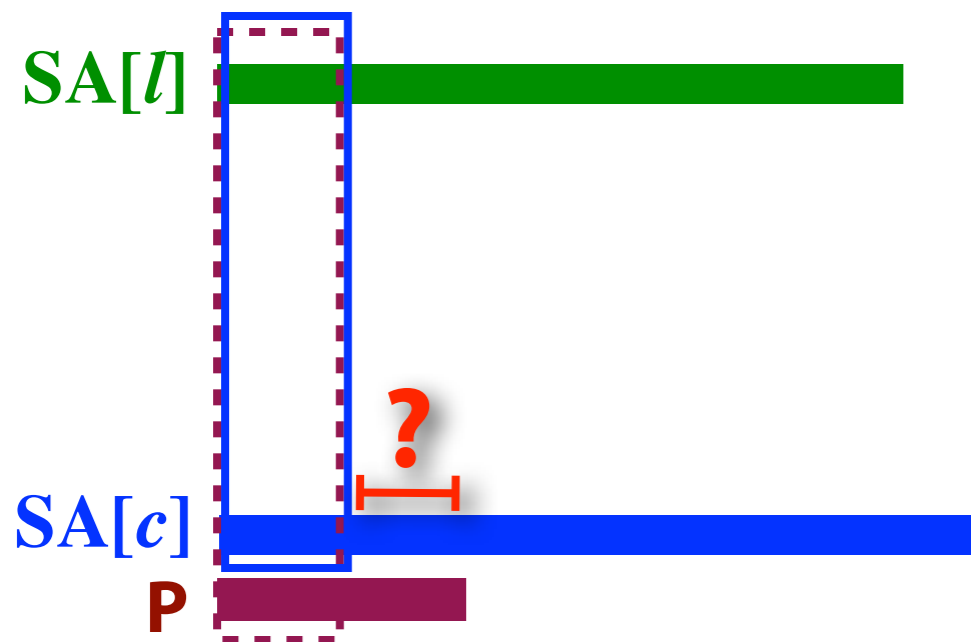
$\text{SA}[r]$

$\text{LCP}(\text{SA}[c], \text{SA}[l]) < \text{LCP}(P, \text{SA}[l])$

In this case, we compute $\text{LCP}(P[u:], \text{SA}[c][u:])$.
 c becomes our new r ,
 and now we know the new $\text{LCP}(P, \text{SA}[r])$, b/c we just computed it!

Suffix array: querying

Case 3: $\text{LCP}(\text{SA}[c], \text{SA}[l]) =$
 $\text{LCP}(\mathbf{P}, \text{SA}[l])$



Must do further character comparisons between \mathbf{P} and $\text{SA}[c]$

Each such comparison either:

- (a) mismatches, leading to a bisection
- (b) matches, in which case $\text{LCP}(\mathbf{P}, \text{SA}[c])$ grows

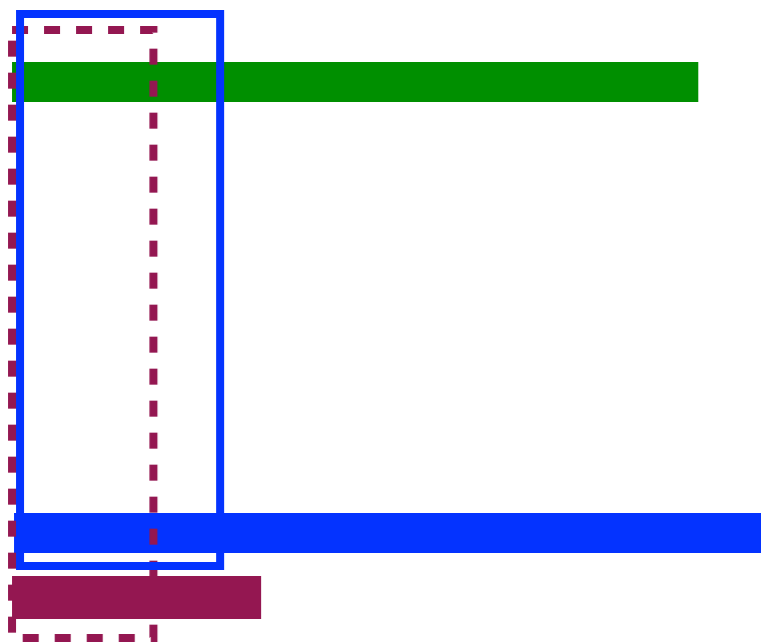
$\text{SA}[r]$ 

$$\text{LCP}(\text{SA}[c], \text{SA}[l]) =$$
$$\text{LCP}(\mathbf{P}, \text{SA}[l])$$

Suffix array: querying

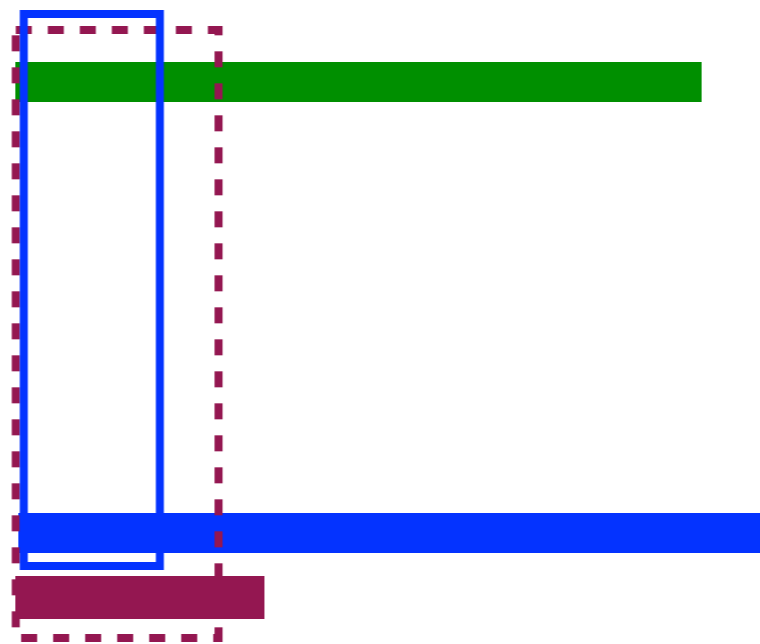
We improved binary search on suffix array from $O(n \log m)$ to $O(n + \log m)$ using information about Longest Common Prefixes (LCPs).

LCPs between P and suffixes of T computed during search, LCPs *among* suffixes of T computed *offline*



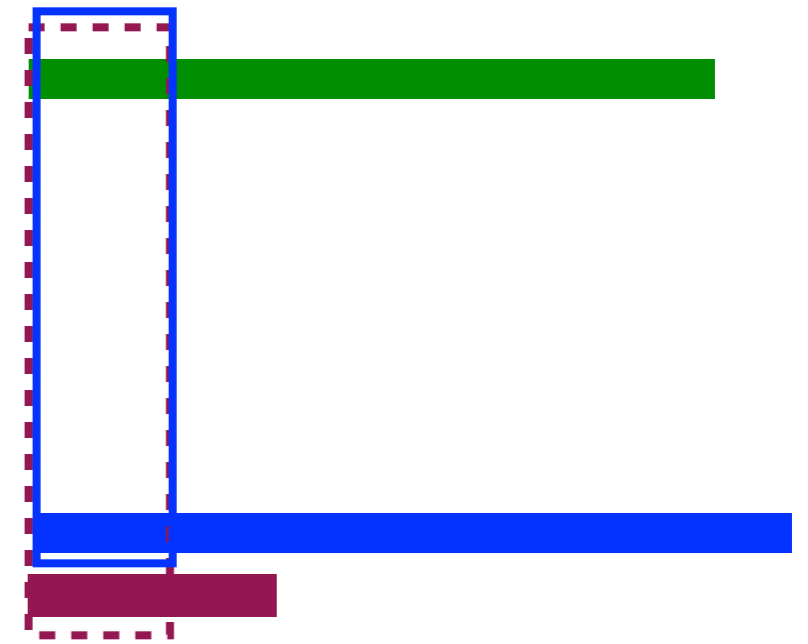
$$\text{LCP}(\text{SA}[c], \text{SA}[l]) > \\ \text{LCP}(P, \text{SA}[l])$$

Bisect right!



$$\text{LCP}(\text{SA}[c], \text{SA}[l]) < \\ \text{LCP}(P, \text{SA}[l])$$

Bisect left!



$$\text{LCP}(\text{SA}[c], \text{SA}[l]) = \\ \text{LCP}(P, \text{SA}[l])$$

Compare some
characters, then bisect!

Sketch of 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.

In case 1 & 2, we make $O(1)$ comparisons and bisect left or right — there are at most $O(\log m)$ bisections.

In case 3 we try to match characters starting at some offset between $SA[c]$ and P . If they match, those characters will never be compared again, so there are at most $O(|P|)$ such comparisons.

Mismatching characters may be compared more than once.

But there can be only 1 mismatch / bisection. There are $O(\log m)$ bisections, so there are at most $O(\log m)$ mismatches.

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



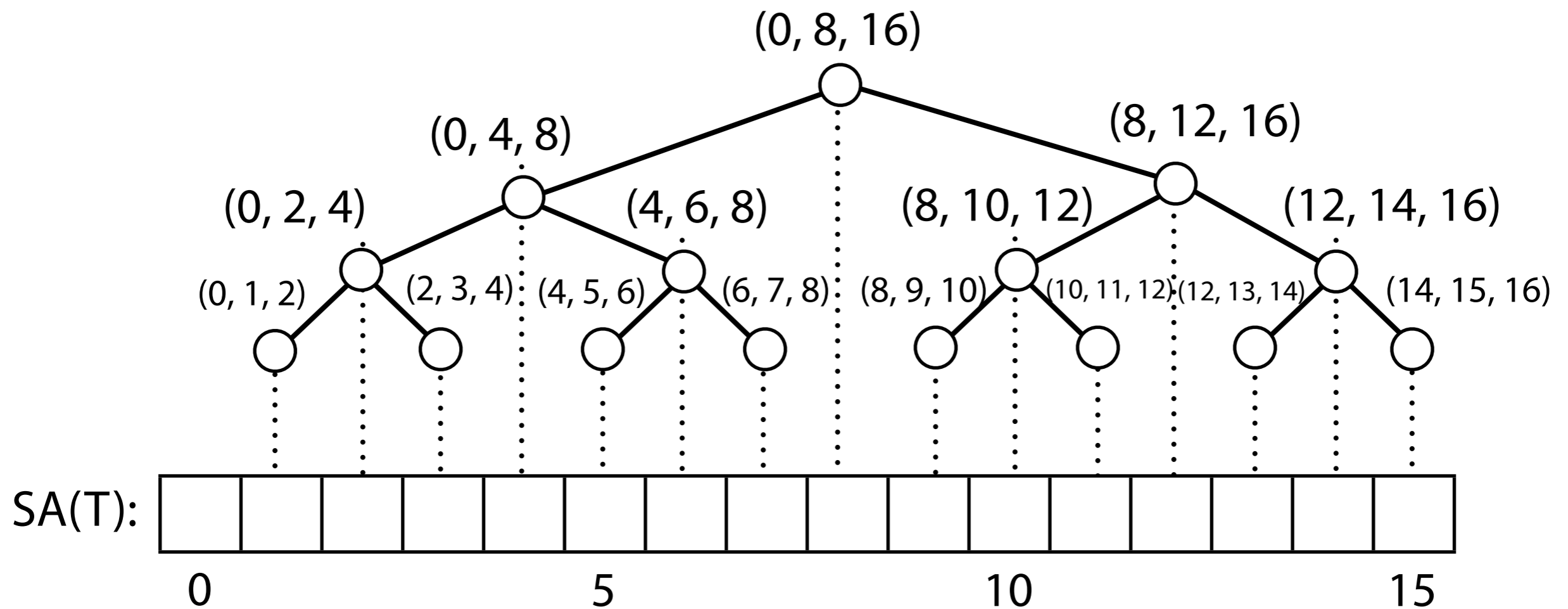
How to pre-compute LCP

- To perform this “efficient” search, we must be able to look up $LCP(SA[c], SA[l])$ and $LCP(SA[c], SA[r])$.
- How can we pre-compute this information *efficiently*?
 - Which LCP values do we need (*hint: not all of them*)?
 - Given LCP for left and right sub-interval of a search, how can we compute LCP for the containing interval?

Suffix array: LCPs

How to pre-calculate LCPs for every (l, c) and (c, r) pair in the search tree?

Triples are (l, c, r) triples



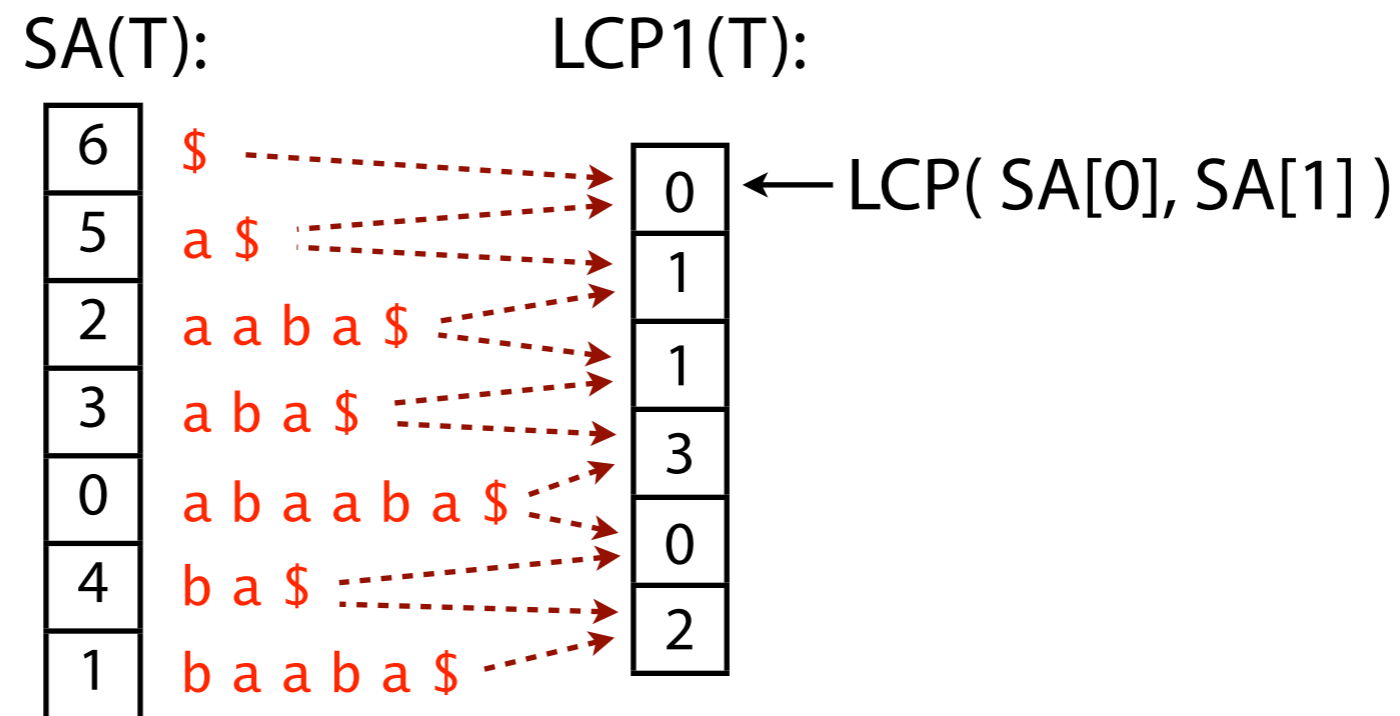
Example where $m = 16$ (incl. \$)

search tree nodes = $m - 1$

Suffix array: LCPs

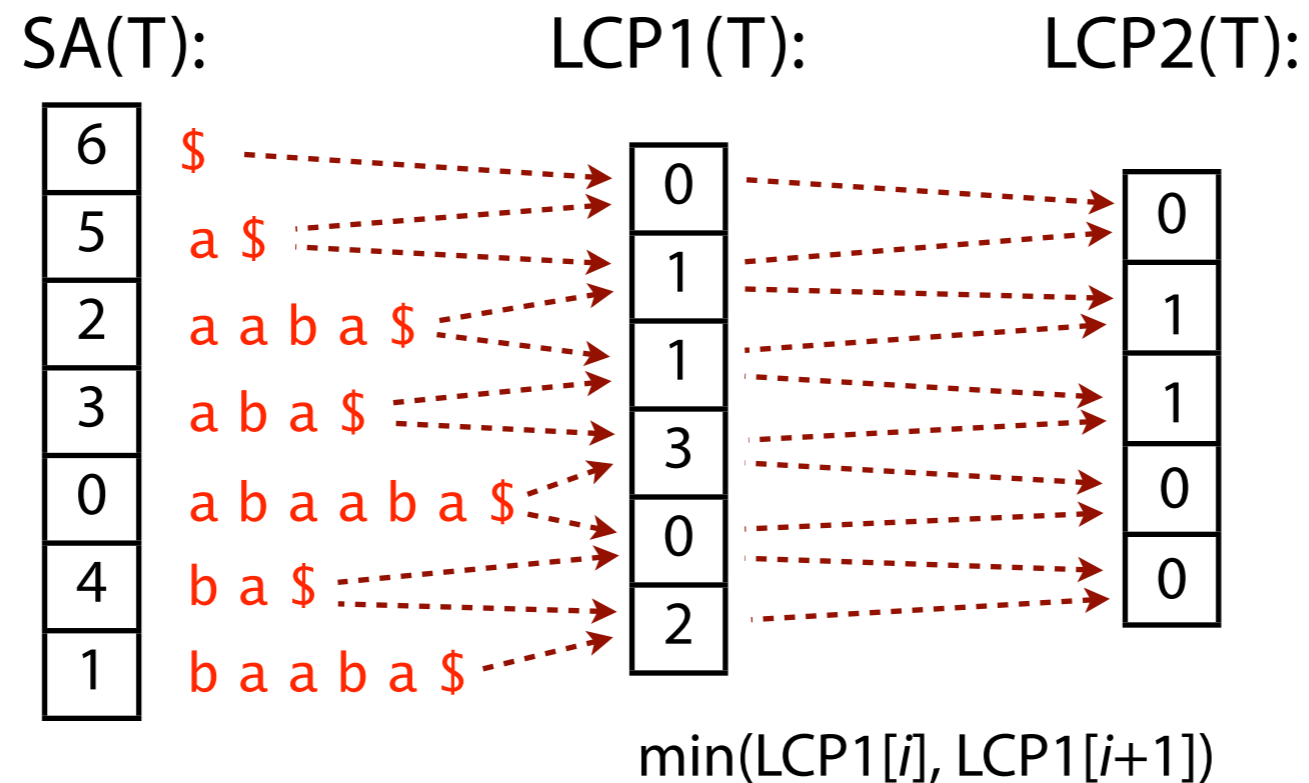
Suffix Array (SA) has m elements

Define LCP1 array with $m - 1$ elements such that $LCP[i] = LCP(SA[i], SA[i+1])$



Suffix array: LCPs

$$\text{LCP2}[i] = \text{LCP}(\text{SA}[i], \text{SA}[i+1], \text{SA}[i+2])$$

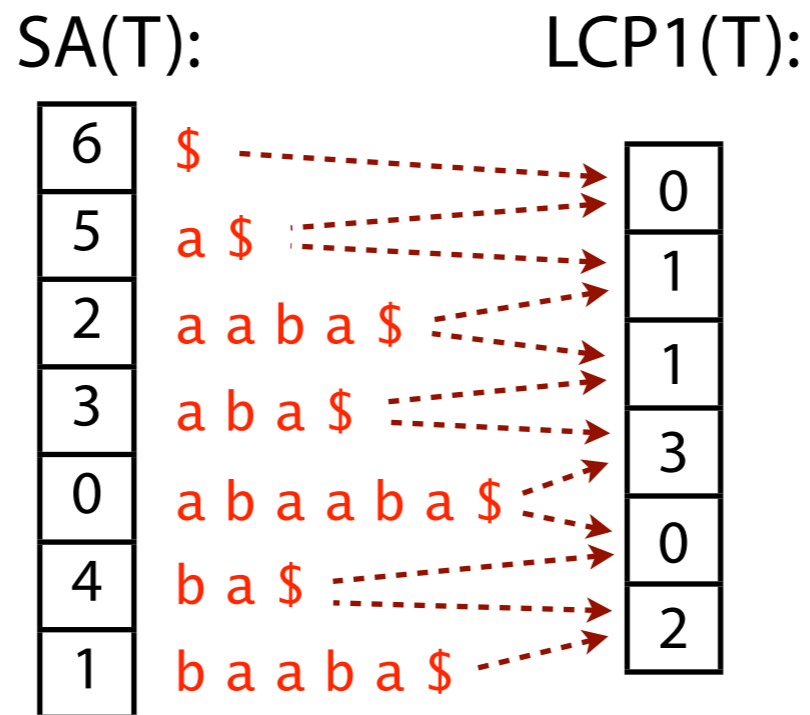


In fact, LCP of a range of consecutive suffixes in SA equals the minimum LCP1 among adjacent pairs in the range

LCP1 is a building block for other useful LCPs

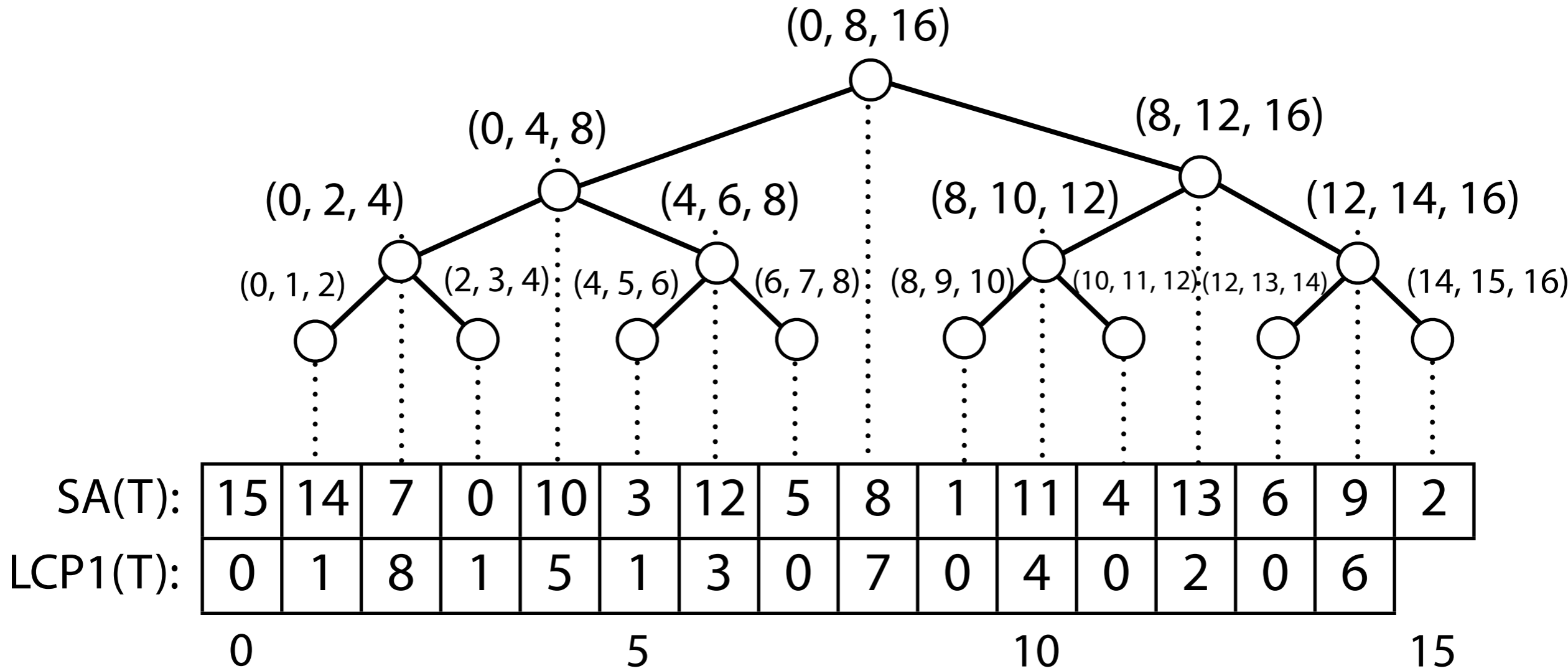
Suffix array: LCPs

Good time to calculate LCP1 it is *at the same time* as we *build* the suffix array, since putting the suffixes in order involves breaking ties after common prefixes



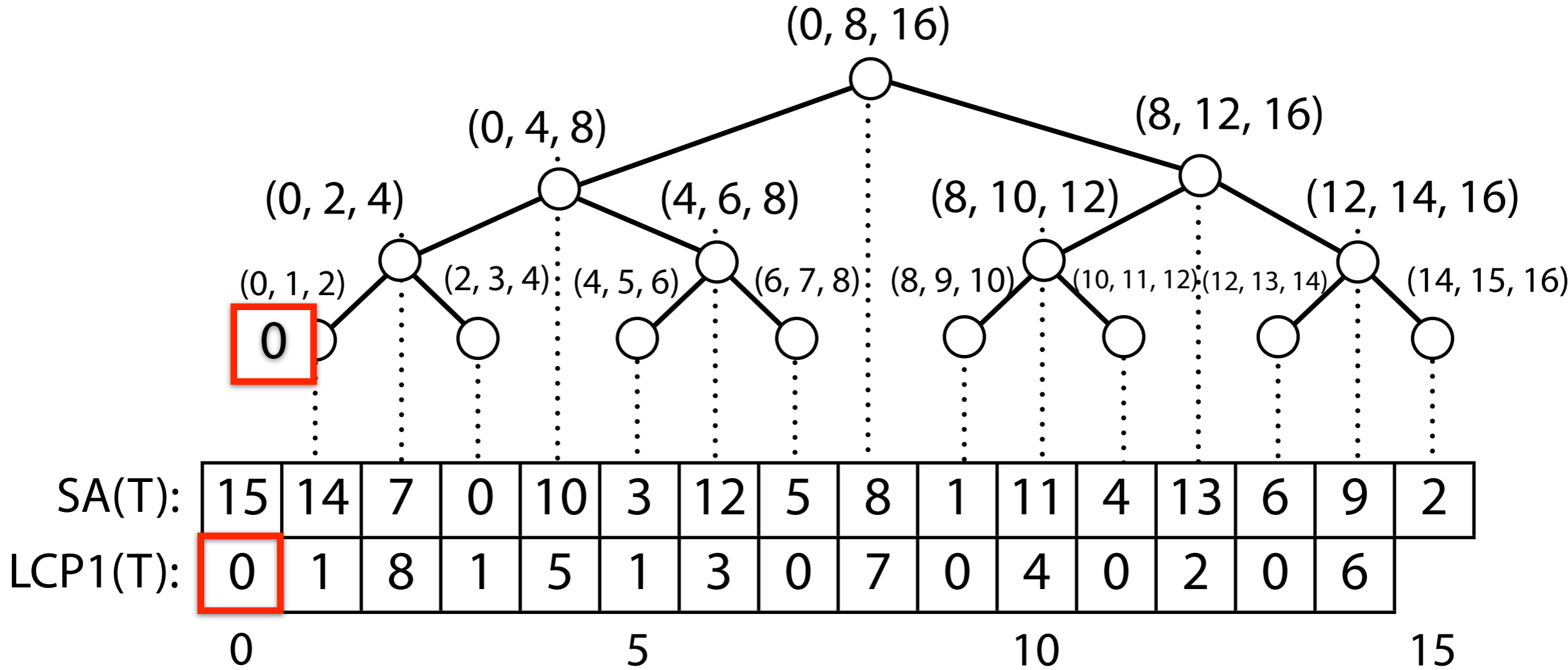
Suffix array: LCPs

T = abracadabracada



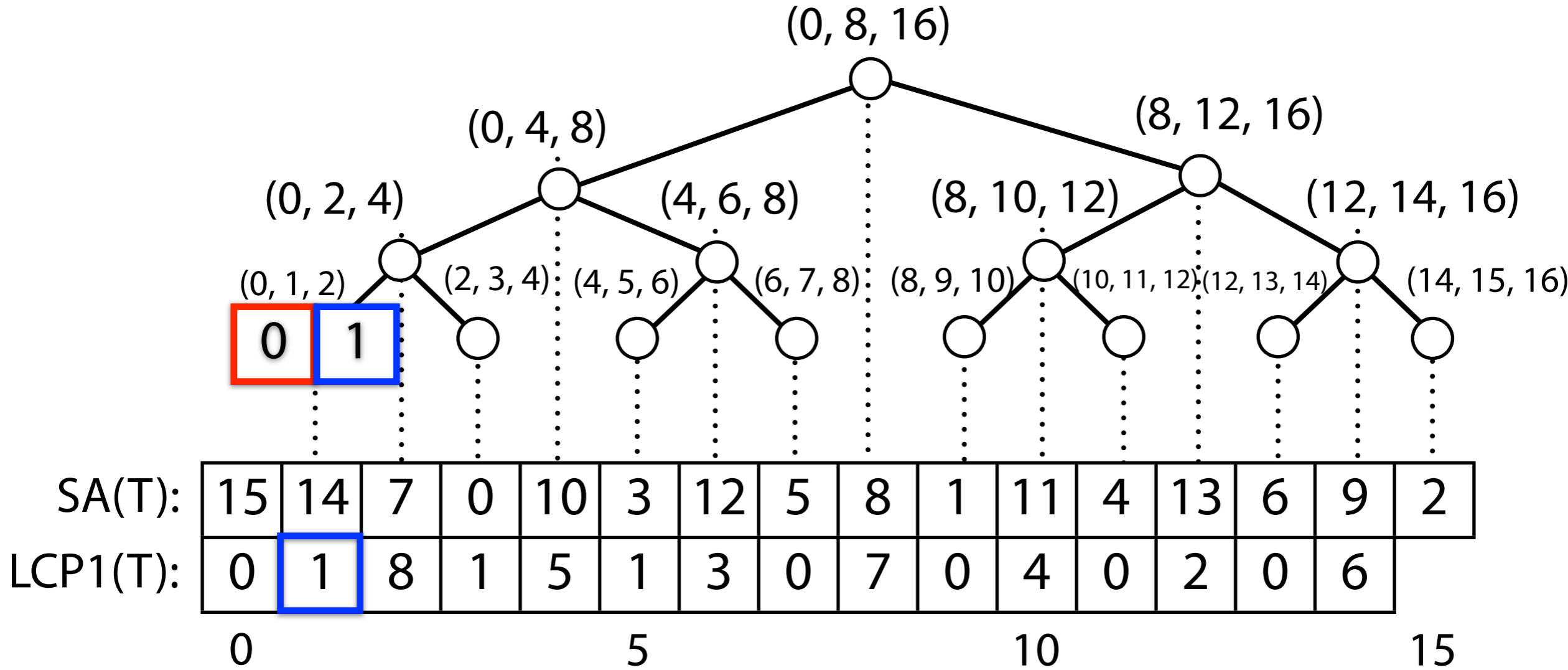
Suffix array: LCPs

T = abracadabracada\$



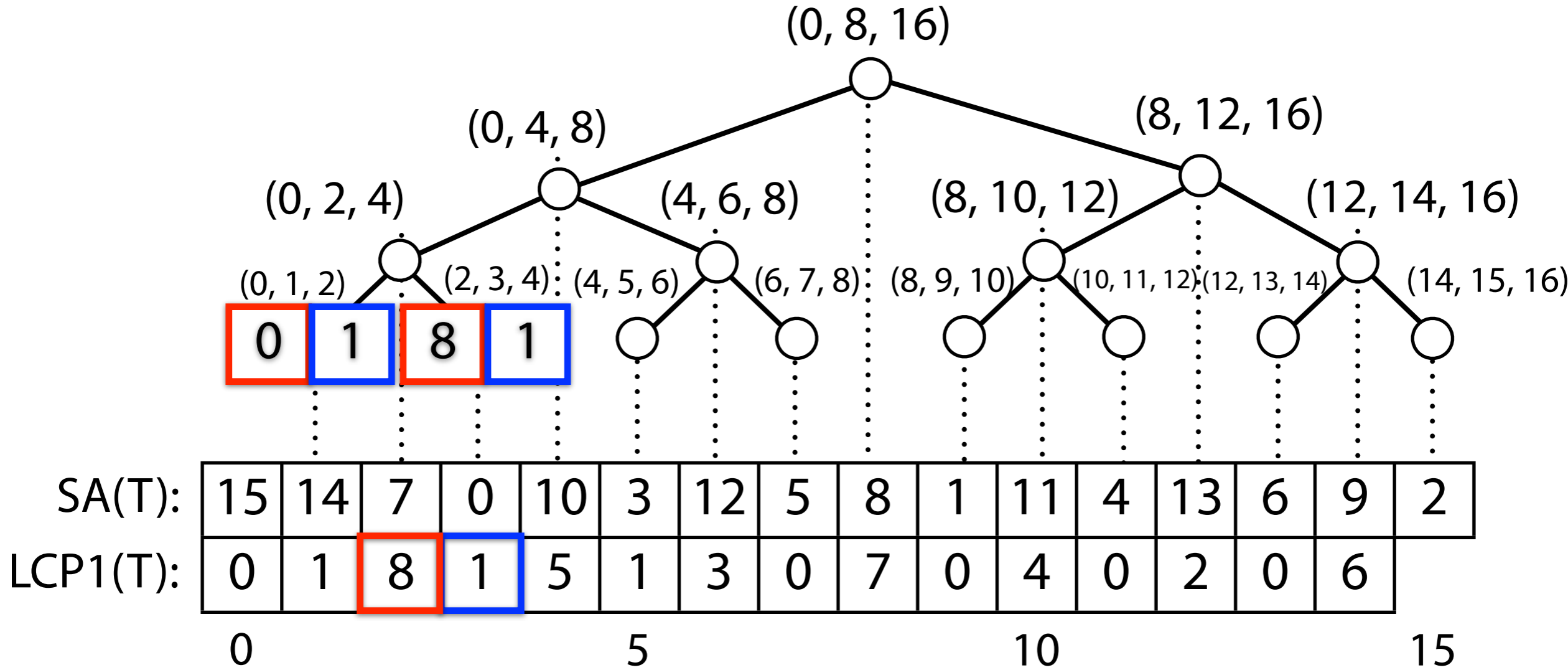
Suffix array: LCPs

T = abracadabracada\$



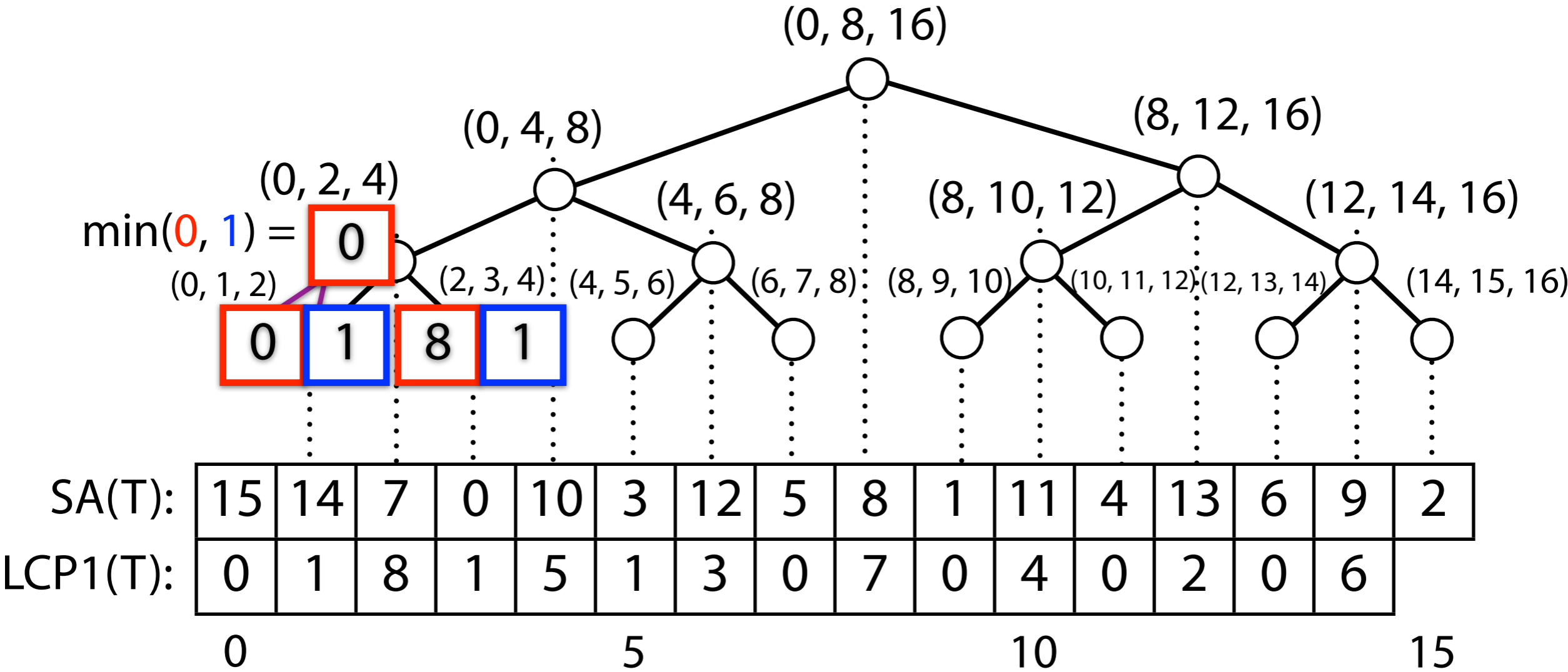
Suffix array: LCPs

T = abracadabracada\$



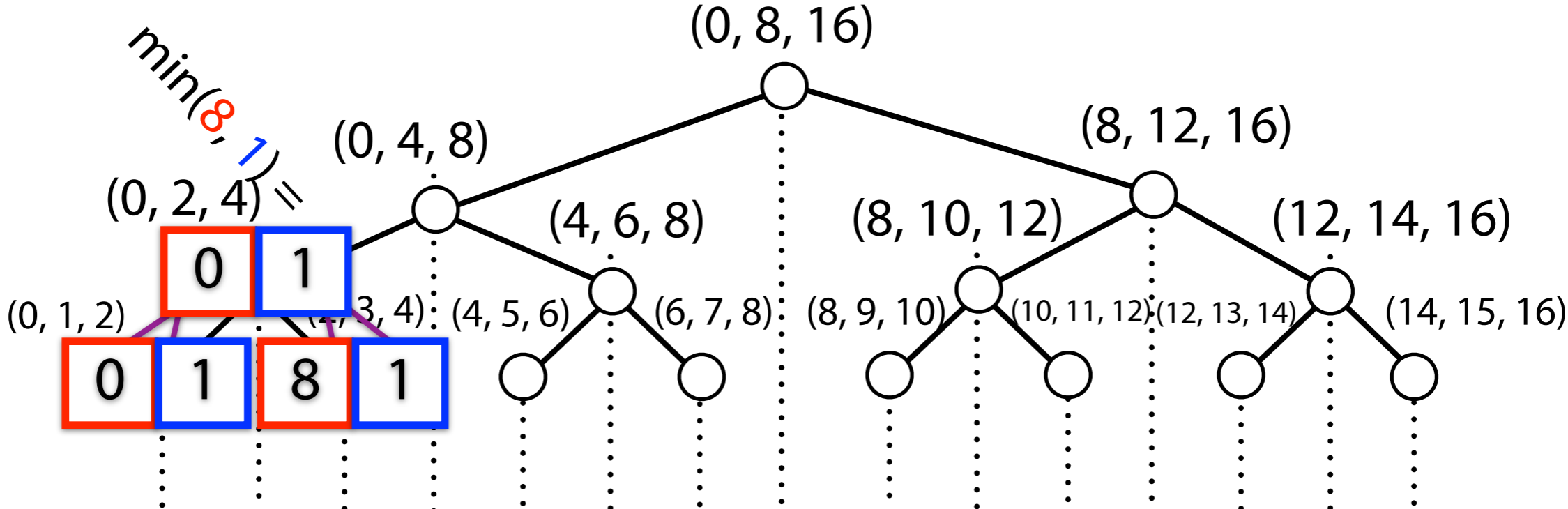
Suffix array: LCPs

T = abracadabracada\$



Suffix array: LCPs

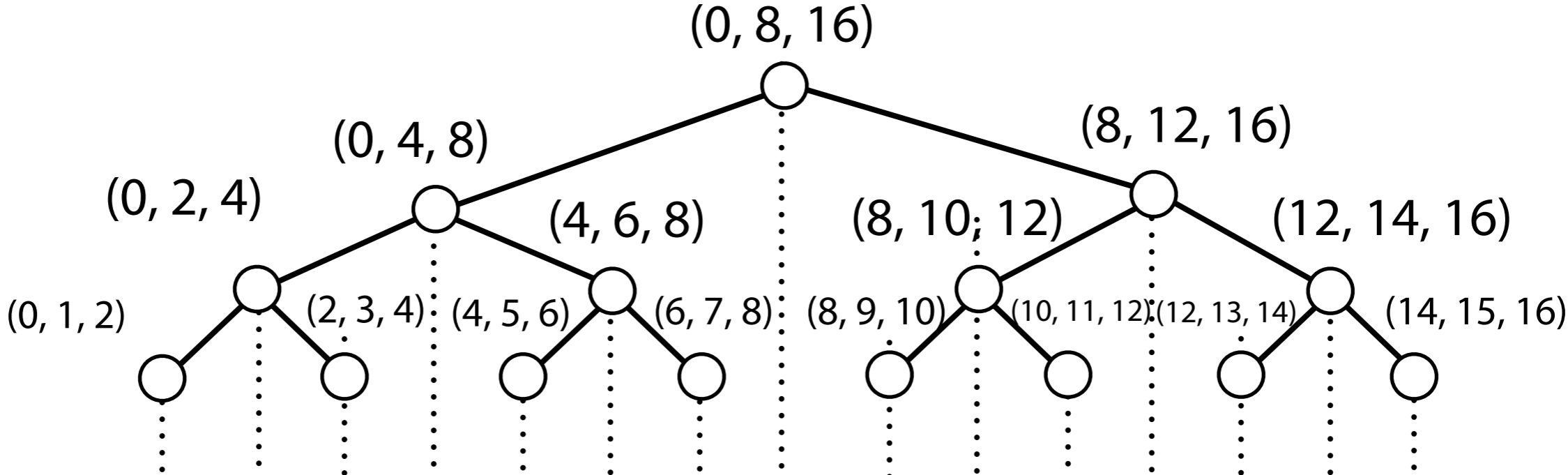
T = abracadabracada\$



SA(T):	15	14	7	0	10	3	12	5	8	1	11	4	13	6	9	2
LCP1(T):	0	1	8	1	5	1	3	0	7	0	4	0	2	0	6	
LCP_LC(T):	0	0	8													
LCP_CR(T):	1	1	1													
	0				5					10					15	

Suffix array: LCPs

T = abracadabracada\$



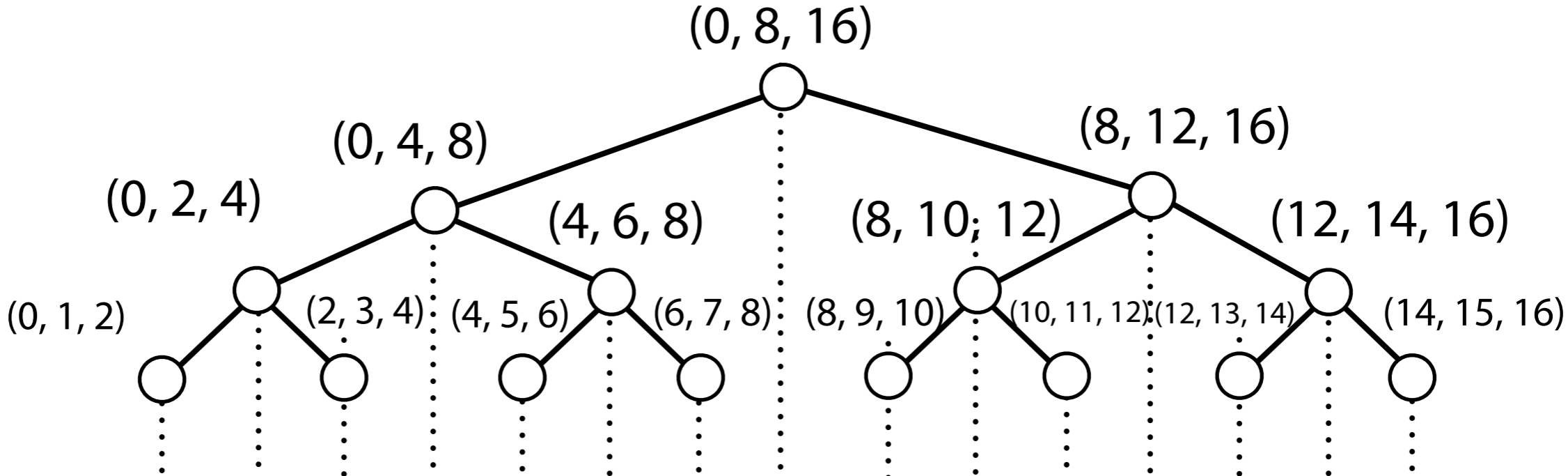
SA(T):	15	14	7	0	10	3	12	5	8	1	11	4	13	6	9	2
LCP1(T):	0	1	8	1	5	1	3	0	7	0	4	0	2	0	6	
LCP _{LC} (T):	0	0	8	0	5	1	3	0	7	0	4	0	2	0	6	
LCP _{CR} (T):	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	
	0				5				10						15	

NOTE: These arrays are “shifted” by 1 — the value in LCP_{LC} corresponding to (0, 1, 2) is at LCP_{LC}[0], not LCP_{LC}[1]. So, to look up LCP(SA[i], SA[c]) we look at LCP_{LC}[c-1]

Suffix array: LCPs

Can be done in:
 $O(m)$ time and space

T = abracadabracada\$



SA(T):	15	14	7	0	10	3	12	5	8	1	11	4	13	6	9	2
LCP1(T):	0	1	8	1	5	1	3	0	7	0	4	0	2	0	6	
LCP _{LC} (T):	0	0	8	0	5	1	3	0	7	0	4	0	2	0	6	
LCP _{CR} (T):	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	
	0				5				10						15	

NOTE: These arrays are “shifted” by 1 — the value in LCP_{LC} corresponding to (0, 1, 2) is at LCP_{LC}[0], not LCP_{LC}[1]. So, to look up LCP(SA[i], SA[c]) we look at LCP_{LC}[c-1]

Suffix array: querying review

We saw 3 ways to query (binary search) the suffix array:

1. Typical binary search. Ignores LCPs. $O(n \log m)$.
2. Binary search with some skipping using LCPs between P and T 's suffixes. Still $O(n \log m)$, but it can be argued it's near $O(n + \log m)$ in practice. Gusfield: "Simple Accelerant"
3. Binary search with skipping using all LCPs, including LCPs among T 's suffixes. $O(n + \log m)$. Gusfield: "Super Accelerant"

How much space do they require?

1. $\sim m$ integers (SA)
2. $\sim m$ integers (SA)
3. $\sim 3m$ integers (SA, LCP_LC, LCP_CR)

Suffix array: performance comparison

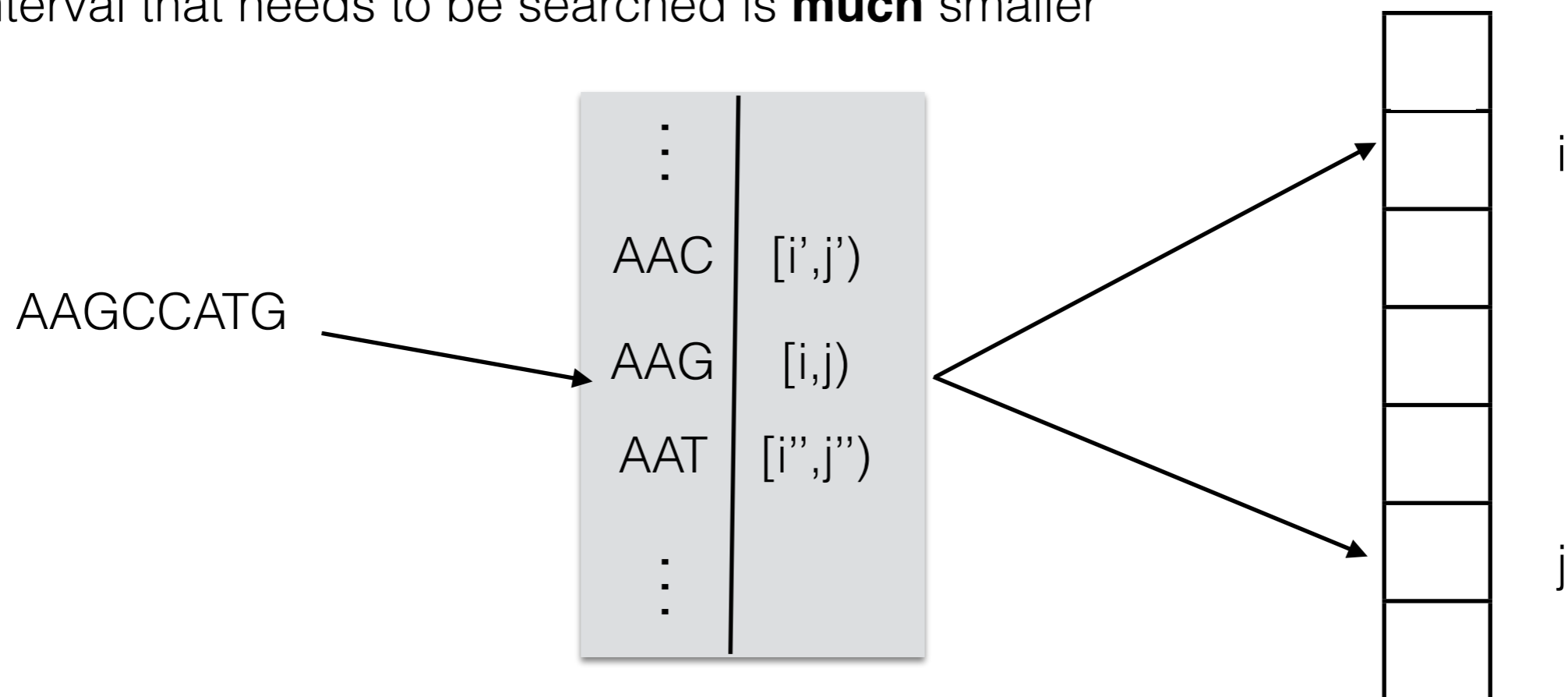
	Super accelerant	Simple accelerant	No accelerant
python -O	68.78 s	69.80 s	102.71 s
pypy -O	5.37 s	5.21 s	8.74 s
# character comparisons	99.5 M	117 M	235 M

Matching 500K 100-nt substrings to the ~ 5 million nt-long *E. coli* genome. Substrings drawn randomly from the genome.

Index building time not included

Another “practical” speedup

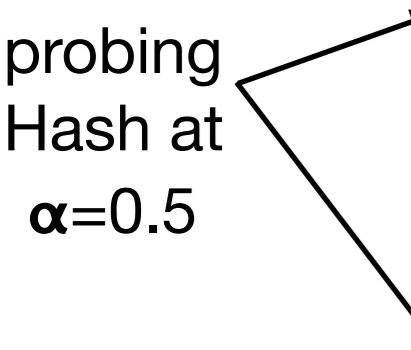
- Imagine you will never search for patterns of length $< k$ (e.g. 4-mers are non-informative in any moderately-sized genome)
- Consider the following “enhanced” suffix array:
 - Build a hash-table from k -mers to suffix array intervals. Now, any pattern of length $k' > k$ must start with some hashed prefix of length k . Generally, the interval that needs to be searched is **much** smaller



Now, you only need to search the interval $[i,j)$ — $O(n * \log(j-i))$ time

Can provide considerable speedup

k=12
Linear probing
Hash at $\alpha=0.5$



	dna	english	proteins	sources	xml
<i>m</i> = 16					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.13	1.34	1.36	1.43	1.35
SA-LUT3	1.17	1.49	1.61	1.65	1.47
SA-hash	3.75	2.88	2.70	2.90	2.03
<i>m</i> = 64					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.12	1.33	1.34	1.42	1.34
SA-LUT3	1.17	1.49	1.58	1.64	1.44
SA-hash	3.81	2.87	2.62	2.75	1.79

Table 1. Speedups with regard to the search speed of the plain suffix array, for the five datasets and pattern lengths $m = 16$ and $m = 64$

Some other clever ideas#:

- Use a k-ary (B-tree) layout
- Use a lookup table where keys are concatenated Huffman codes of fixed bit length
- Use alternative strategy (doubling/galloping) to find the right SA boundary

“Two Simple Full-Text Indexes Based on the Suffix Array”, Szymon Grabowski and Marcin Raniszewski

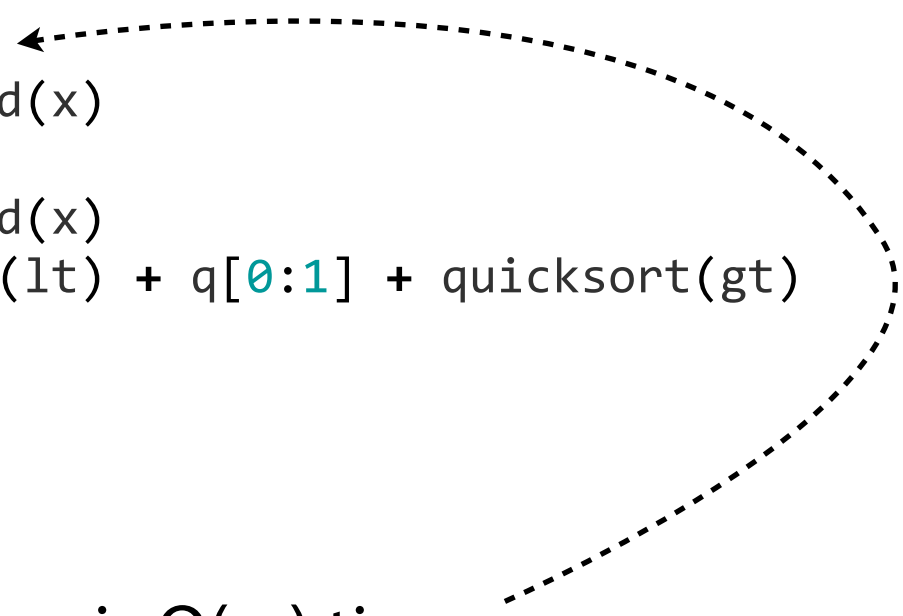
Kowalski, Tomasz, et al. "Suffix arrays with a twist." *arXiv preprint arXiv:1607.08176* (2016).

Suffix array: sorting suffixes

One idea: Use your favorite sort, e.g., quicksort

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

```
def quicksort(q):  
    lt, gt = [], []  
    if len(q) <= 1:  
        return q  
    for x in q[1:]:  
        if x < q[0]:  
            lt.append(x)  
        else:  
            gt.append(x)  
    return quicksort(lt) + q[0:1] + quicksort(gt)
```



Expected time: $O(m^2 \log m)$

Not $O(m \log m)$ because a suffix comparison is $O(m)$ time

Suffix array: sorting suffixes

One idea: Use a sort algorithm that's aware that the items being sorted are strings, e.g. "multikey quicksort"

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

Essentially $O(m^2)$ time

Bentley, Jon L., and Robert Sedgwick. "Fast algorithms for sorting and searching strings." *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1997

Suffix array: sorting suffixes

Another idea: Use a sort algorithm that's aware that the items being sorted are all suffixes of the same string

Original suffix array paper suggested an $O(m \log m)$ algorithm

Manber U, Myers G. "Suffix arrays: a new method for on-line string searches." SIAM Journal on Computing 22.5 (1993): 935-948.

Other popular $O(m \log m)$ algorithms have been suggested

Larsson NJ, Sadakane K. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-43/(1999), Department of Computer Science, Lund University, Sweden, 1999.

More recently $O(m)$ algorithms have been demonstrated!

Kärkkäinen J, Sanders P. "Simple linear work suffix array construction." Automata, Languages and Programming (2003): 187-187.

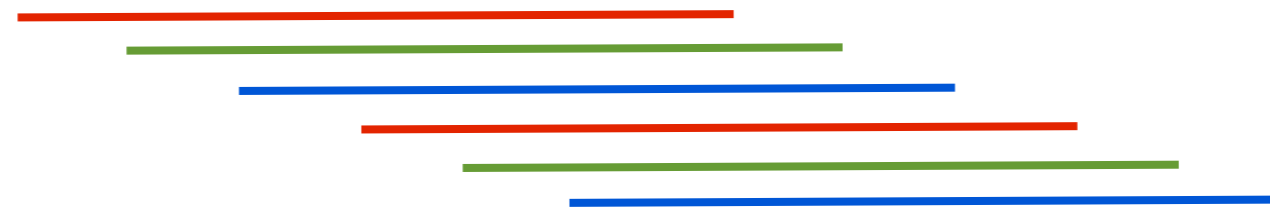
Ko P, Aluru S. "Space efficient linear time construction of suffix arrays." Combinatorial Pattern Matching. Springer Berlin Heidelberg, 2003.

The Skew Algorithm (aka DC3)

Kärkkäinen & Sanders, 2003

- **Main idea: Divide suffixes into 3 groups:**
 - Those starting at positions $i=0,3,6,9,\dots$ ($i \bmod 3 = 0$)
 - Those starting at positions $1,4,7,10,\dots$ ($i \bmod 3 = 1$)
 - Those starting at positions $2,5,8,11,\dots$ ($i \bmod 3 = 2$)
- For simplicity, assume text length is a multiple of 3 after padding with a special character.

$T[0, n) =$ 0 1 2 3 4 5 6 7 8 9 10 11 12
y a b b a d a b b a d o o



$SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0)$

Basic Outline:

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

Step 0 — Constructing a sample

These are called the “sample suffixes”

Step 0: Construct a sample. For $k = 0, 1, 2$, define

$$B_k = \{i \in [0, n] \mid i \bmod 3 = k\}.$$

Let $C = B_1 \cup B_2$ be the set of *sample positions* and S_C the set of *sample suffixes*.

Example. $B_1 = \{1, 4, 7, 10\}$, $B_2 = \{2, 5, 8, 11\}$, i.e., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Step 1 — Sorting the sample

Step 1: Sort sample suffixes. For $k = 1, 2$, construct the strings

$$R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \cdots [t_{\max B_k} t_{\max B_k + 1} t_{\max B_k + 2}]$$

whose characters are triples $[t_i t_{i+1} t_{i+2}]$. Note that the last character of R_k is always unique because $t_{\max B_k + 2} = 0$. Let $R = R_1 \odot R_2$ be the concatenation of R_1 and R_2 . Then the (nonempty) suffixes of R correspond to the set S_C of sample suffixes: $[t_i t_{i+1} t_{i+2}] [t_{i+3} t_{i+4} t_{i+5}] \cdots$ corresponds to S_i . The correspondence is order preserving, i.e., by sorting the suffixes of R we get the order of the sample suffixes S_C .

Example. $R = [\text{abb}][\text{ada}][\text{bba}][\text{do0}][\text{bba}][\text{dab}][\text{bad}][\text{o00}]$.

Step 1 — Sorting the sample

To sort the suffixes of R , first radix sort the characters of R and rename them with their ranks to obtain the string R' . If all characters are different, the order of characters gives directly the order of suffixes. Otherwise, sort the suffixes of R' using Algorithm DC3.

Example. $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ and $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

Step 1.5 — Sorting the sample

Example. $R = [\text{abb}][\text{ada}][\text{bba}][\text{do0}][\text{bba}][\text{dab}][\text{bad}][\text{o00}]$.

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $\text{rank}(S_i)$ denote the rank of S_i in the sample set S_C . Additionally, define $\text{rank}(S_{n+1}) = \text{rank}(S_{n+2}) = 0$. For $i \in B_0$, $\text{rank}(S_i)$ is undefined.

Example.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\text{rank}(S_i)$	\perp	1	4	\perp	2	6	\perp	5	3	\perp	7	8	\perp	0	0

Step 1.5 — Sorting the sample

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C . Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

Example. $rank(S_i)$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	\perp	1	4	\perp	2	6	\perp	5	3	\perp	7	8	\perp	0	0

$T[0, n) =$

i	0	1	2	3	4	5	6	7	8	9	10	11
	y	a	b	b	a	d	a	b	b	a	d	o

Example. $R = [abb][ada][bba][do0][bba][dab][bad][o00]$.

Example. $rank(S_i)$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	\perp	1	4	\perp	2	6	\perp	5	3	\perp	7	8	\perp	0	0

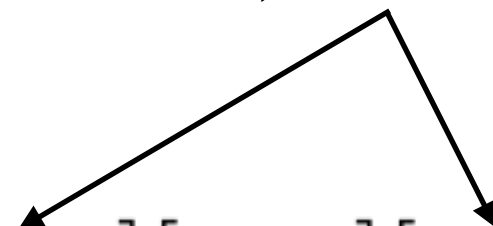
Step 1.5 — Sorting the sample

Once the sample suffixes are sorted, assign a rank to each suffix. For $i \in C$, let $rank(S_i)$ denote the rank of S_i in the sample set S_C . Additionally, define $rank(S_{n+1}) = rank(S_{n+2}) = 0$. For $i \in B_0$, $rank(S_i)$ is undefined.

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Example.	$rank(S_i)$	\perp	1	4	\perp	2	6	\perp	5	3	\perp	7	8	\perp	0	0

Note: After only 1 level of recursion, these suffixes would be “tied”

Example. $R = [abb][ada][bba][do0][bba][dab][bad][o00]$.



The resolved ranks here represent what we'd get after a second level of recursion.

Step 1.5 — Sorting the sample

$T[0, n) =$

0	1	2	3	4	5	6	7	8	9	10	11
y	a	b	b	a	d	a	b	b	a	d	o

Example. $R = [abb][ada][bba][do0][bba][dab][bad][o00]$.

1 2 4 7 4 6 3 8

$R_2 = [247][463][474][638]$

These suffixes were tied at the previous level, but here, we can resolve them. The *lexical renaming* allows us to compare longer and longer suffixes of the text.

Step 2 — Sorting the non-sample suffixes

Step 2: Sort nonsample suffixes. Represent each nonsample suffix $S_i \in S_{B_0}$ with the pair $(t_i, \text{rank}(S_{i+1}))$. Note that $\text{rank}(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

$$S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})).$$

The pairs $(t_i, \text{rank}(S_{i+1}))$ are then radix sorted.

Example. $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(0, 0) < (\mathbf{a}, 5) < (\mathbf{a}, 7) < (\mathbf{b}, 2) < (\mathbf{y}, 1)$.

Step 2 — Sorting the non-sample suffixes

Step 3: Merge. The two sorted sets of suffixes are merged using a standard comparison-based merging. To compare suffix $S_i \in S_C$ with $S_j \in S_{B_0}$, we distinguish two cases:

$$\begin{aligned} i \in B_1 : \quad S_i \leq S_j &\iff (t_i, \overset{S_{B1}}{\downarrow} \text{rank}(\overset{S_{B2}}{\downarrow} S_{i+1})) \leq (t_j, \overset{S_{B0}}{\downarrow} \text{rank}(\overset{S_{B1}}{\downarrow} S_{j+1})) \\ i \in B_2 : \quad S_i \leq S_j &\iff (t_i, \overset{S_{B2}}{\uparrow} t_{i+1}, \overset{S_{B0}}{\uparrow} \text{rank}(\overset{S_{B1}}{\uparrow} S_{i+2})) \leq (t_j, \overset{S_{B0}}{\uparrow} t_{j+1}, \overset{S_{B1}}{\uparrow} \text{rank}(\overset{S_{B2}}{\uparrow} S_{j+2})) \end{aligned}$$

Note that the ranks are defined in all cases.

Example. $S_1 < S_6$ because $(\mathbf{a}, 4) < (\mathbf{a}, 5)$ and $S_3 < S_8$ because $(\mathbf{b}, \mathbf{a}, 6) < (\mathbf{b}, \mathbf{a}, 7)$.

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: $T(n) \leq cn + T(2n/3)$ for some c .
- Guess: $T(n) \leq 3cn$
- Induction step: assume that is true for all $i < n$.
- $T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn \quad \square$

Using the suffix array for read alignment: STAR

BIOINFORMATICS ORIGINAL PAPER

Vol. 29 no. 1 2013, pages 15–21
doi:10.1093/bioinformatics/bts635

Sequence analysis

Advance Access publication October 25, 2012

STAR: ultrafast universal RNA-seq aligner

Alexander Dobin^{1,*}, Carrie A. Davis¹, Felix Schlesinger¹, Jorg Drenkow¹, Chris Zaleski¹, Sonali Jha¹, Philippe Batut¹, Mark Chaisson² and Thomas R. Gingeras¹

¹Cold Spring Harbor Laboratory, Cold Spring Harbor, NY, USA and ²Pacific Biosciences, Menlo Park, CA, USA

Associate Editor: Inanc Birol

Seeding through SA search for MMPs

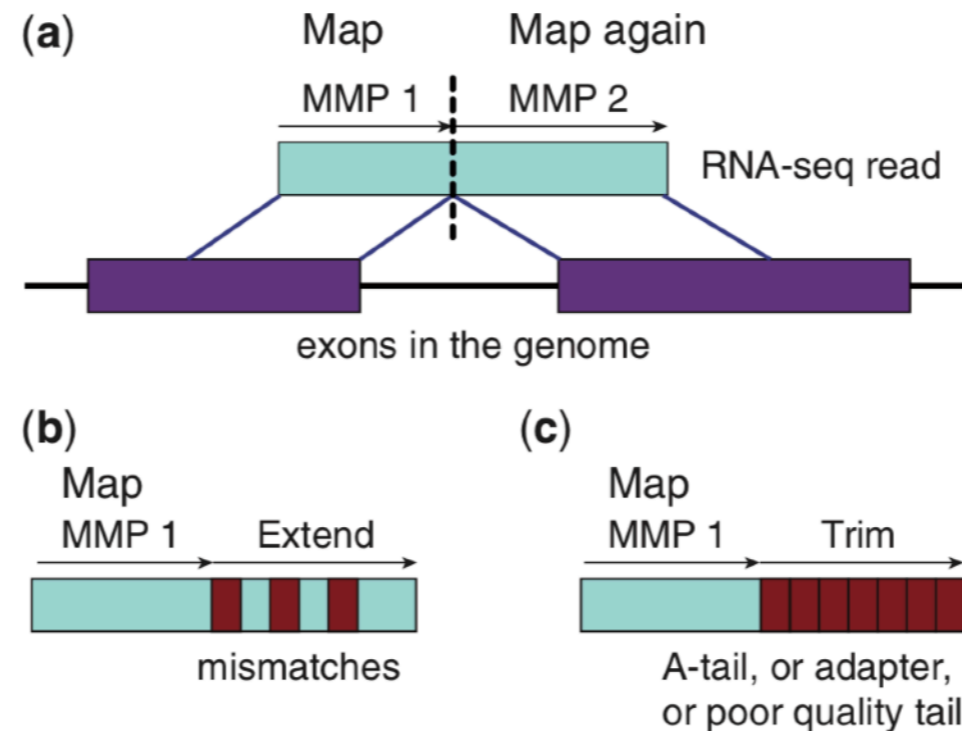


Fig. 1. Schematic representation of the Maximum Mappable Prefix search in the STAR algorithm for detecting (a) splice junctions, (b) mismatches and (c) tails

With read sequence R , read location i , and reference sequence G , the $MMP(R,i,G)$ is defined as the longest substring $(R_i, R_{i+1}, \dots, R_{i+MML-1})$ where MML is the maximum mappable length.

MMPs are computed starting at 5' end, but also at regular intervals in the read. The read is also searched in the 3'→5' direction.

Question: How do you search for an MMP in the suffix array?

Seeding through SA search for MMPs

To speed up suffix array search even further, STAR takes advantage of the heuristic we discussed above:

1.2. Pre-indexing of suffix arrays

While suffix array search is theoretically fast owing to its binary nature, in practice it may suffer from non-locality resulting in persistent cache misses which deteriorate the performance. To alleviate this problem we developed a pre-indexing strategy. After the SA is generated, we find the locations of all possible L-mers in the SA, $L \leq L_{\max}$, where L_{\max} is user defined and is typically 12-15. Since the nucleotide alphabet contains only four letters, there are $N_L = 4^L$ different L-mers for which the SA locations have to be stored. For example, if $L = L_{\max} = 14$, $N_L \sim 268\text{M}$ and for 33-bit SA indices it will require 1GB of storage. All L-mers with $L < L_{\max}$ will require 1/3 more of storage space. Using the L-mer indices we can immediately bound each search in the SA for all strings longer than L_{\max} , and obtain the complete answer for all strings shorter than L_{\max} . This procedure makes the SA search more local and speeds it up by a factor of 2-4.

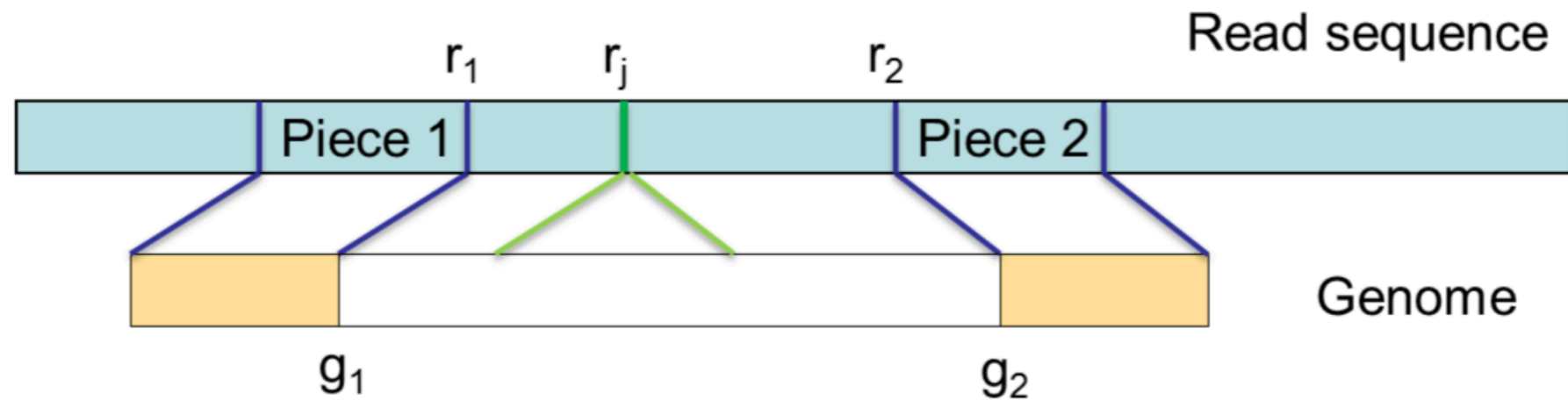
Seeds are clustered into windows and “stitched”

Seeds are filtered by frequency of occurrence to select “anchors” (essentially, infrequently occurring seeds)

Alignment windows (genomic regions) are selected around anchors

All co-linear matches within an alignment window (anchor and non-anchor) are stitched together to form a linear alignment for the whole fragment (ends of a paired-end read are treated as a single fragment)

Stitching and Extension



$$\max_{r_1 < r_j < r_2} \left\{ \sum_{r=1}^{r_j - r_1} \begin{bmatrix} 1 & \text{if } R(r_1 + r) = G(g_1 + r) \text{ \& } R(r_1 + r) \neq G(g_1 + r + \Delta) \\ -1 & \text{if } R(r_1 + r) \neq G(g_1 + r) \text{ \& } R(r_1 + r) = G(g_1 + r + \Delta) \\ 0 & \text{otherwise} \end{bmatrix} - P_{gap}(r_j) \right\}$$

Note: this is a modified recurrence that allows only **1** gap between the two “pieces” being stitched together. This leads to a runtime that is proportional to $r_2 - r_1$, but this places structural constraints on the types of alignments that can be found.

Scoring alignments

$$S = + \sum_{\text{match}} P_m - \sum_{\text{mismatch}} P_{mm} - \sum_{\text{insertion}} P_{ins} - \sum_{\text{deletion}} P_{del} - \sum_{\text{gap}} P_{gap} .$$

The alignment is scored in a straightforward way. Here, ins / del are indels in the stitching, while “gap” is taken to be an intronic gap between parts of a read or read ends. Gaps are scored differently, with a logarithmic penalty in their length.

Collecting Alignment Results

Finally, alignments from all alignment windows are collected and sorted by score and alignments within a user-specified distance from the best-scoring alignment are reported.

STAR has other abilities we won't discuss in detail (e.g. finding chimeric alignments by letting reads span multiple alignment windows), and has been *heavily* updated since publication (still in active development). It's now also commonly used for e.g. fusion detection and can even align circular transcripts or allow back-splicing in alignments.

We will explore the *results* from the STAR paper in a later lecture along with results from other “full text” aligners.