Suffix arrays

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Suffix array

$T\$$ = abaaba$ ← As with suffix tree, *T* is part of 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

Suffix array

$O(m)$ space, same as suffix tree. Is constant factor smaller?

32-bit integer can distinguish characters in the human genome, so suffix array is ~12 GB, smaller than MUMmer's 47 GB suffix tree.

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: binary search

Python has **`bisect`** module for binary search

`bisect.bisect_left(a, x)`: Leftmost offset where we can insert **x** into **a** to maintain sorted order. **a** is already sorted!

`bisect.bisect_right(a, x)`: Like **`bisect_left`**, but returning *rightmost* instead of leftmost offset

```
from bisect import bisect_left, bisect_right

a = [1, 2, 3, 3, 3, 4, 5]
print(bisect_left(a, 3), bisect_right(a, 3)) # output: (2, 5)

a = [2, 4, 6, 8, 10]
print(bisect_left(a, 5), bisect_right(a, 5)) # output: (2, 2)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

We can straightforwardly use binary search to find a range of elements in a sorted list that *equal* some query:

```
from bisect import bisect_left, bisect_right

strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# Get range of elements that equal query string ‘awl’
st, en = bisect_left(strls, 'awl'), bisect_right(strls, 'awl')

print(st, en) # output: (2, 3)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

Can also use binary search to find a range of elements in a sorted list with some query as a *prefix*:

```
from bisect import bisect_left, bisect_right

strls = ['a', 'awkward', 'awl', 'awls', 'axe', 'axes', 'bee']

# Get range of elements with ‘aw’ as a prefix
st, en = bisect_left(strls, 'aw'), bisect_left(strls, 'ax')

print(st, en) # output: (1, 4)
```

Python example: http://nbviewer.ipython.org/6753277

Suffix array: binary search

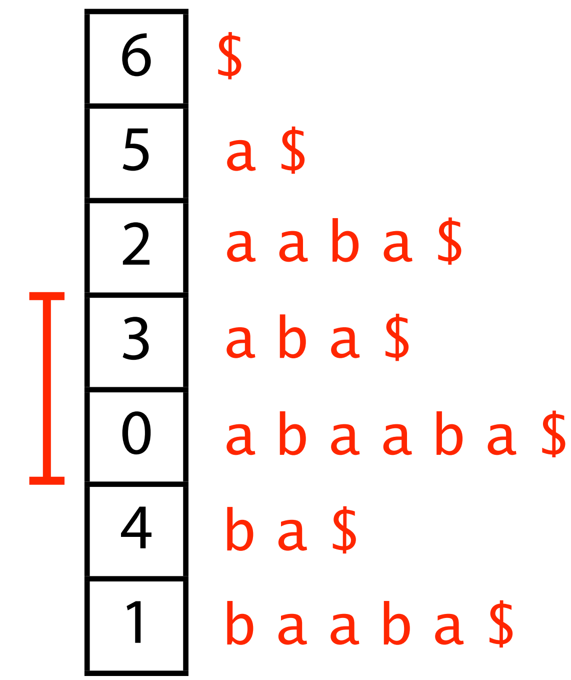
We can do the same thing for a sorted list of suffixes:

```python
from bisect import bisect_left, bisect_right

t = 'abaaba$'
suffixes = sorted([t[i:] for i in xrange(len(t))])

st, en = bisect_left(suffixes, 'aba'),
         bisect_left(suffixes, 'abb')

print(st, en) # output: (3, 5)
```



Python example: http://nbviewer.ipython.org/6753277

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?

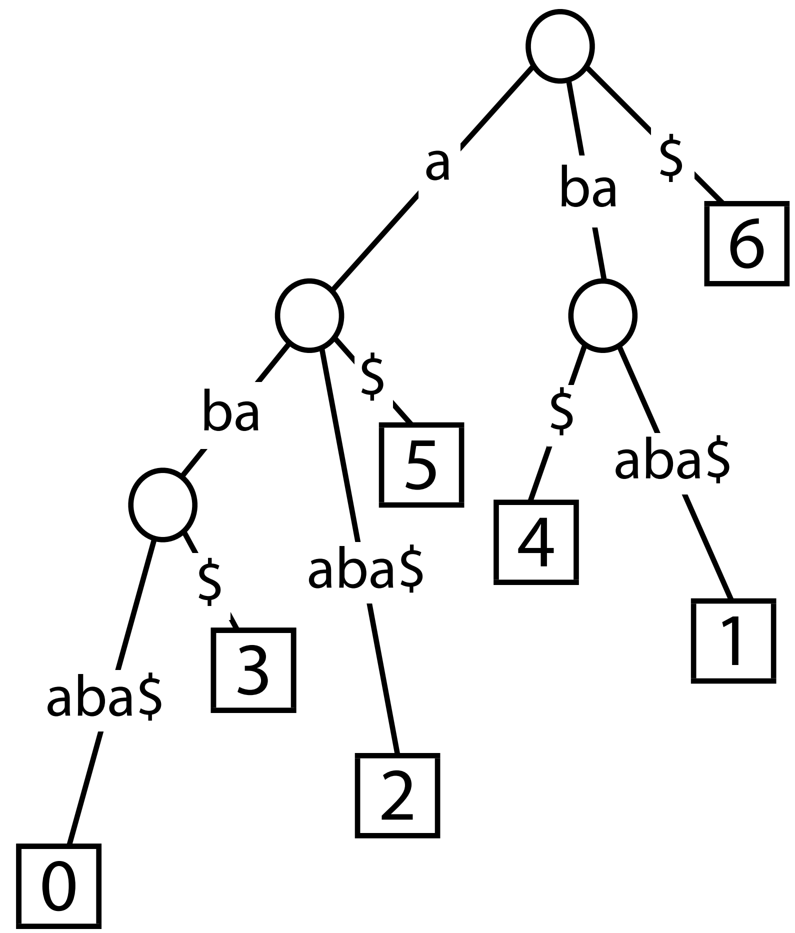
$O(\log_2 m)$ bisections, $O(n)$ comparisons per bisection, so $O(n \log m)$

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

Contrast suffix array: O(n log m) with suffix tree: O(n)

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 $



But we can improve bound for suffix array...

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

"left" "center" "right"

Notation: We'll use use **SA[*l*]** to refer to the suffix corresponding to suffix-array element ***l***. We could write *T*[**SA[*l*]**:], but that's too verbose.

Throughout the search, invariant is maintained:

SA[*l*] < *P* < **SA[*r*]**

Suffix array: querying

Throughout search, invariant is maintained:

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

What do we do at each iteration?

Let c = floor(($r + l$) / 2)

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

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

When to stop?

$P < \mathbf{SA}[c]$ and $c = l + 1$ - answer is c

$P > \mathbf{SA}[c]$ and $c = r - 1$ - answer is r

Suffix array: querying

```
def binarySearchSA(t, sa, p):
    assert t[-1] == '$' # t already has terminator
    assert len(t) == len(sa) # sa is the suffix array for t
    if len(t) == 1: return 1
    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]
    while True:
        c = (l + r) // 2
        # determine whether p < T[sa[c]:] by doing comparisons
        # starting from left-hand sides of p and T[sa[c]:]
        plt = True # assume p < T[sa[c]:] until proven otherwise
        i = 0
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far
        if plt:
            if c == l + 1: return c
            r = c
        else:
            if c == r - 1: return r
            l = c
```

loop iterations ≈ length of Longest Common Prefix (LCP) of *P* and **SA[*c*]**

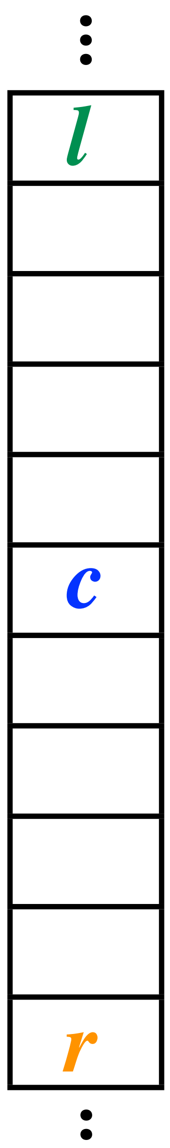
If we already know something about LCP of *P* and **SA[*c*]**, we can save work

Python example: http://nbviewer.ipython.org/6765182

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.

SA(T)



LCP(***P***, **SA[*l*]**) = 3

"Length of the LCP"

LCP(***P***, **SA[*c*]**) ≥ 3

LCP(***P***, **SA[*r*]**) = 5

More generally:

LCP(***P***, **SA[*c*]**) ≥ **min(**LCP(***P***, **SA[*l*]**), LCP(***P***, **SA[*r*]**)**)**

We can skip character comparisons

Suffix array: querying

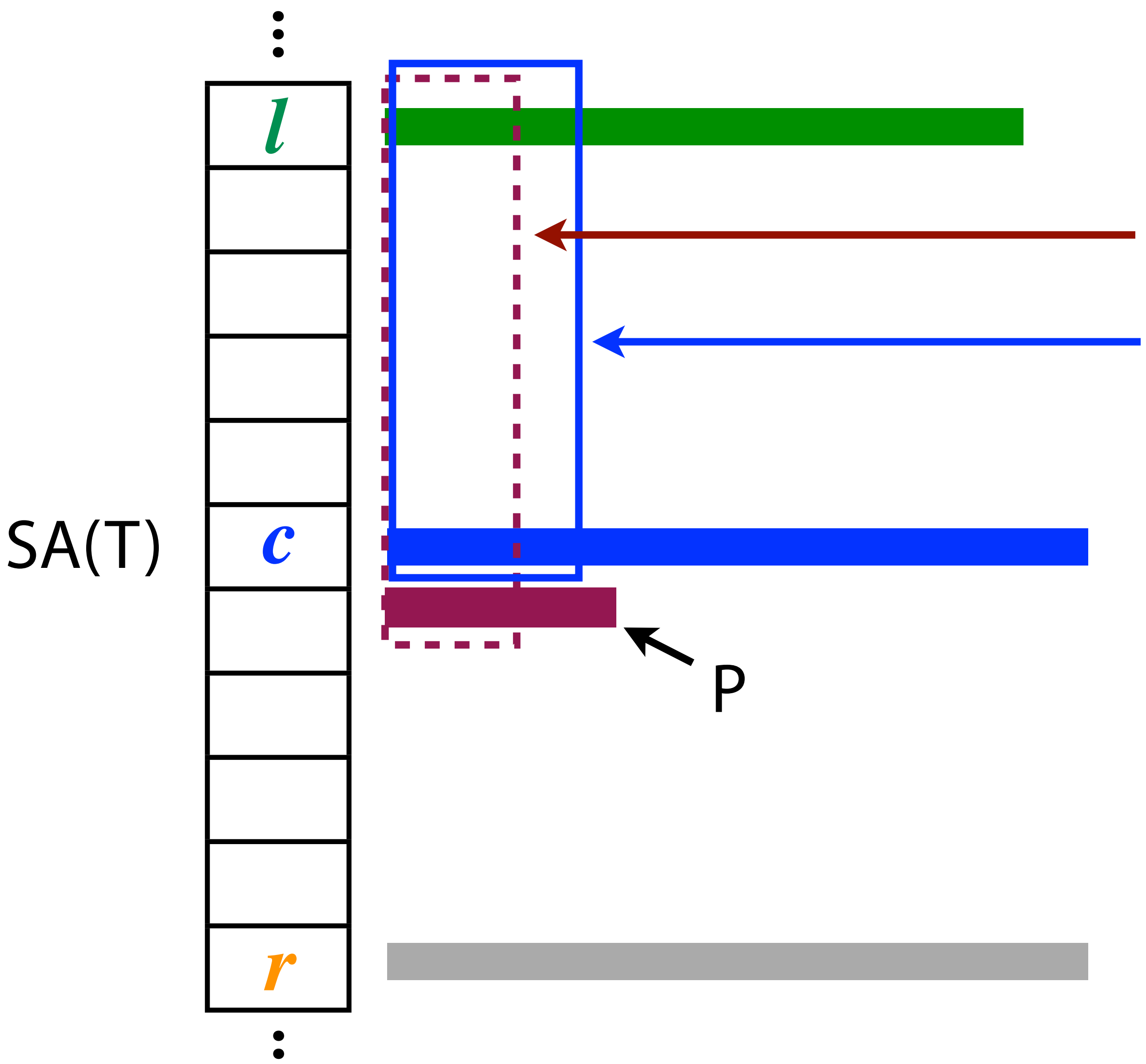
```
def binarySearchSA_lcp1(t, sa, p):
    if len(t) == 1: return 1
    l, r = 0, len(sa) # invariant: sa[l] < p < sa[r]
    lcp_lp, lcp_rp = 0, 0
    while True:
        c = (l + r) // 2
        plt = True
        i = min(lcp_lp, lcp_rp)
        while i < len(p) and sa[c]+i < len(t):
            if p[i] < t[sa[c]+i]:
                break # p < T[sa[c]:]
            elif p[i] > t[sa[c]+i]:
                plt = False
                break # p > T[sa[c]:]
            i += 1 # tied so far
        if plt:
            if c == l + 1: return c
            r = c
            lcp_rp = i
        else:
            if c == r - 1: return r
            l = c
            lcp_lp = i
```

Worst-case time bound is still $O(n \log m)$, but we're closer

Python example: http://nbviewer.ipython.org/6765182

Suffix array: querying

Take an iteration of binary search:



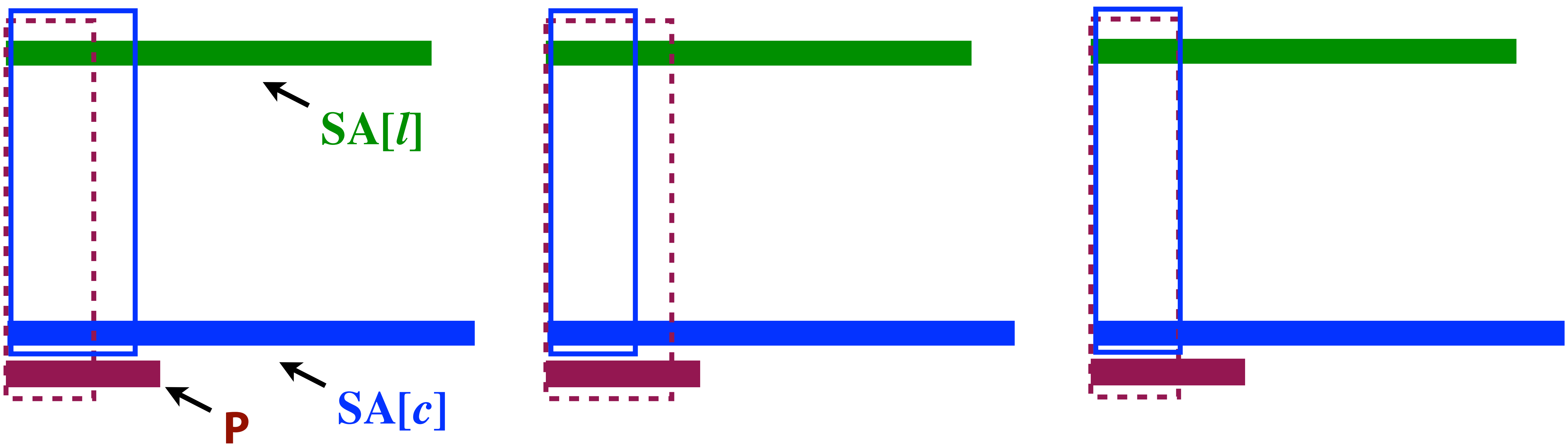
Say we know

LCP(*P*, **SA[*l*]**), and

LCP(**SA[*c*]**, **SA[*l*]**)

Suffix array: querying

Three cases:



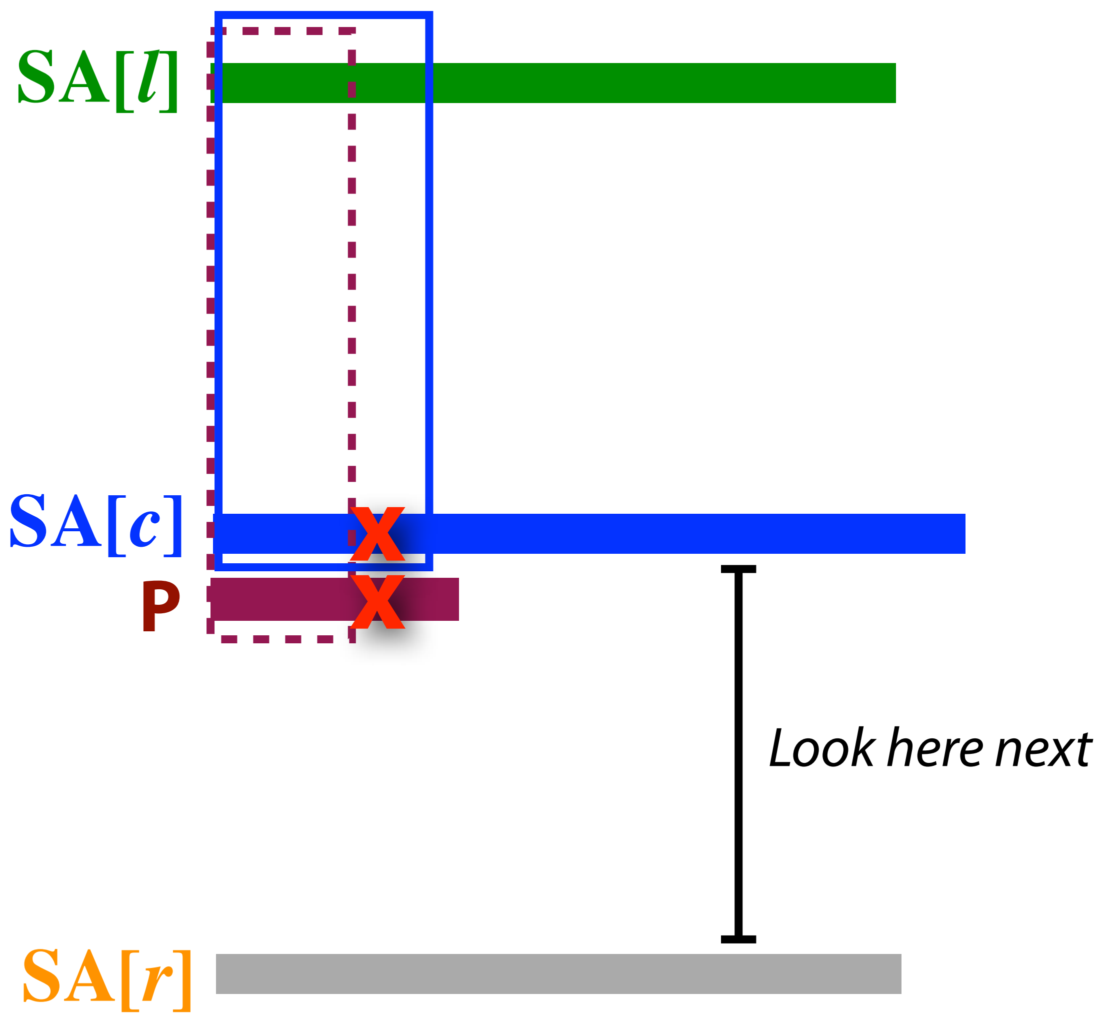
LCP(**SA[*c*]**, **SA[*l*]**) > LCP(***P***, **SA[*l*]**)

LCP(**SA[*c*]**, **SA[*l*]**) < LCP(***P***, **SA[*l*]**)

LCP(**SA[*c*]**, **SA[*l*]**) = LCP(***P***, **SA[*l*]**)

Suffix array: querying

Case 1:



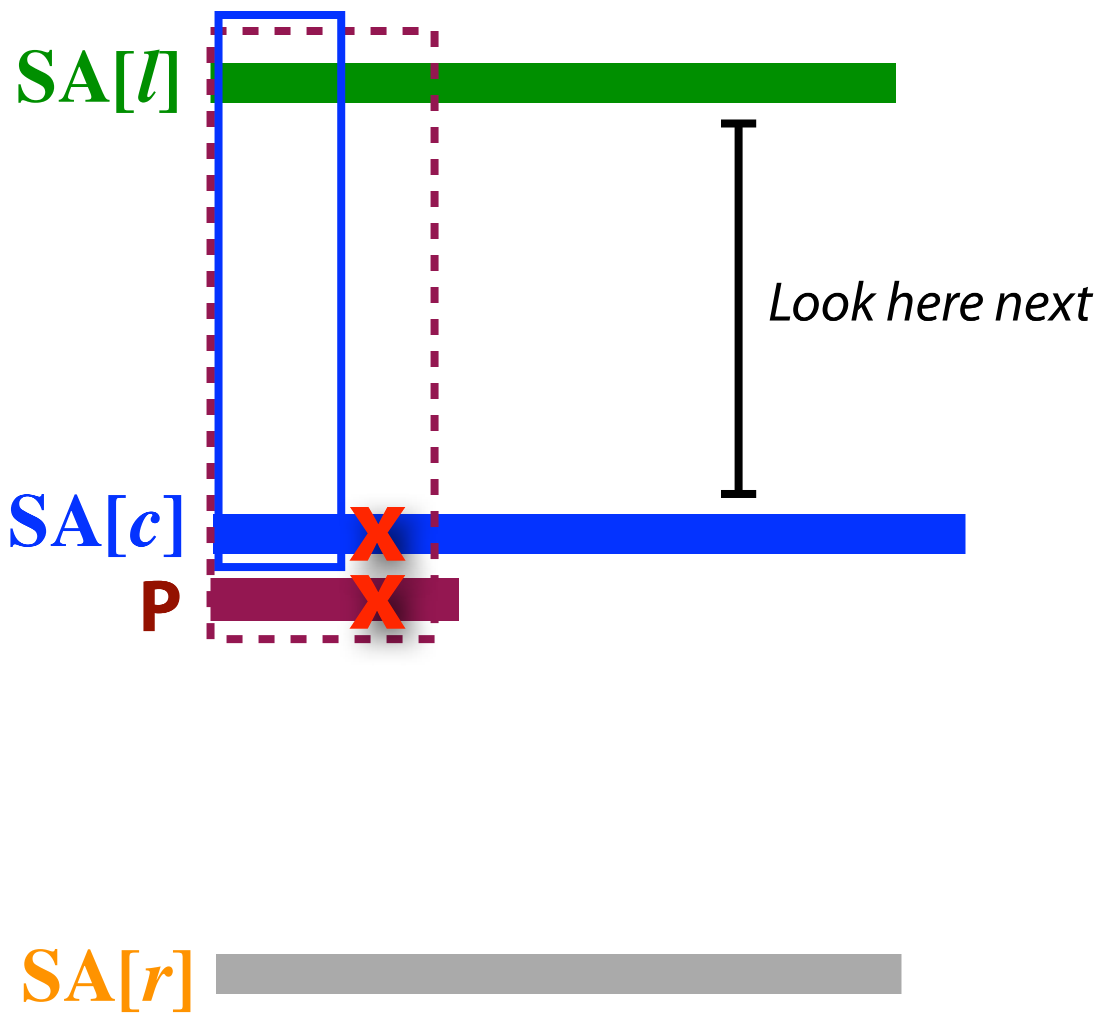
Next char of *P* after the LCP(*P*, SA[*l*]) must be *greater than* corresponding char of SA[*c*]

P > SA[*c*]

LCP(SA[*c*], SA[*l*]) > LCP(*P*, SA[*l*])

Suffix array: querying

Case 2:



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

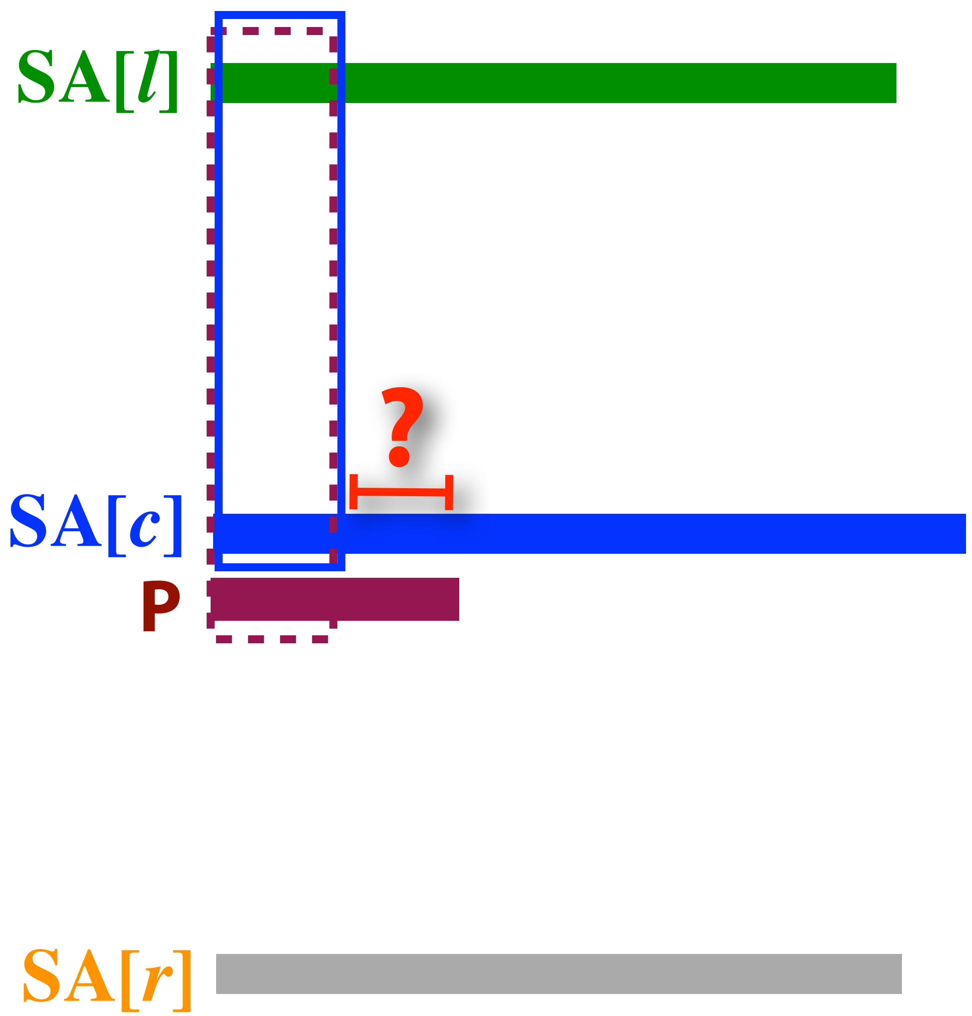
P < **SA[*c*]**

LCP(**SA[*c*]**, **SA[*l*]**) <

LCP(***P***, **SA[*l*]**)

Suffix array: querying

Case 3:



Must do further character comparisons between **P** and **SA[*c*]**

Each such comparison either:

(a) mismatches, leading to a bisection

(b) matches, in which case LCP(***P***, **SA[c]**) grows

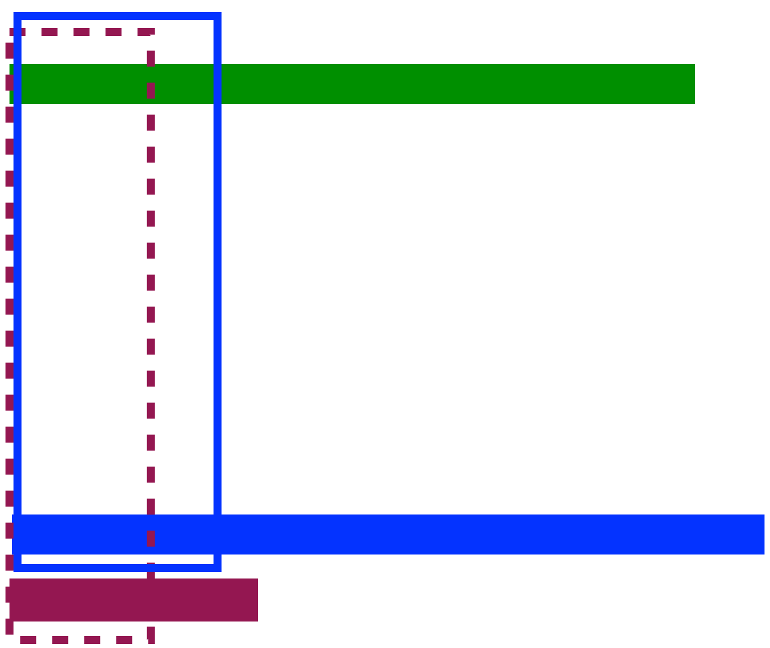
LCP(**SA[*c*]**, **SA[*l*]**) =

LCP(***P***, **SA[*l*]**)

Suffix array: querying

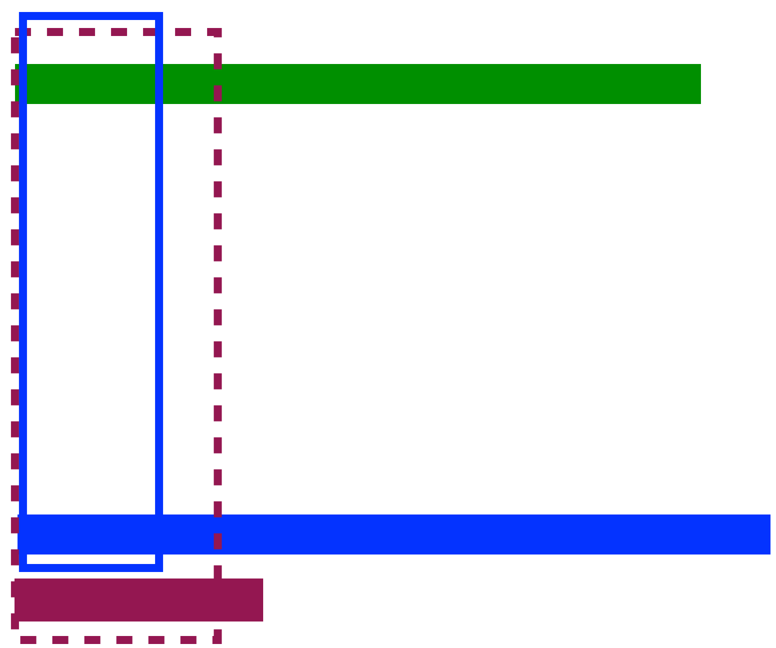
We improved binary search on suffix array from $O(n \log m)$ to $O(n + \log m)$ using information about Longest Common Prefixes (LCPs).

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



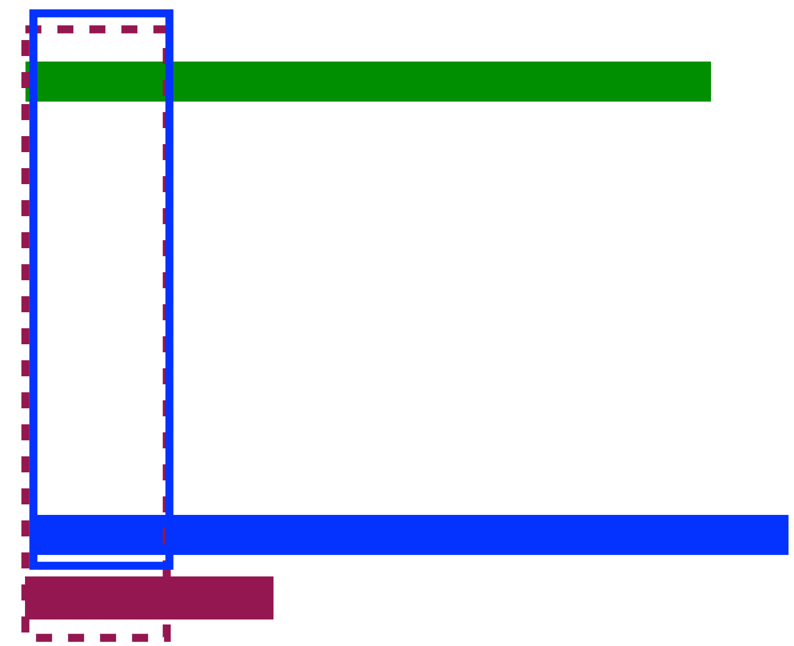
$\mathrm{LCP}(\mathbf{SA}[c], \mathbf{SA}[l]) > \mathrm{LCP}(P, \mathbf{SA}[l])$

Bisect right!



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

Bisect left!



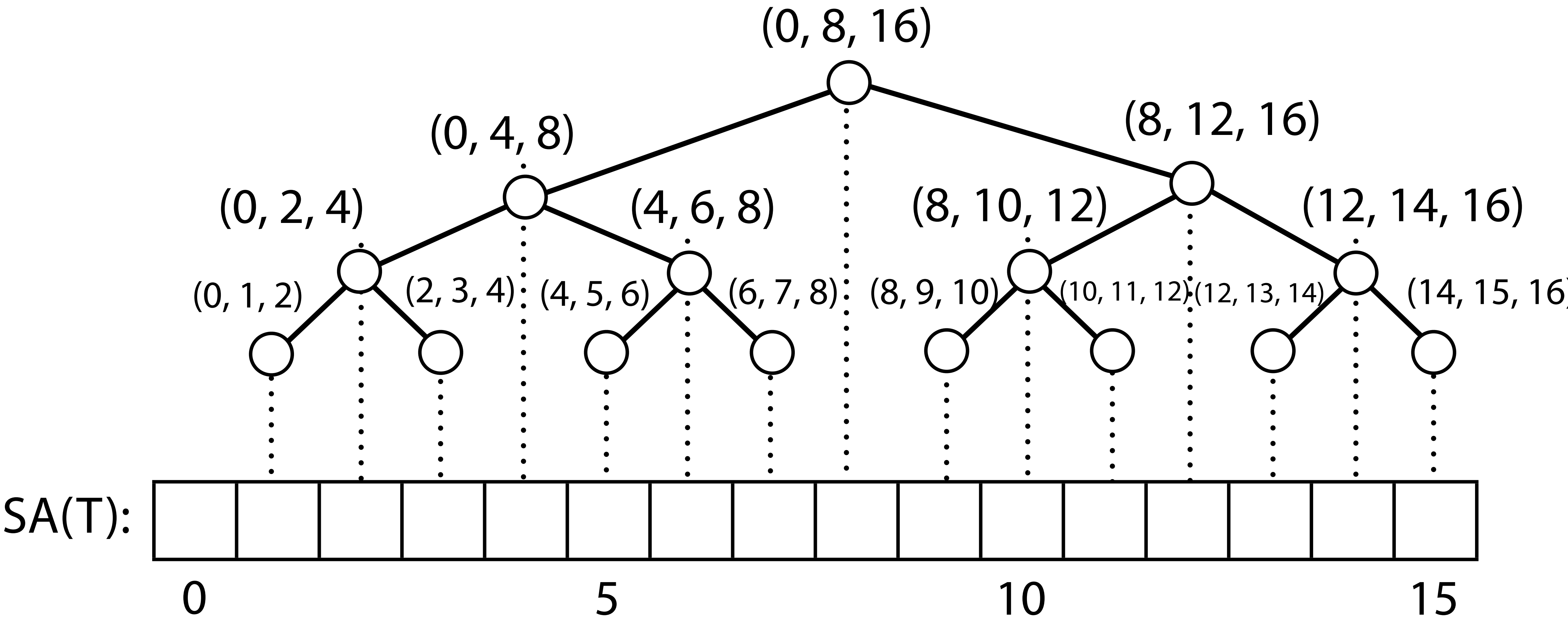
$\mathrm{LCP}(\mathbf{SA}[c], \mathbf{SA}[l]) = \mathrm{LCP}(P, \mathbf{SA}[l])$

Compare some characters, then bisect!

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

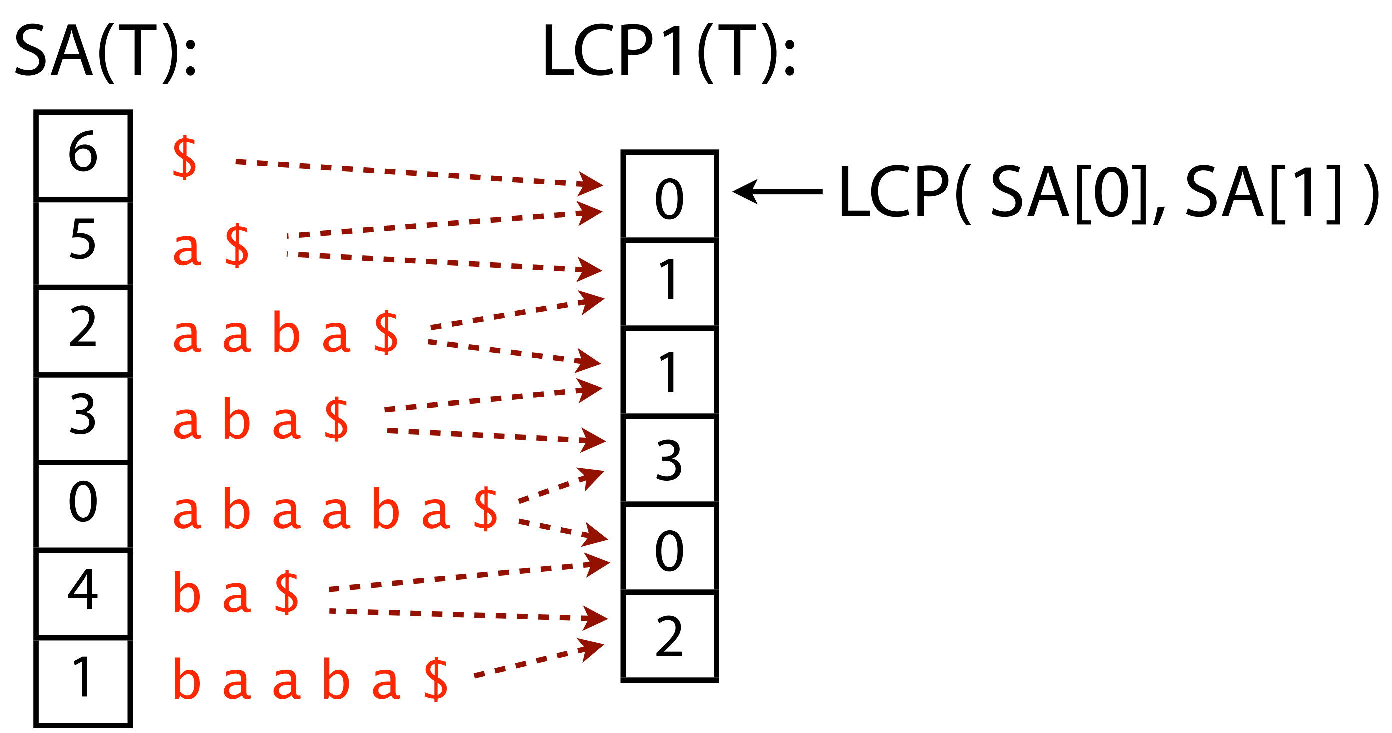


Example where *m* = 16 (incl. **$**)   # search tree nodes = *m* -1

Suffix array: LCPs

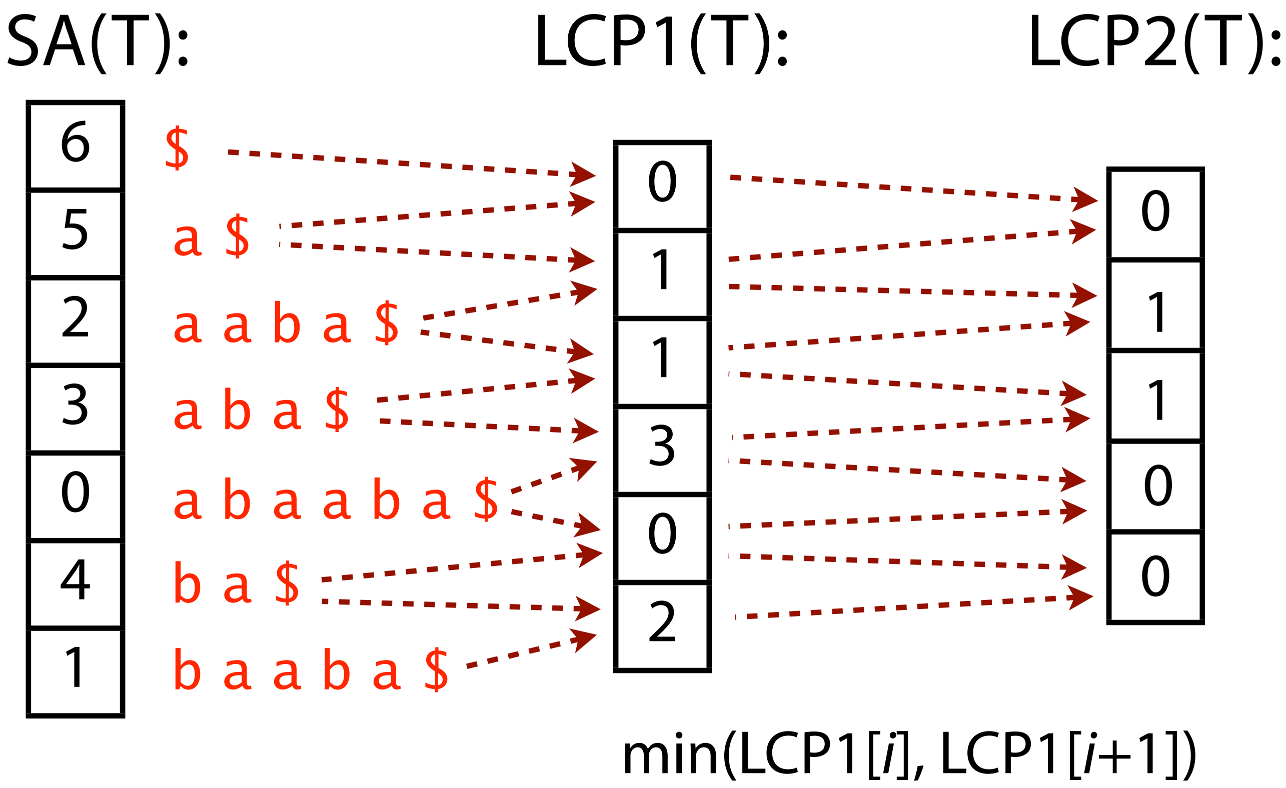
Suffix Array (SA) has *m* elements

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



Suffix array: LCPs

$LCP2[i] = LCP(SA[i], SA[i+1], SA[i+2])$

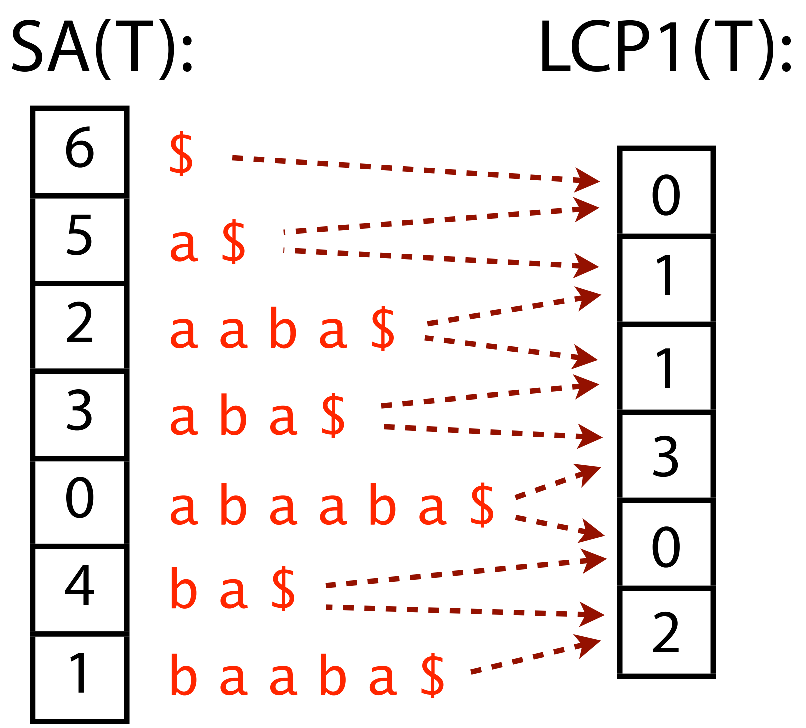


In fact, LCP of a range of consecutive suffixes in SA equals the minimum LCP1 among adjacent pairs in the range

LCP1 is a building block for other useful LCPs

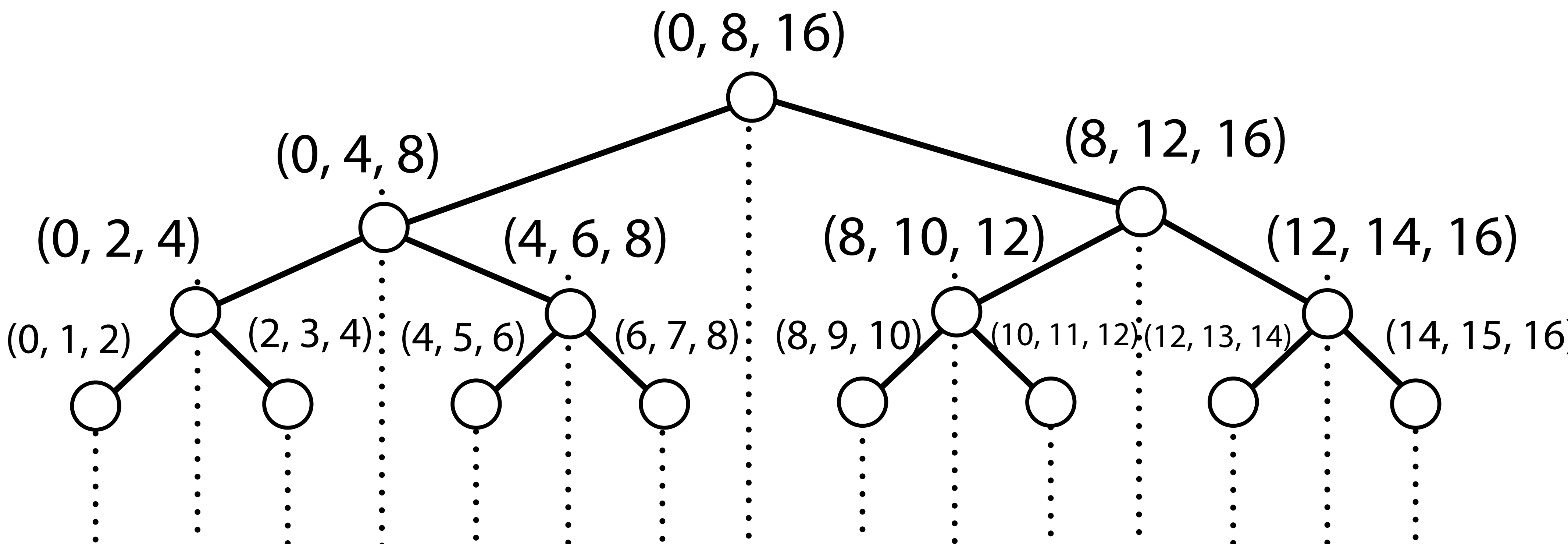
Suffix array: LCPs

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

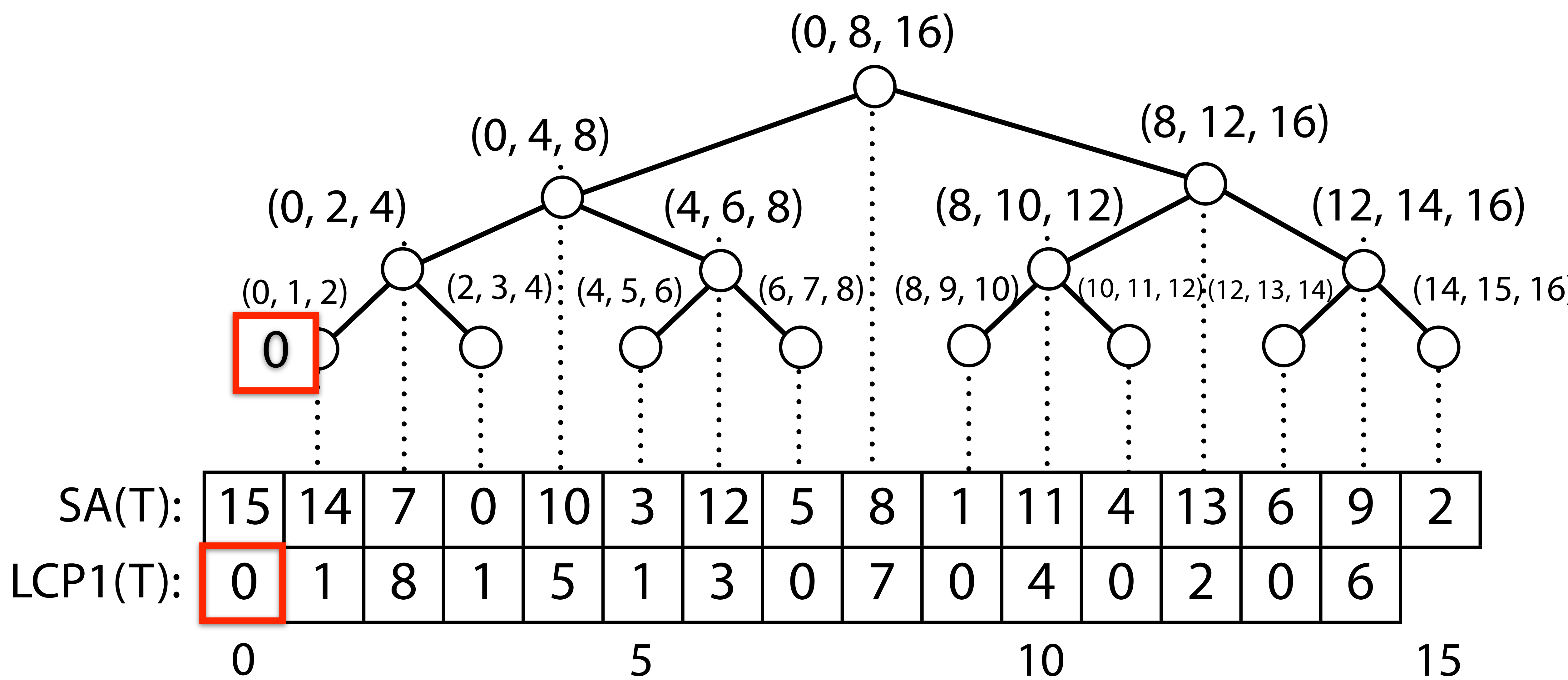
T = abracadabracada



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	
	0					5					10					15

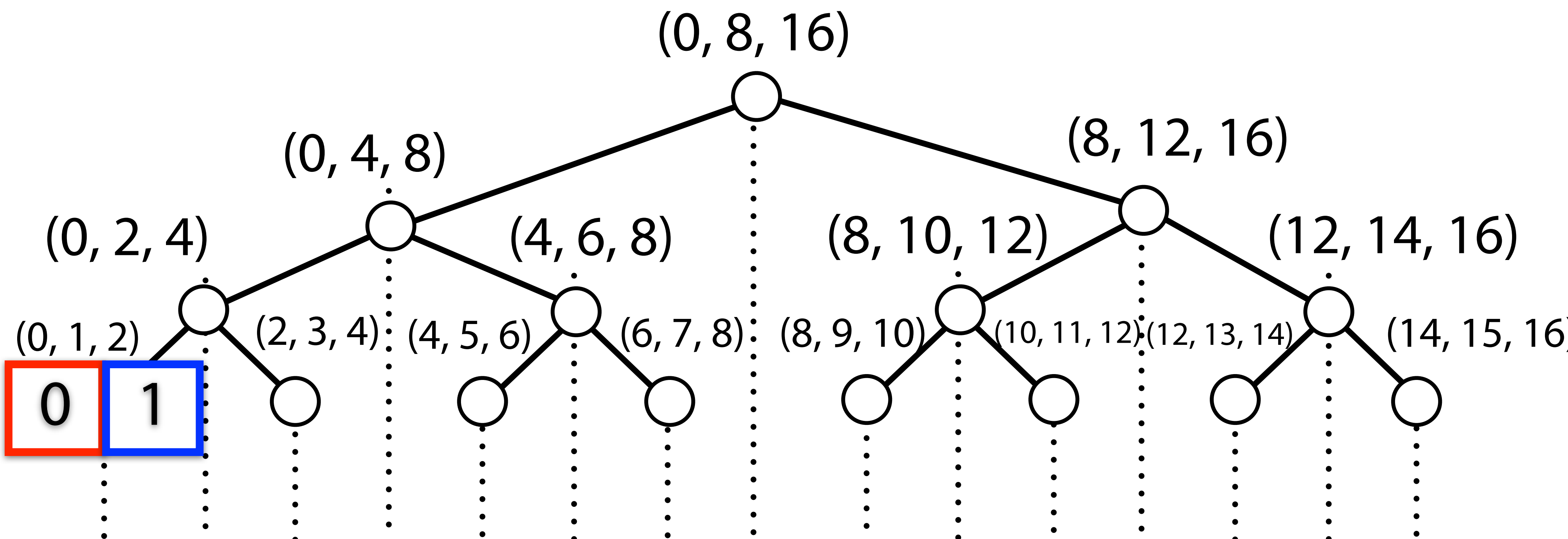
Suffix array: LCPs

T = abracadabracada



Suffix array: LCPs

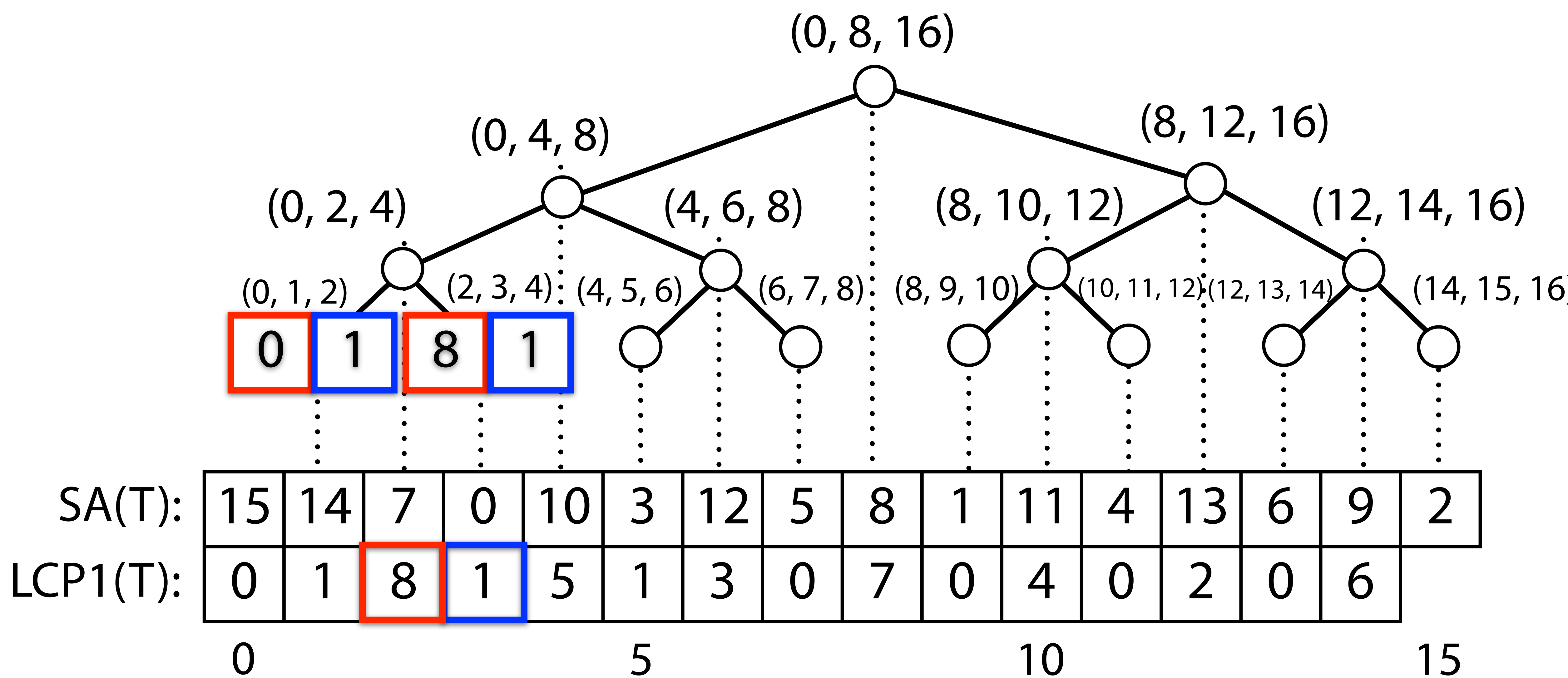
T = abracadabracada



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	
	0					5					10					15

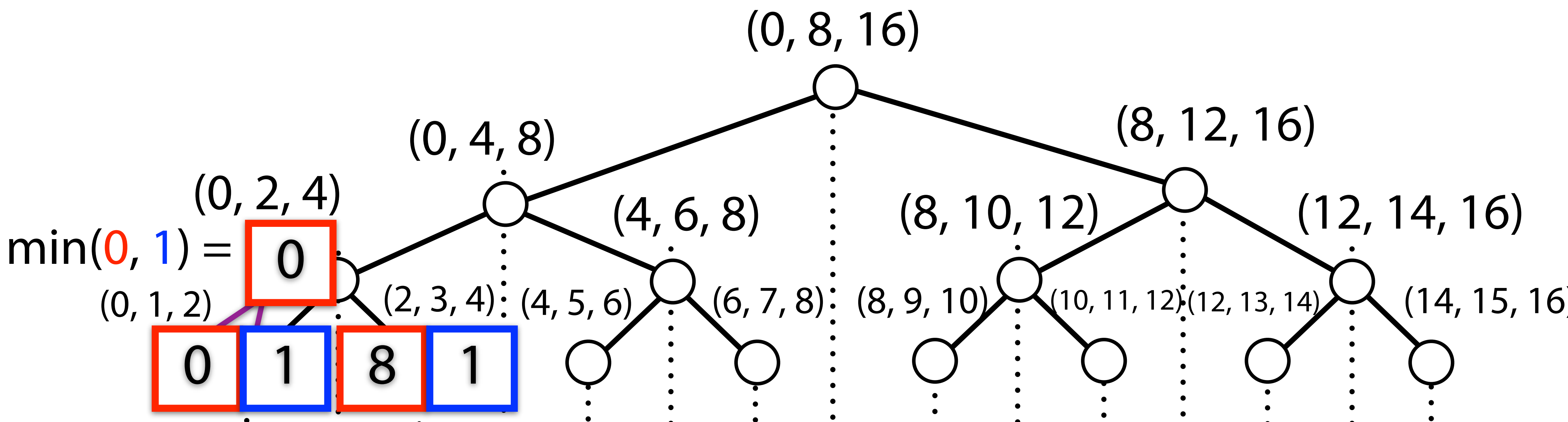
Suffix array: LCPs

T = abracadabracada



Suffix array: LCPs

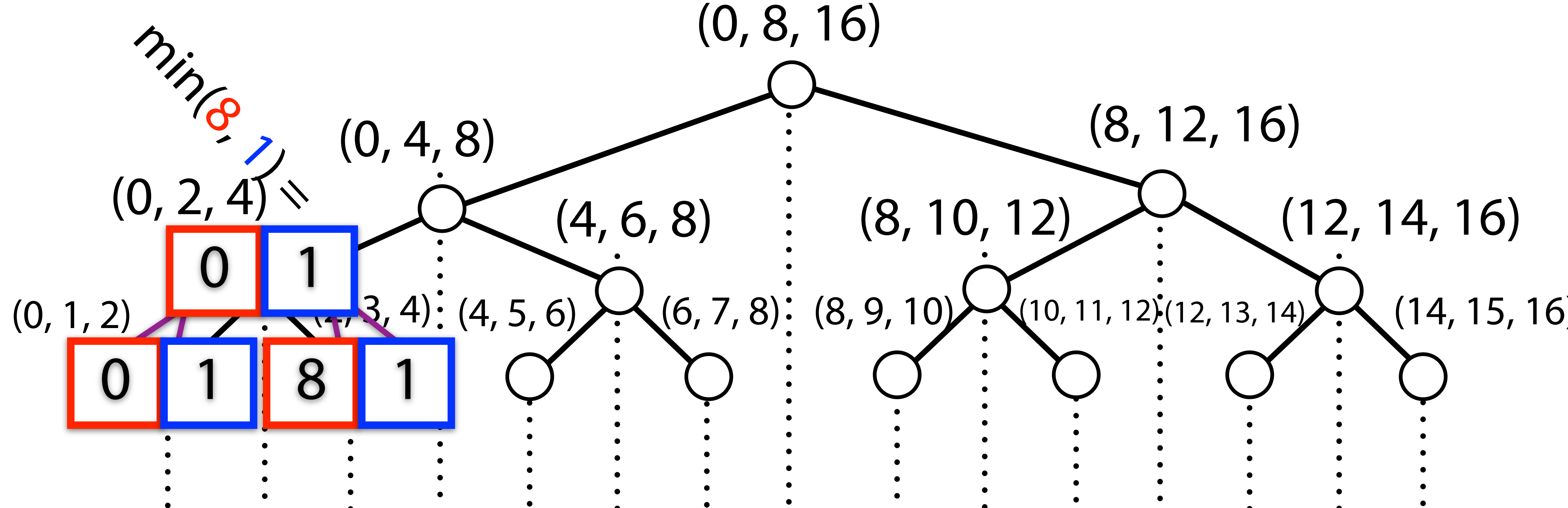
T = abracadabracada



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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	

Suffix array: LCPs

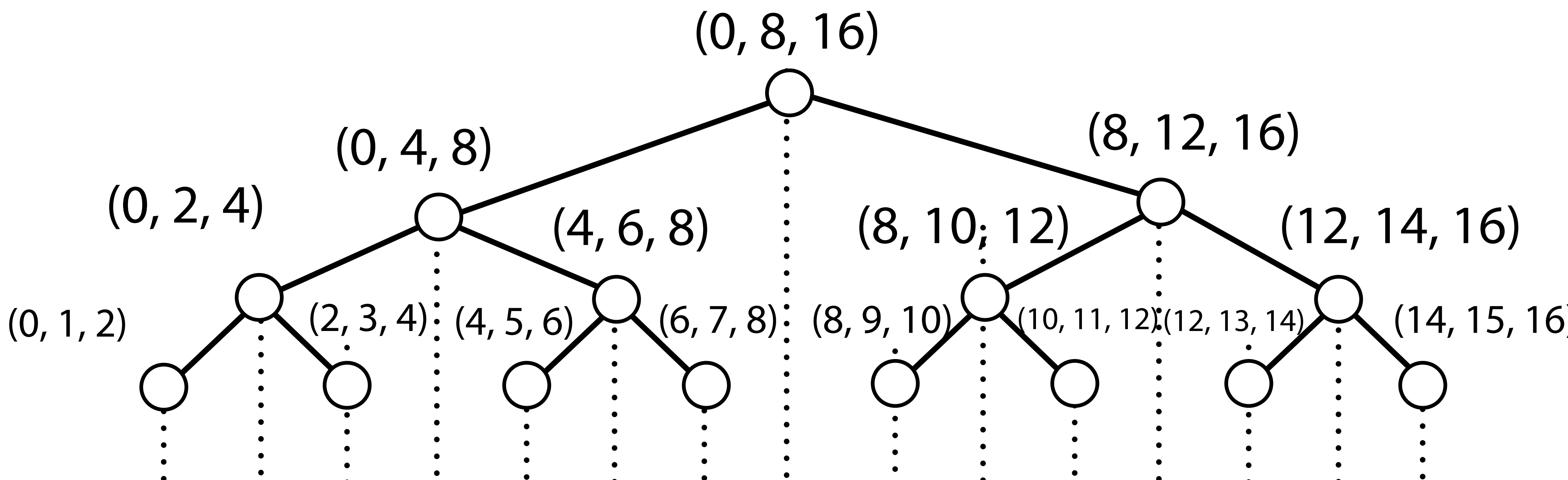
T = abracadabracada



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

Suffix array: LCPs

```
# Calculates (l, c) LCPs and (c, r) LCPs from LCP1 array.  Returns
# pair where first element is list of LCPs for (l, c) combos and
# second is LCPs for (c, r) combos.
def precomputeLcps(lcp1):
    llcp, rlcp = [None] * len(lcp1), [None] * len(lcp1)
    lcp1 += [0]
    def precomputeLcpsHelper(l, r):
        if l == r-1: return lcp1[l]
        c = (l + r) // 2
        llcp[c-1] = precomputeLcpsHelper(l, c)
        rlcp[c-1] = precomputeLcpsHelper(c, r)
        return min(llcp[c-1], rlcp[c-1])
    precomputeLcpsHelper(0, len(lcp1))
    return llcp, rlcp
```

$O(m)$ time and space

Python example: http://nbviewer.ipython.org/6783863

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.
3. Binary search with skipping using all LCPs, including LCPs among T's suffixes. $O(n + \log m)$.

Gusfield: "Simple Accelerant"

Gusfield: "Super Accelerant"

How much space do they require?

1. ~m integers (SA)
2. ~m integers (SA)
3. ~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

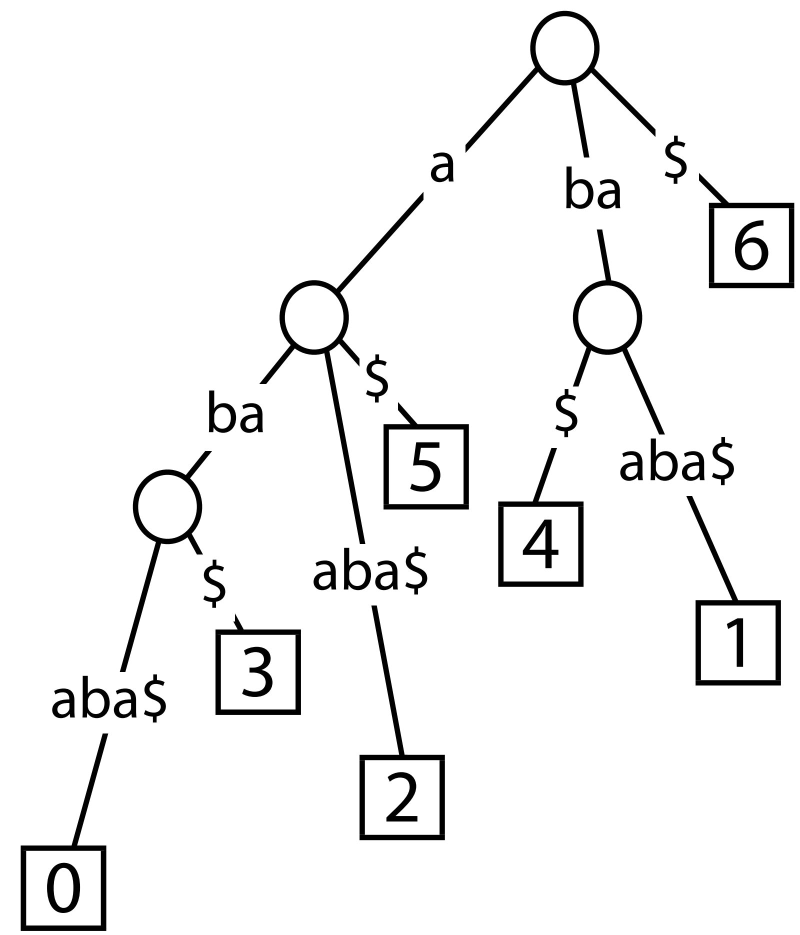
Matching 500K 100-nt substrings to the ~ 5 million nt-long *E. coli* genome. Substrings drawn randomly from the genome.

Index building time not included

Suffix array: building

Given *T*, how to we efficiently build *T*'s suffix array?

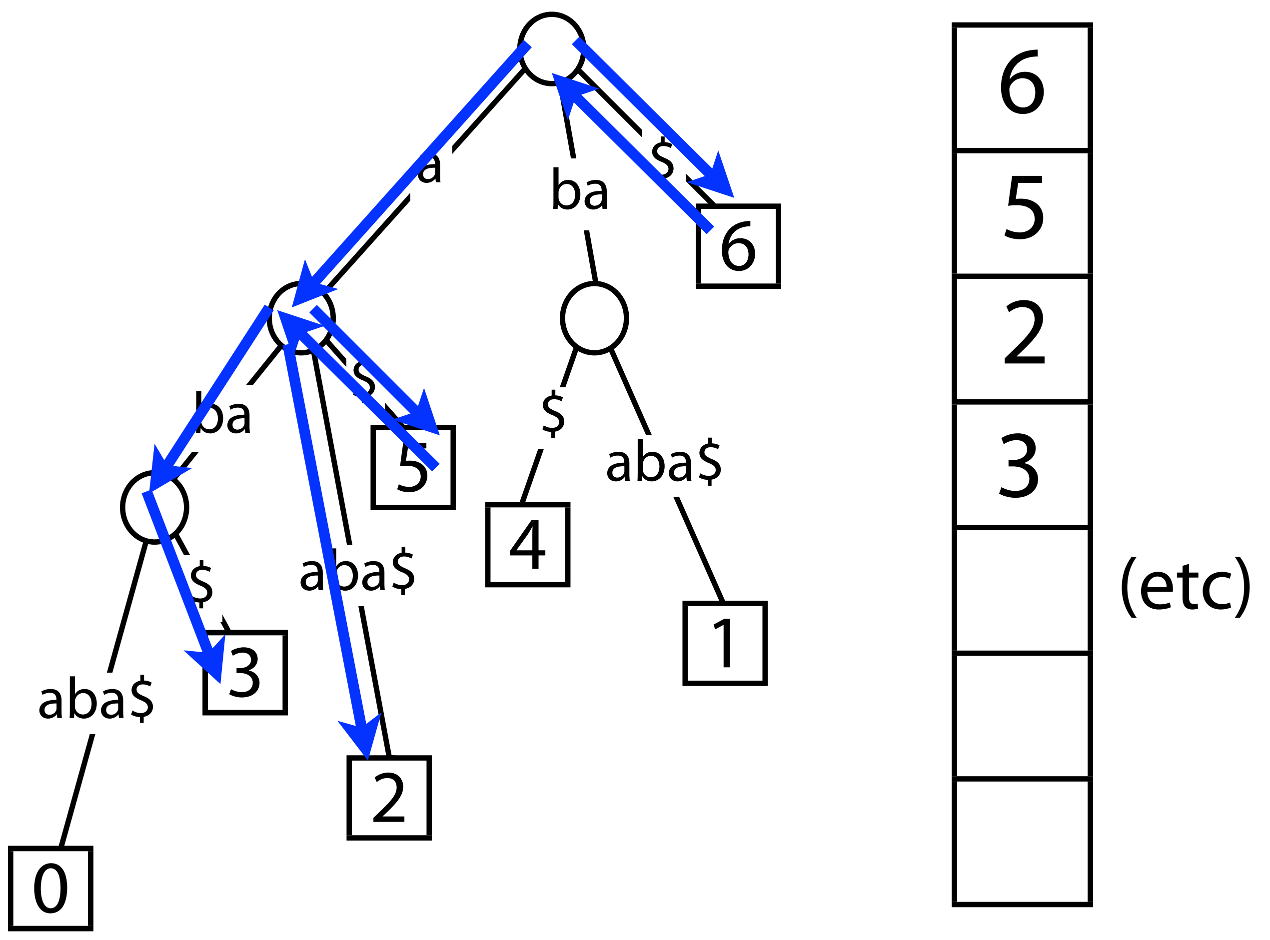
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $



Suffix array: building SA

Idea: Build suffix tree, do a lexicographic depth-first traversal reporting leaf offsets as we go

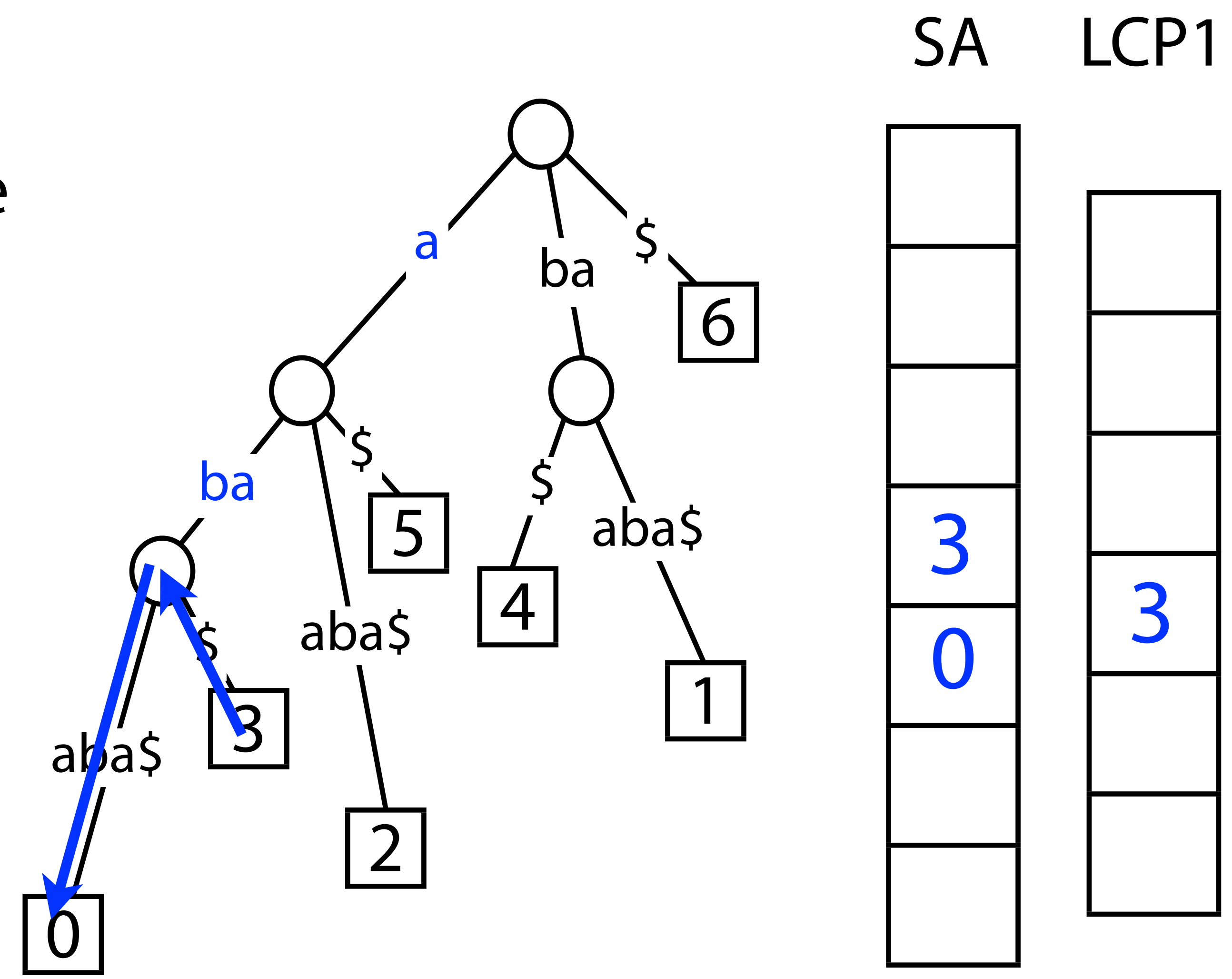
Traverse $O(m)$ nodes and emit m integers, so $O(m)$ time assuming edges are already ordered



Suffix array: building LCP1

Can calculate LCP1 at the same time

Yes: on our way from one leaf to the next, record the shallowest "label depth" observed



Suffix array: SA and LCP from suffix tree: implementation

```
def saLcp(self):
    # Return suffix array and an LCP1 array corresponding to this
    # suffix tree.  self.root is root, self.t is the text.
    self.minSinceLeaf = 0
    sa, lcp1 = [], []
    def __visit(n):
        if len(n.out) == 0:
            # leaf node, record offset and LCP1 with previous leaf
            sa.append(len(self.t) - n.depth)
            lcp1.append(self.minSinceLeaf)
            # reset LCP1 to depth of this leaf
            self.minSinceLeaf = n.depth
        # visit children in lexicographical order
        for c, child in sorted(n.out.iteritems()):
            __visit(child)
            # after each child visit, perhaps decrease
            # minimum-depth-since-last-leaf value
            self.minSinceLeaf = min(self.minSinceLeaf, n.depth)
    __visit(self.root)
    return sa, lcp1[1:]
```

This is a member function from a SuffixTree class, the rest of which isn't shown

Python example: http://nbviewer.ipython.org/6796858

Suffix array: building

Suffix trees are big. Given *T*, how do we efficiently build *T*'s suffix array *without* first building a suffix tree?

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: 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 Sedgewick. "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.

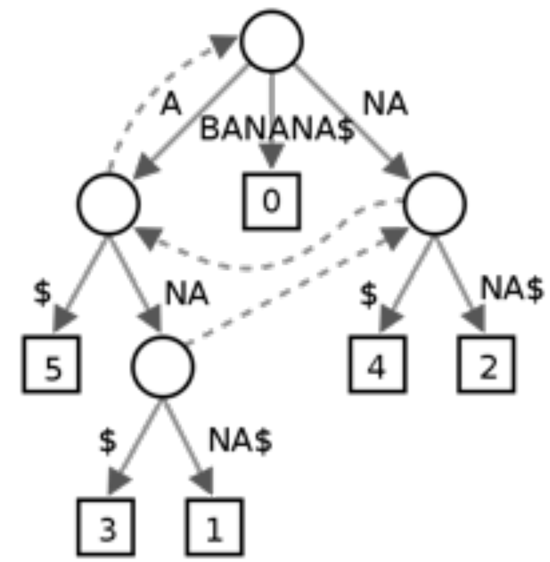
And there are comparable advances with repsect to LCP1

Suffix array: summary

Suffix array gives us index that is:

(a) Just *m* integers, with O(*n* log *m*) worst-case query time, but close to O(*n* + log *m*) in practice

or **(b)** 3*m* integers, with O(*n* + log *m*) worst case



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array

(a) will often be preferable: index for entire human genome fits in ~12 GB instead of > 45 GB