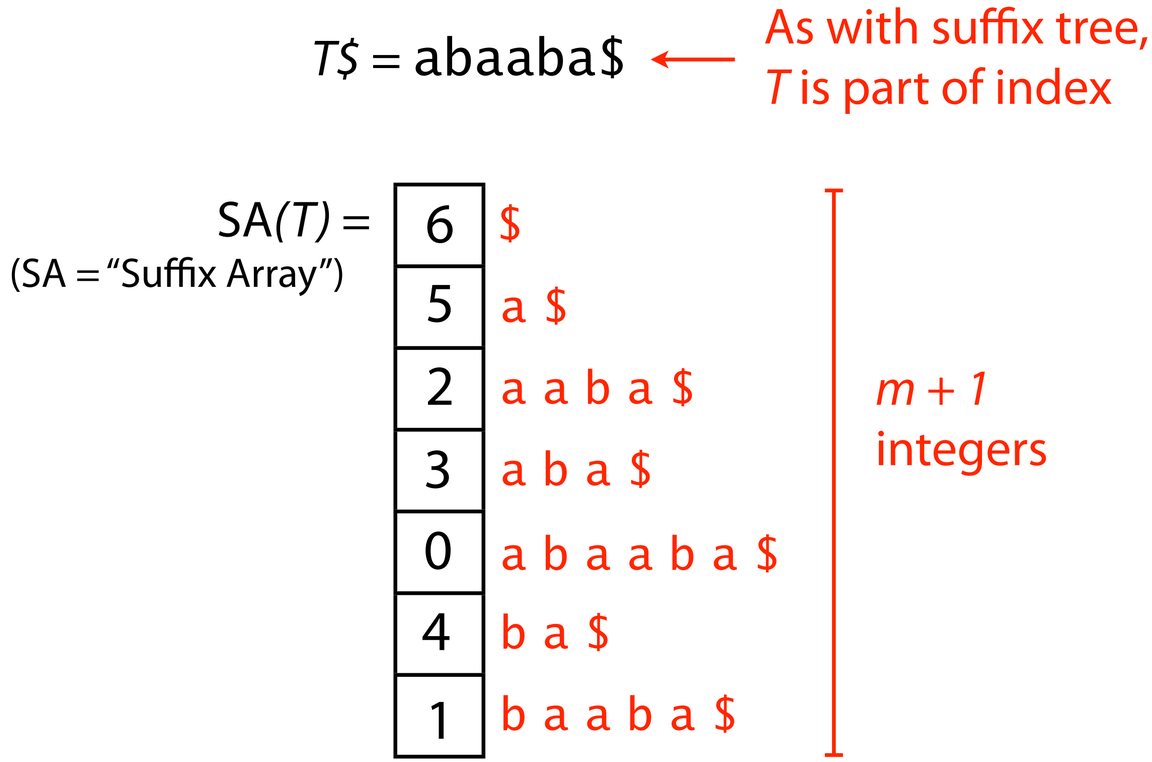
Suffix Arrays

UNIVERSITY OF MARYLAND

All slides in this lecture **not** marked with "*" are courtesy of Ben Langmead (www.langmead-lab.org/teaching-materials).

Suffix array



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 = cattcat$

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

index of suffix	suffix of s
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$

slide courtesy of Carl Kingsford

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
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
1
4
0
6
3
2

slide courtesy of Carl Kingsford

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.

*slide courtesy of Carl Kingsford

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

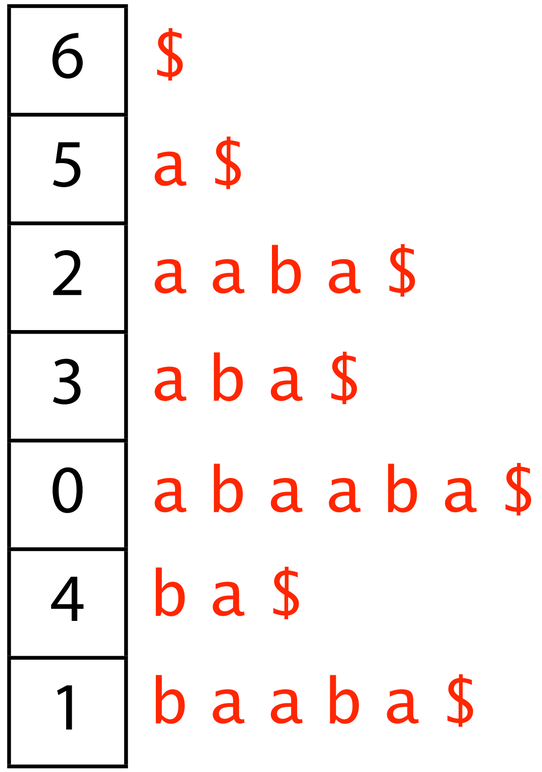
6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

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



Suffix array: binary search

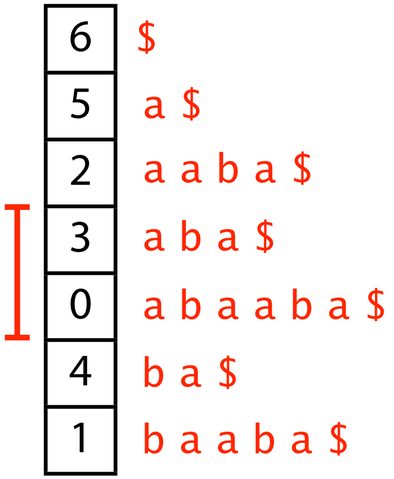
We can do the same thing for a sorted list of suffixes:

```
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*?

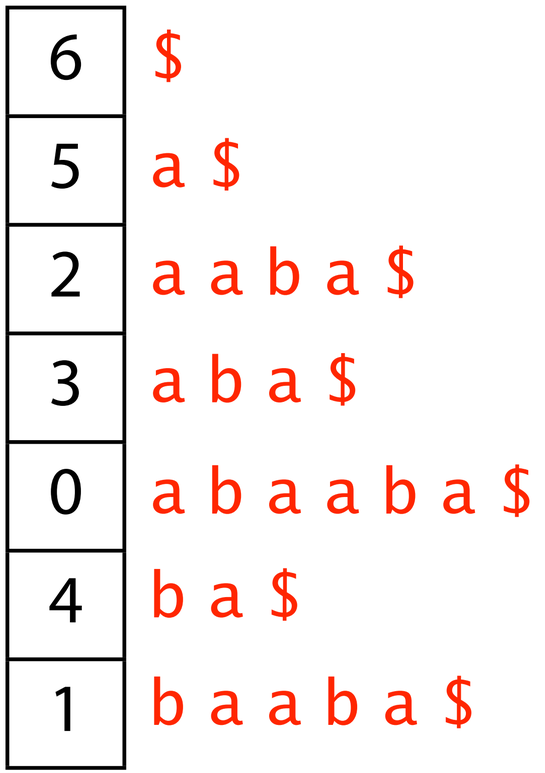
Do binary search, check whether *P* is a prefix of the suffix there

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 $O(n \log m)$

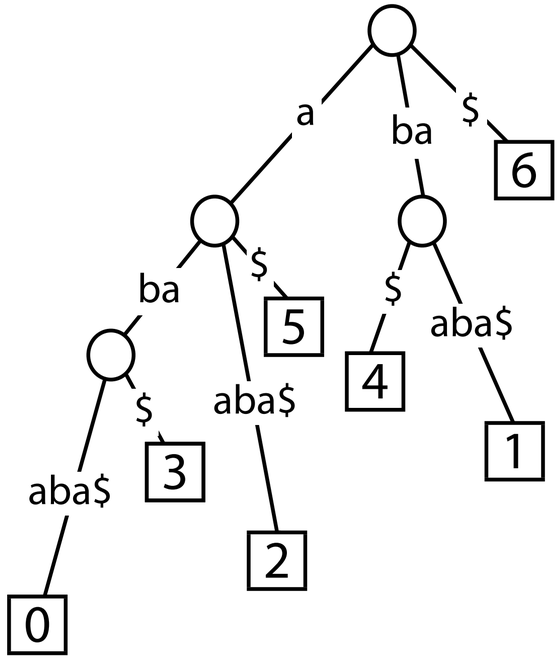


Suffix array: querying

Won't cover this DS. Provides asymptotically time-optimal operations for most queries, but with less-than-stellar constant factors.

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 $l = 0$, $c = floor(m/2)$ and $r = m$ (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:

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

What do we do at each iteration?

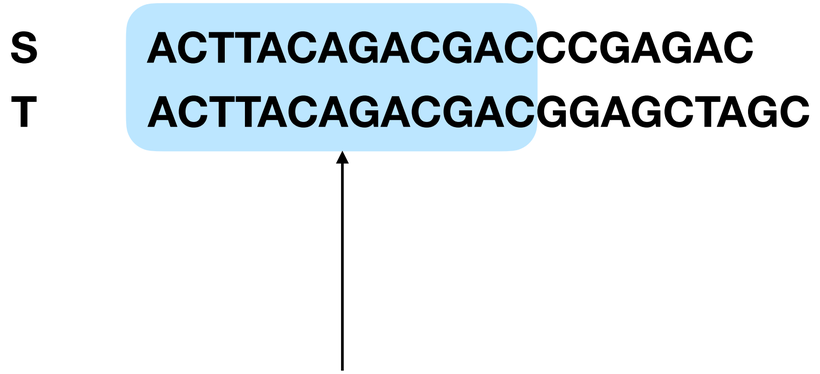
Let c = floor(($r + l$) / 2)
If $P < SA[c]$, either stop or let $r = c$ and iterate
If P > $SA[c]$, either stop or let $l = c$ and iterate

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).



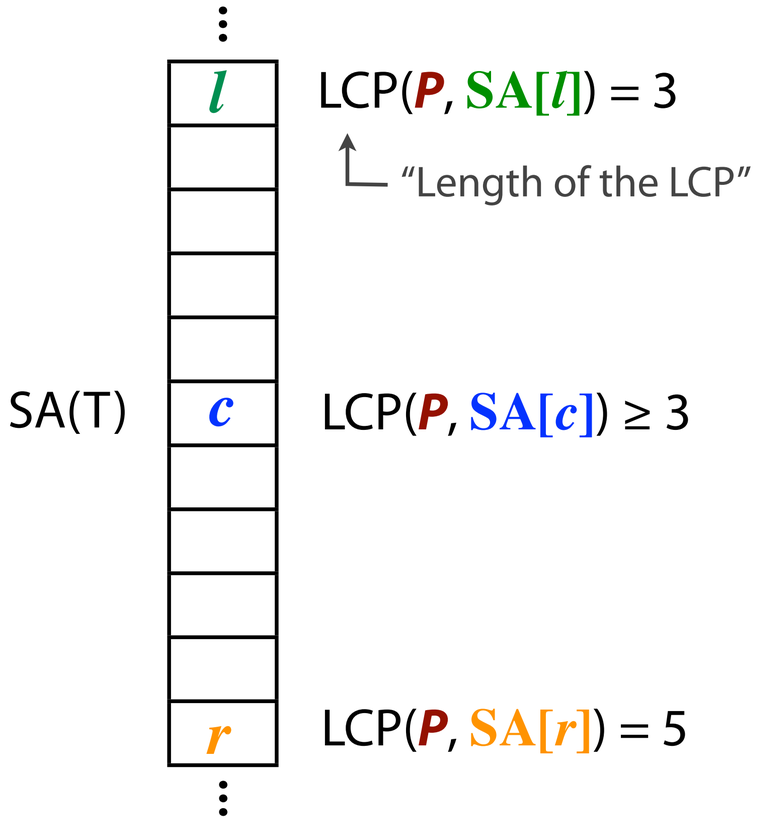
LCP(S,T) = ACTTACAGACGAC

|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 $\mathbf{SA}[c]$ and we've already compared *P* to $\mathbf{SA}[l]$ and $\mathbf{SA}[r]$ in previous iterations.



More generally:

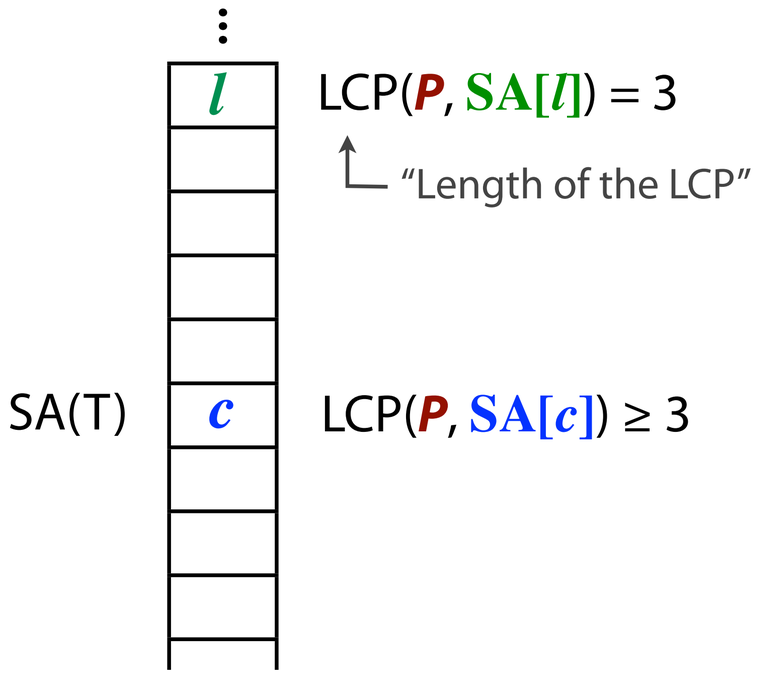
LCP(***P***, $\mathbf{SA}[c]$) ≥

min(LCP(***P***, $\mathbf{SA}[l]$), LCP(***P***, $\mathbf{SA}[r]$)**)**

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*]**) ≥

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

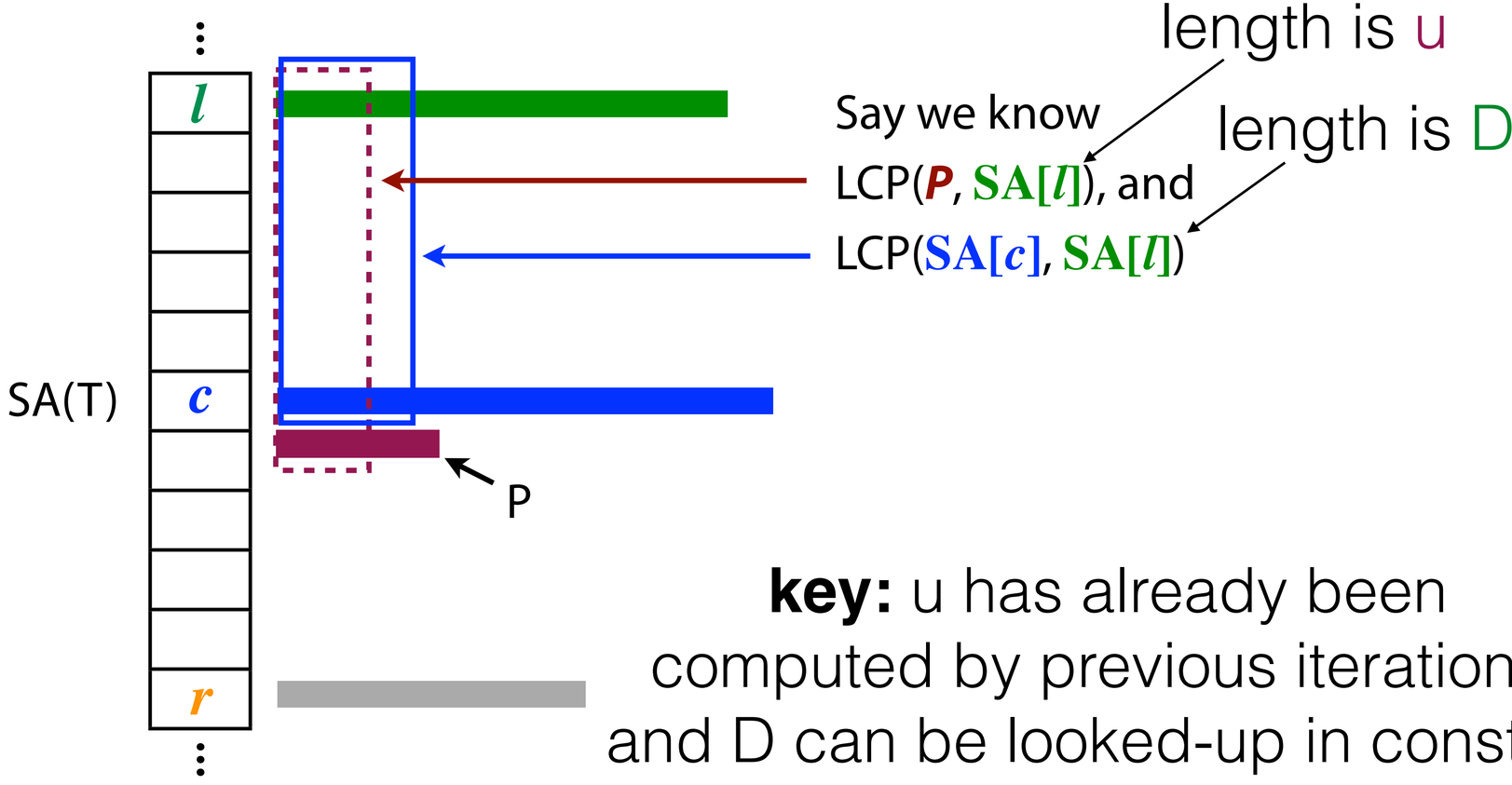
$S=ac_{M-2}b$, P=c

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

Suffix array: querying

Assume, wlog, that
D = LCP(SA[l], SA[c]) ≥ D' = LCP(SA[c], SA[r])
otherwise there are symmetric cases.

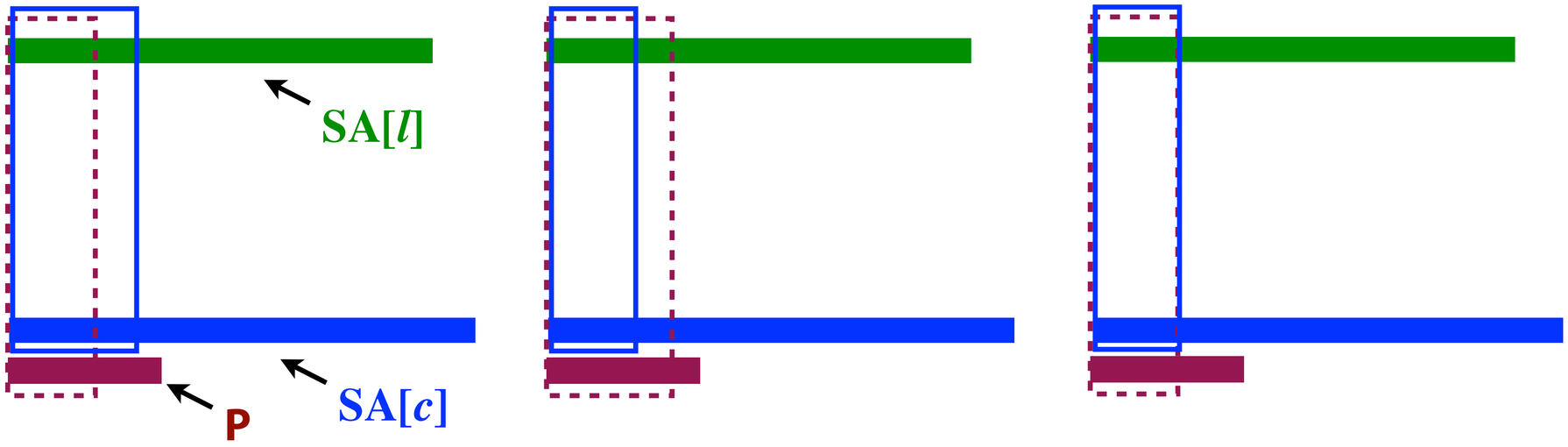
Take an iteration of binary search:



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' = LCP(P, SA[r]) is larger, 3 symmetric cases.



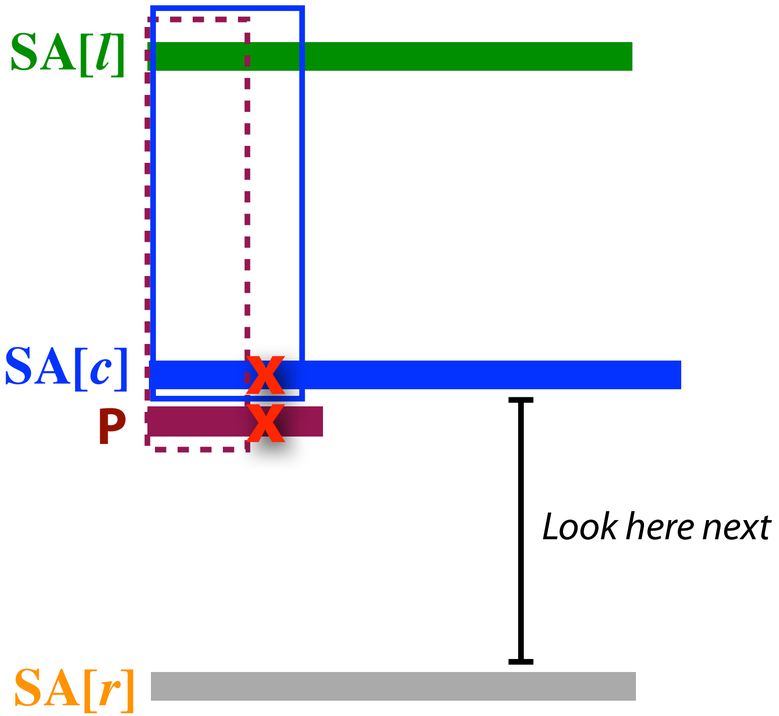
LCP(**SA[*c*]**, **SA[*l*]**) > LCP(***P***, **SA[*l*]**)

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

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

Suffix array: querying

Case 1: LCP(**SA[*c*]**, **SA[*l*]**) > LCP(***P***, **SA[*l*]**)



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

P > **SA[*c*]**

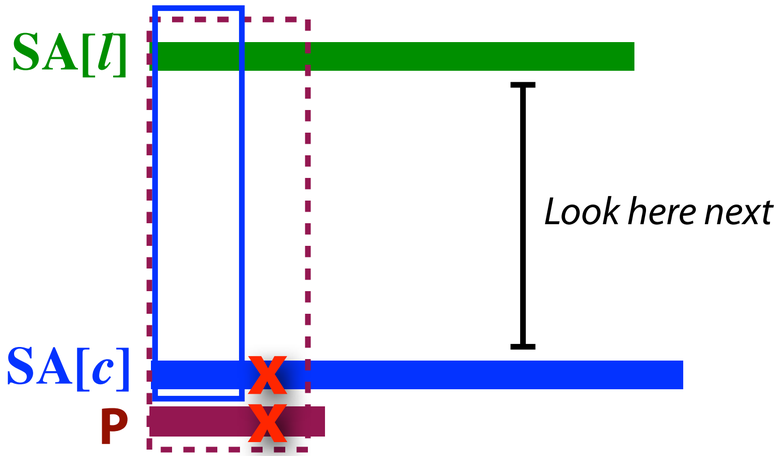
In this case, we compute LCP(P[u:], SA[c][u:]). c becomes our new l, and now we know the new LCP(P, SA[l]), b/c we just computed it!

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

Suffix array: querying

Case 2: LCP(**SA[*c*]**, **SA[*l*]**) <

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



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*]**) <

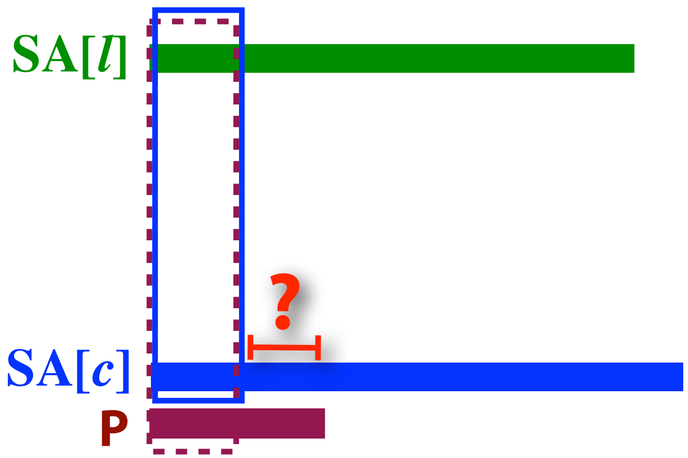
LCP(***P***, **SA[*l*]**)

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

Suffix array: querying

Case 3: LCP(**SA[*c*]**, **SA[*l*]**) =

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



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

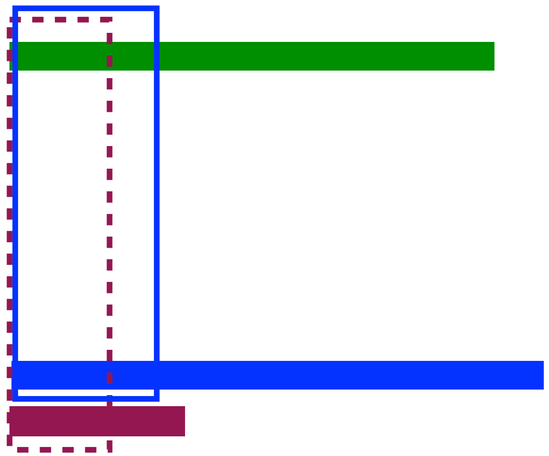
LCP(**SA[*c*]**, **SA[*l*]**) =

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

Suffix array: querying

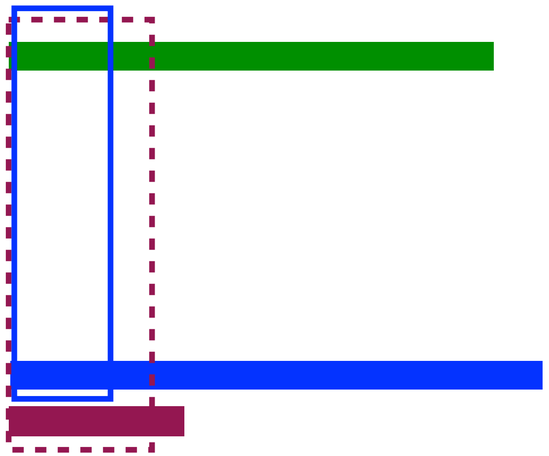
We improved binary search on suffix array from $O(n \log m)$ to $O(n + \log m)$ using information about Longest Common Prefixes (LCPs).

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



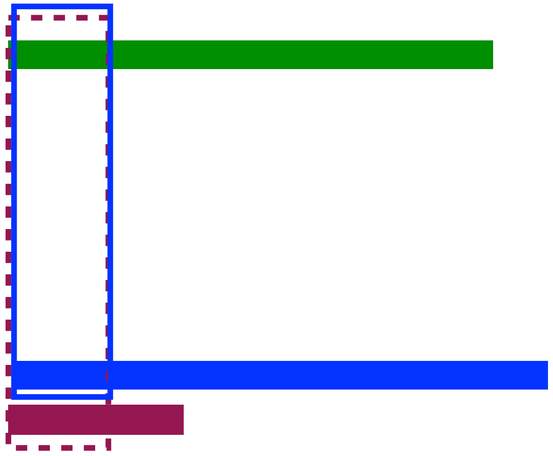
LCP(**SA[*c*]**, **SA[*l*]**) > LCP(***P***, **SA[*l*]**)

Bisect right!



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

Bisect left!



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

□

*slide courtesy of Carl Kingsford

How to pre-compute LCP

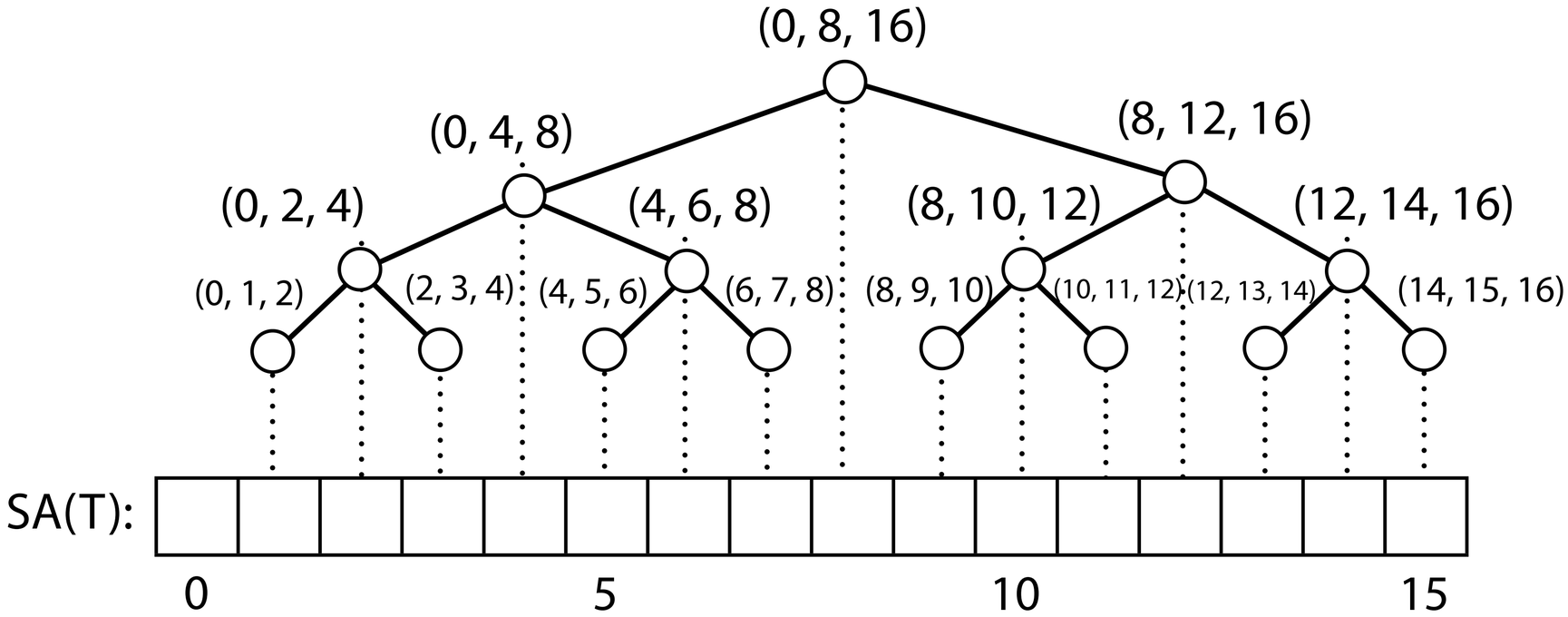
- To perform this “efficient” search, we must be able to look up LCP(SA[c], SA[l]) and LCP(SA[c], SA[r]).
- How can we pre-compute this information *efficiently?*
  - Which LCP values do we need (*hint: not all of them)*?
  - Given LCP for left and right sub-interval of a search, how can we compute LCP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

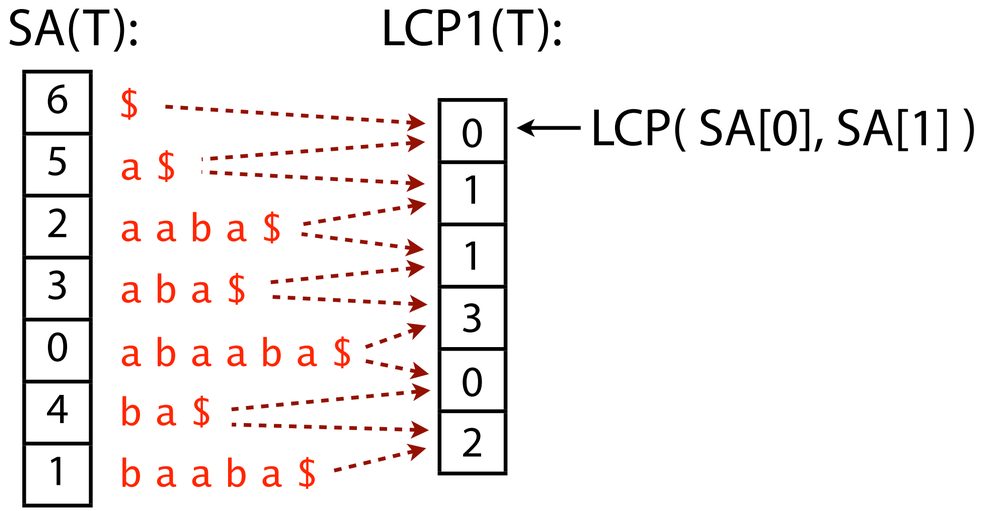


Example where $m = 16$ (incl. **$**) # search tree nodes = m -1

Suffix array: LCPs

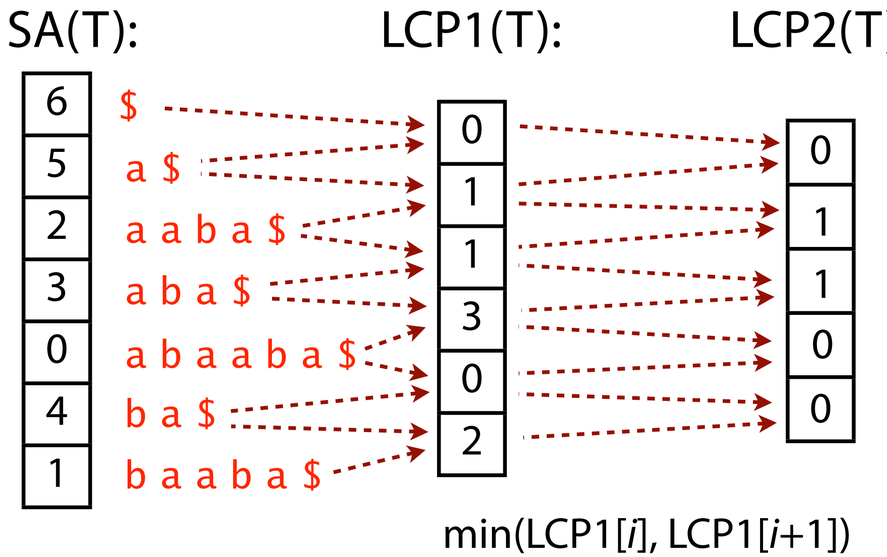
Suffix Array (SA) has *m* elements

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



Suffix array: LCPs

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

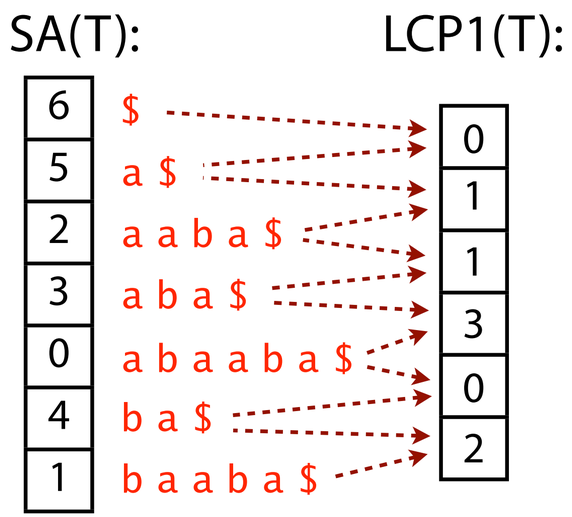


In fact, LCP of a range of consecutive suffixes in SA equals the minimum LCP1 among adjacent pairs in the range

LCP1 is a building block for other useful LCPs

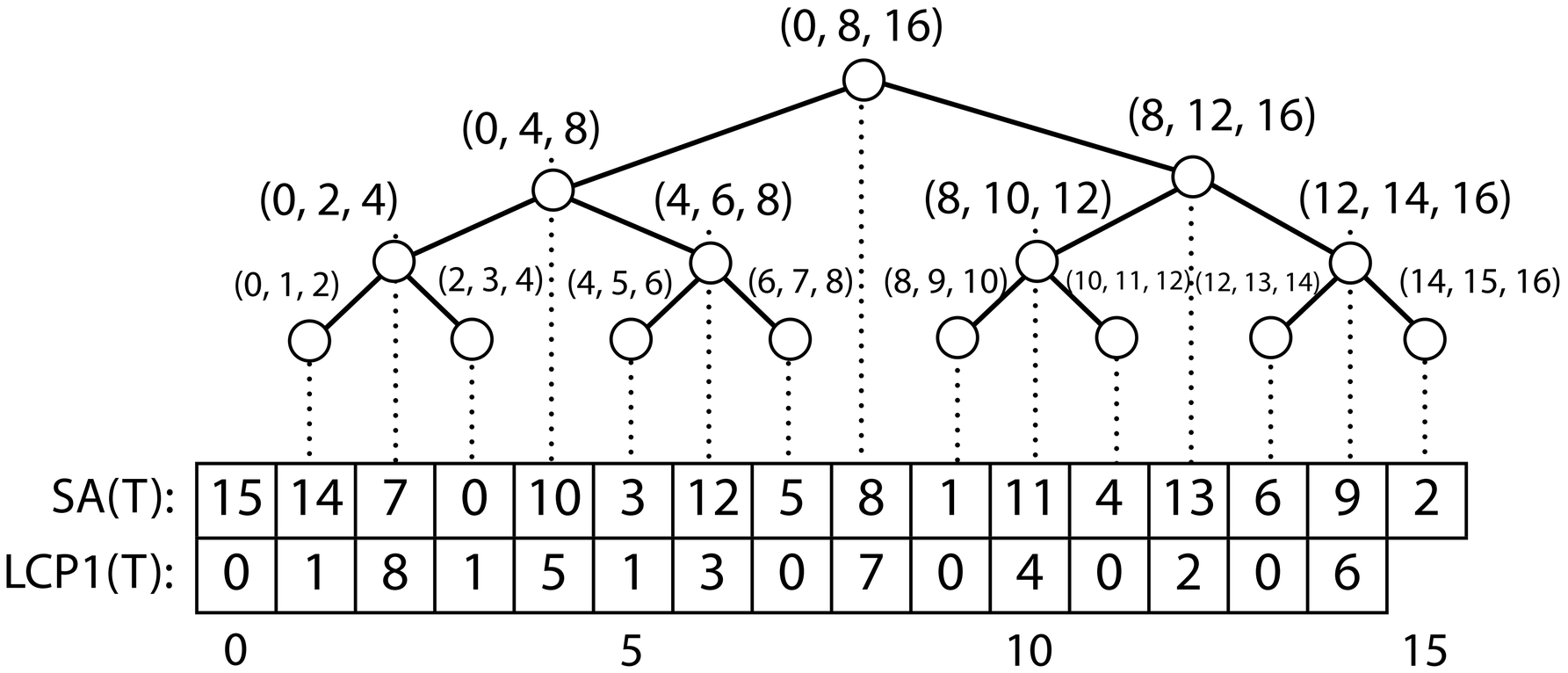
Suffix array: LCPs

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



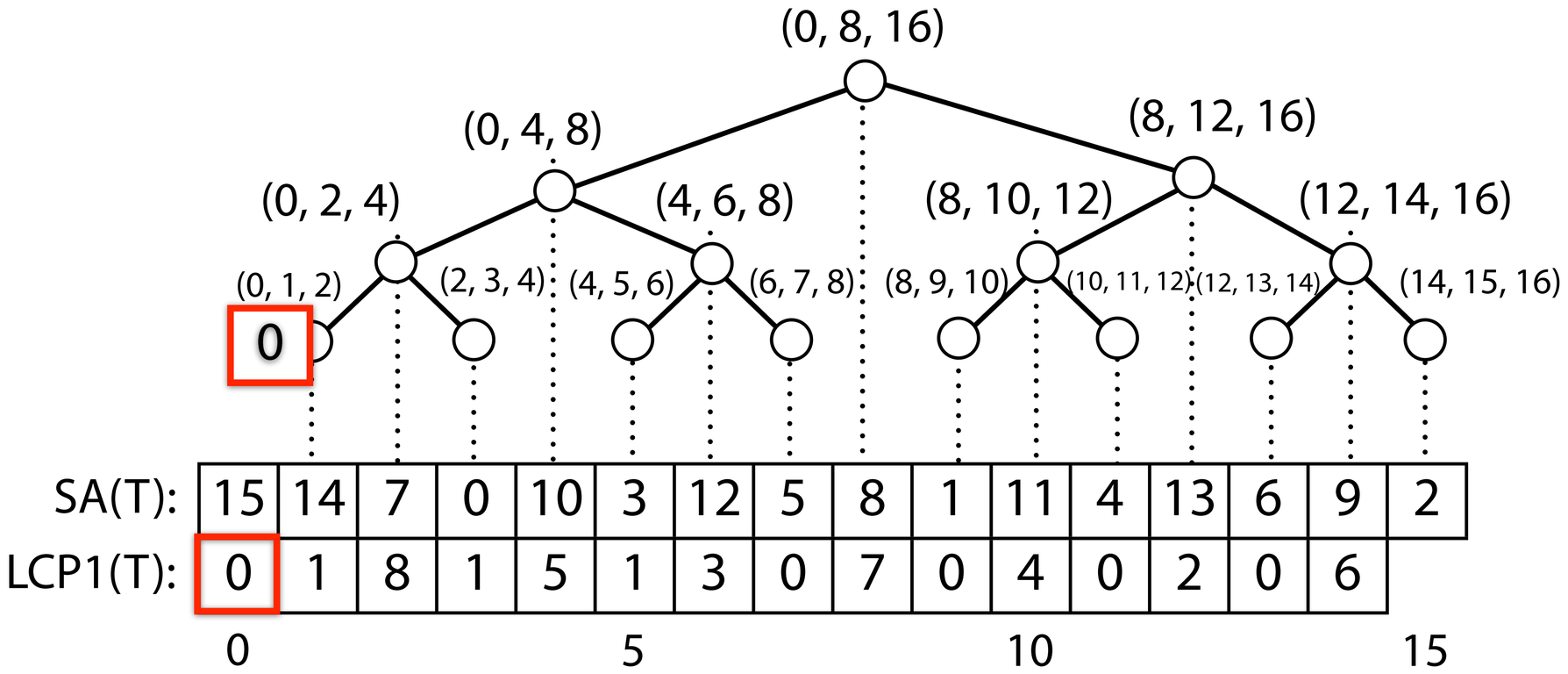
Suffix array: LCPs

T = abracadabracada



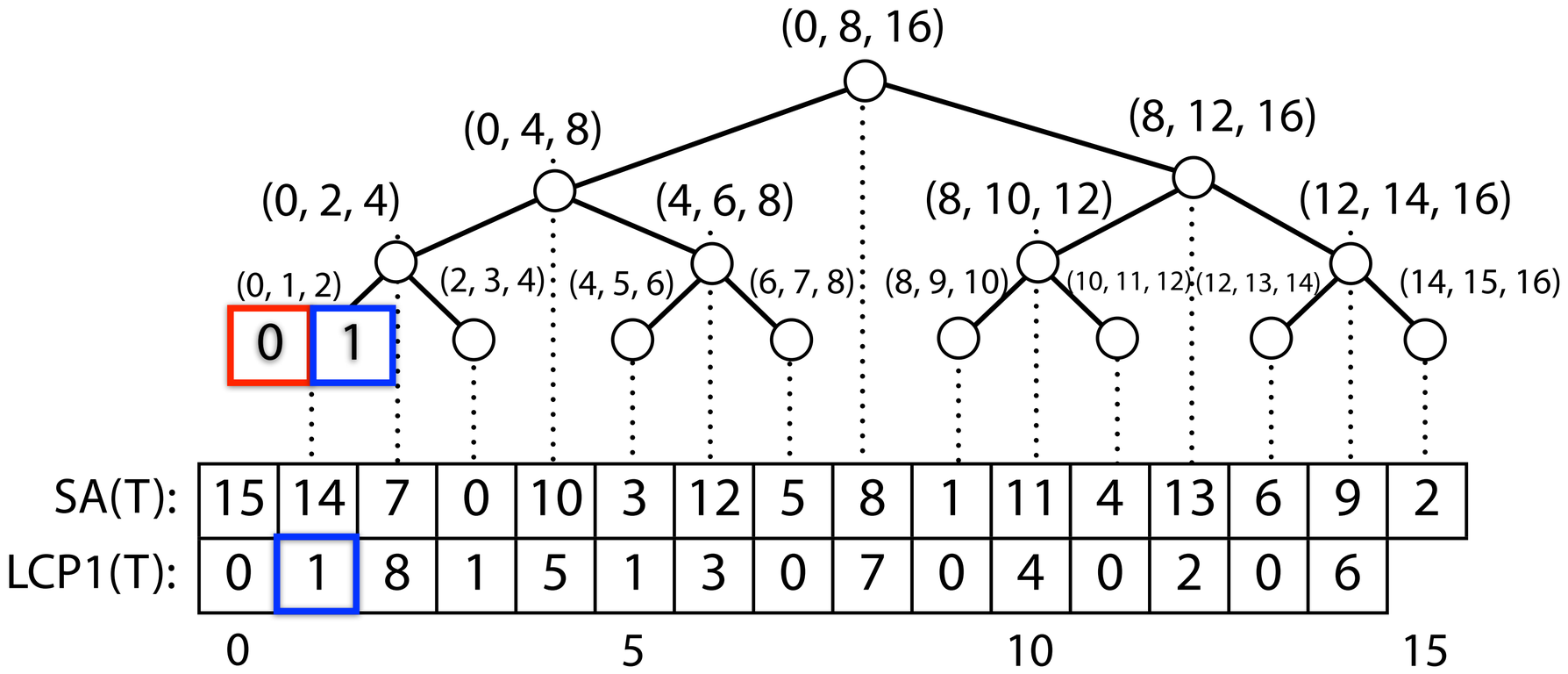
Suffix array: LCPs

T = abracadabracada$



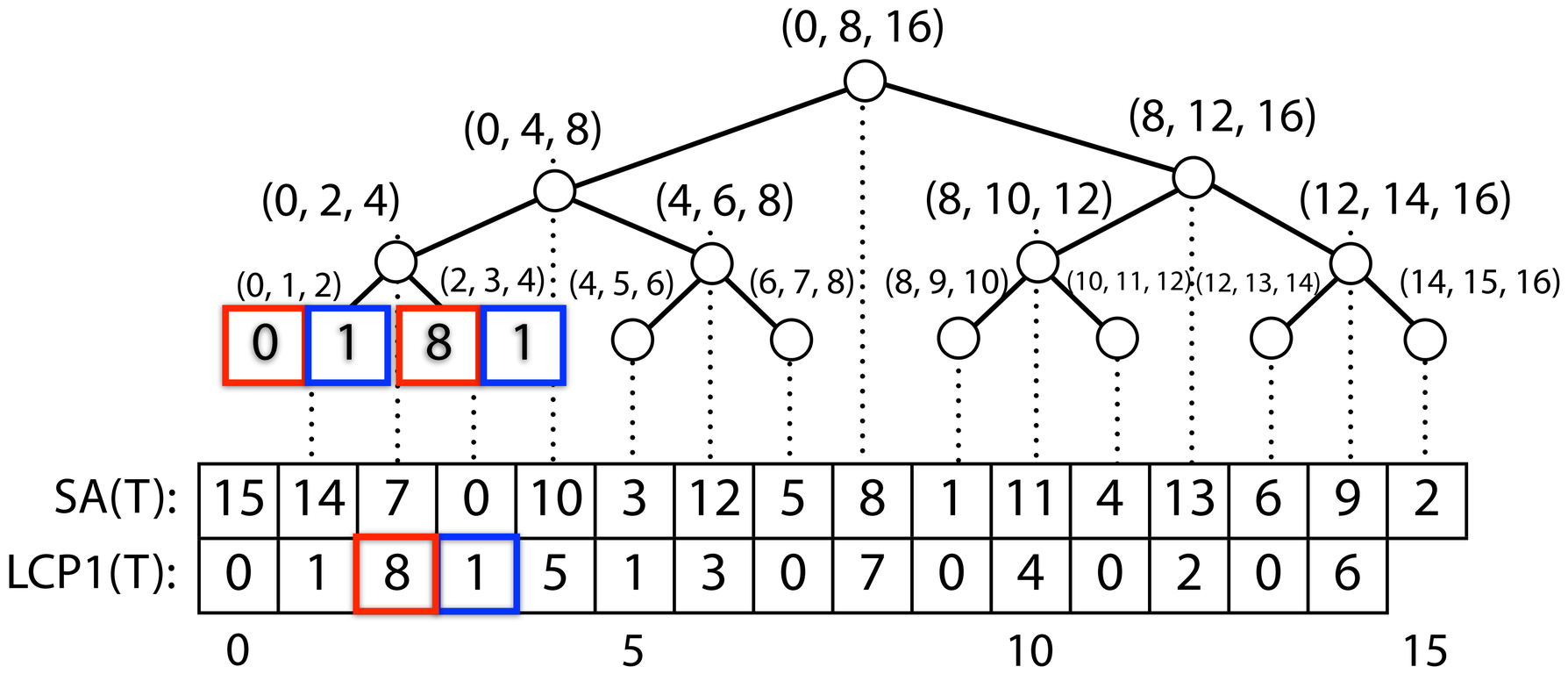
Suffix array: LCPs

T = abracadabracada$



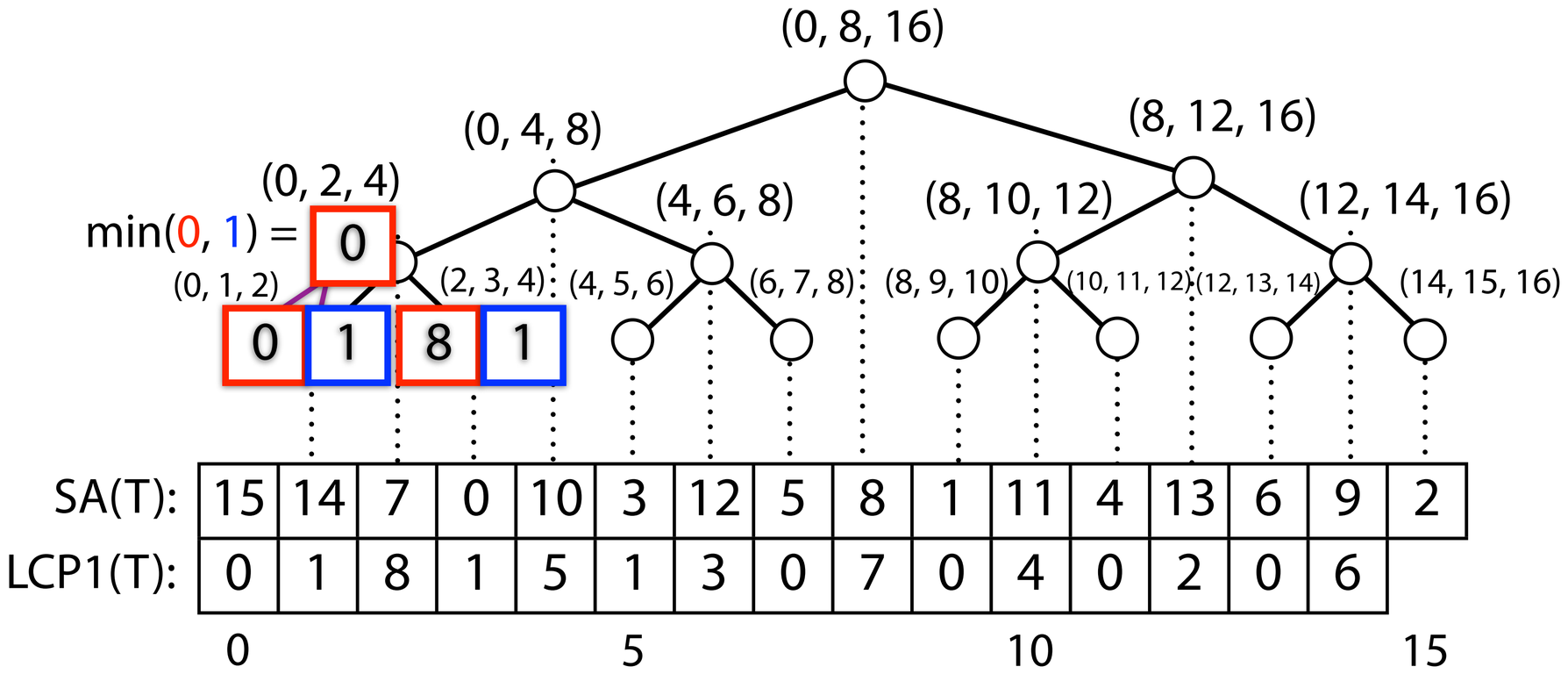
Suffix array: LCPs

T = abracadabracada$



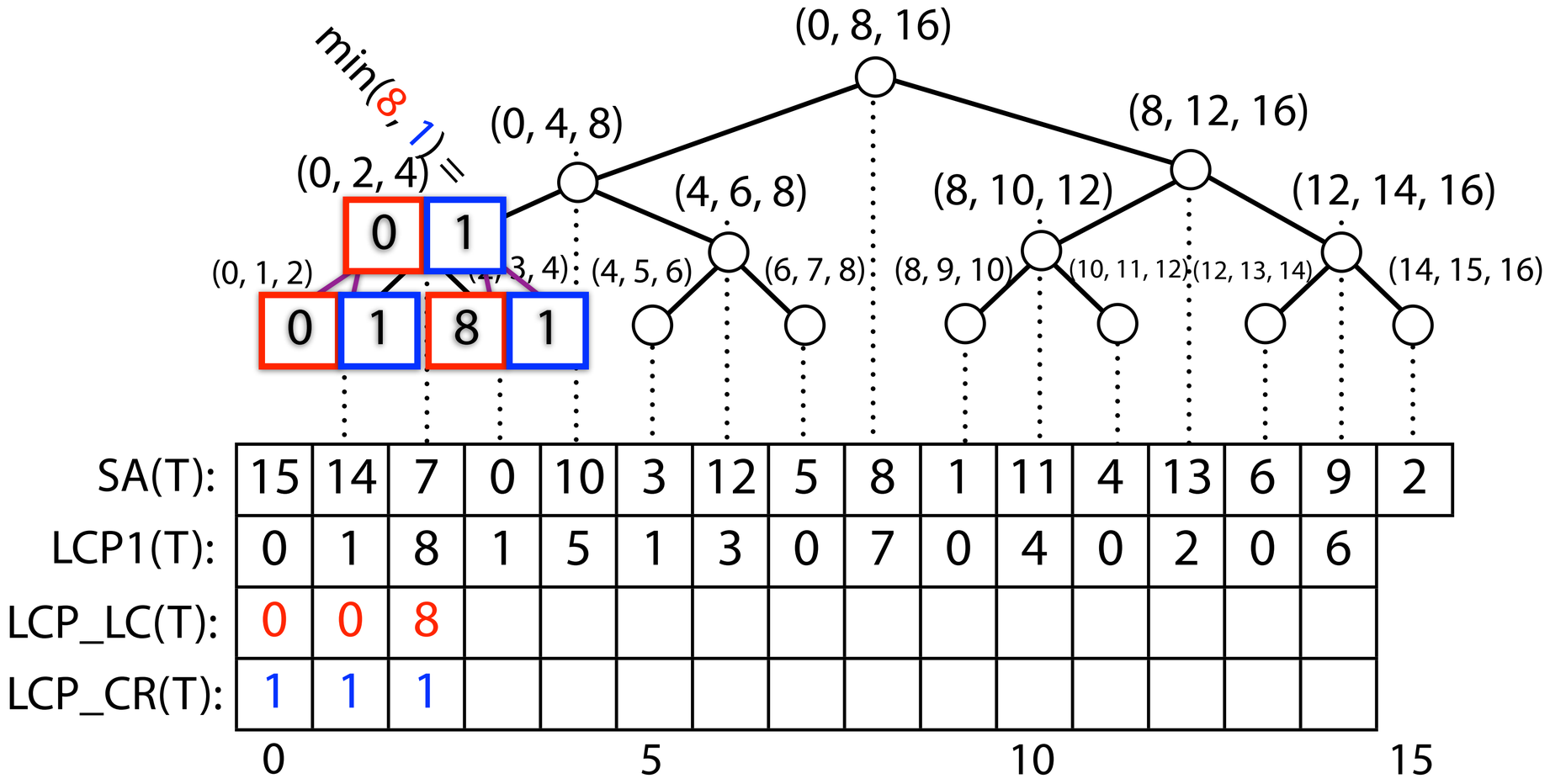
Suffix array: LCPs

T = abracadabracada$



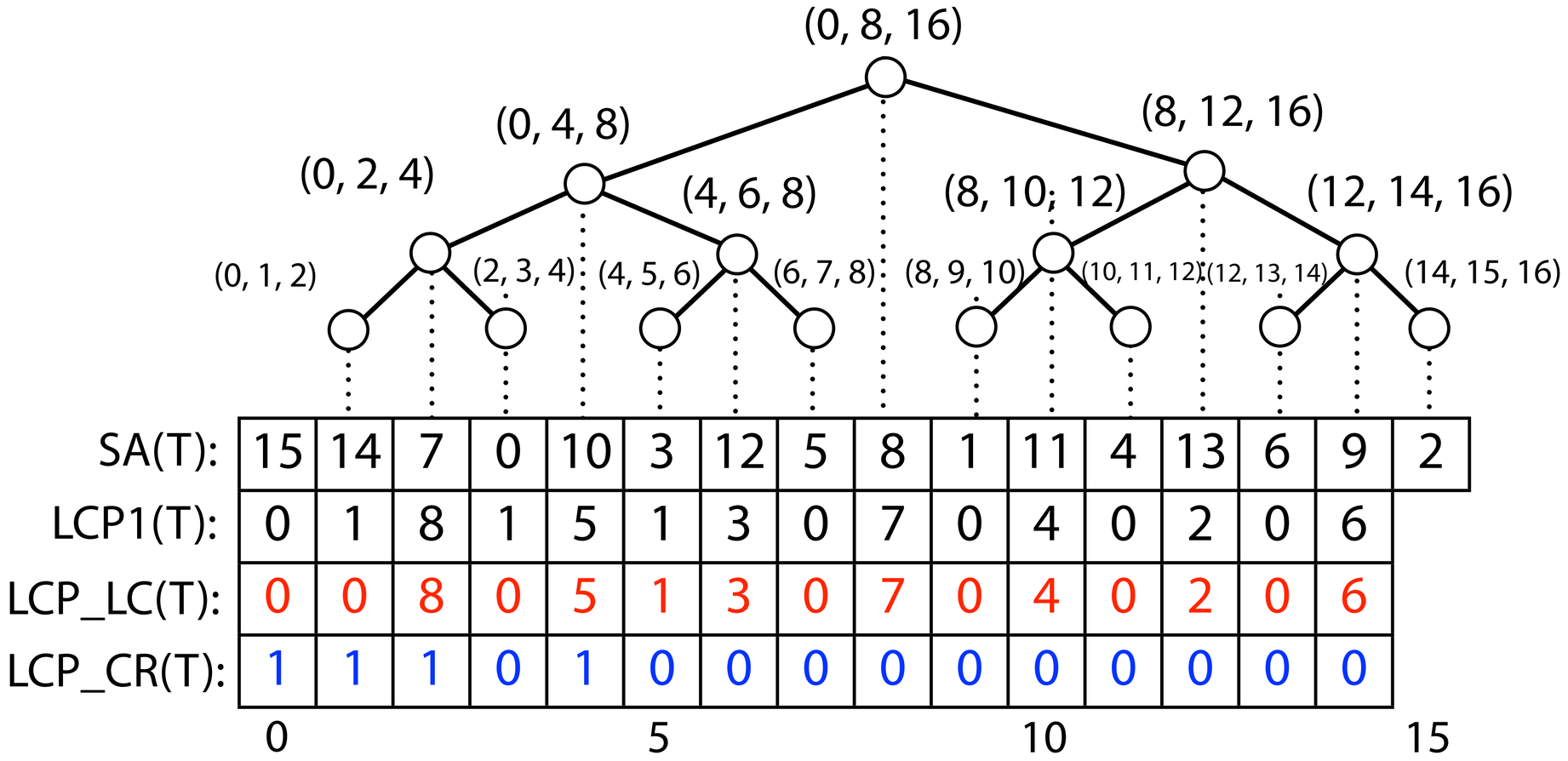
Suffix array: LCPs

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	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

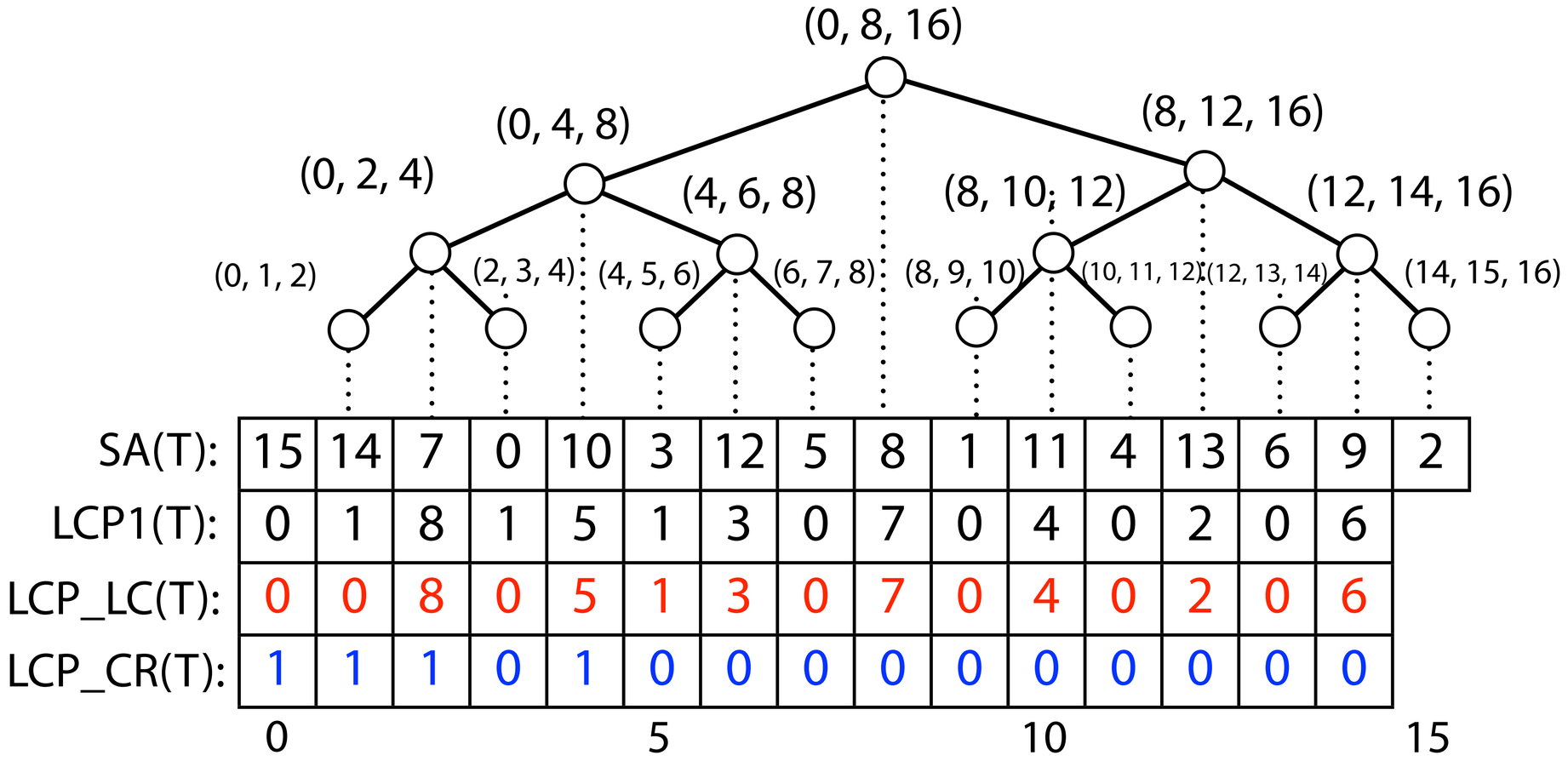
NOTE: These arrays are “shifted” by 1 — the value in LCP_LC corresponding to (0, 1, 2) is at LCP_LC[0], not LCP_LC[1]. So, to look up LCP(SA[I], SA[c]) we look at LCP_LC[c-1]

Suffix array: LCPs

T = abracadabracada$

Can be done in:

O(*m*) time and space



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.
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

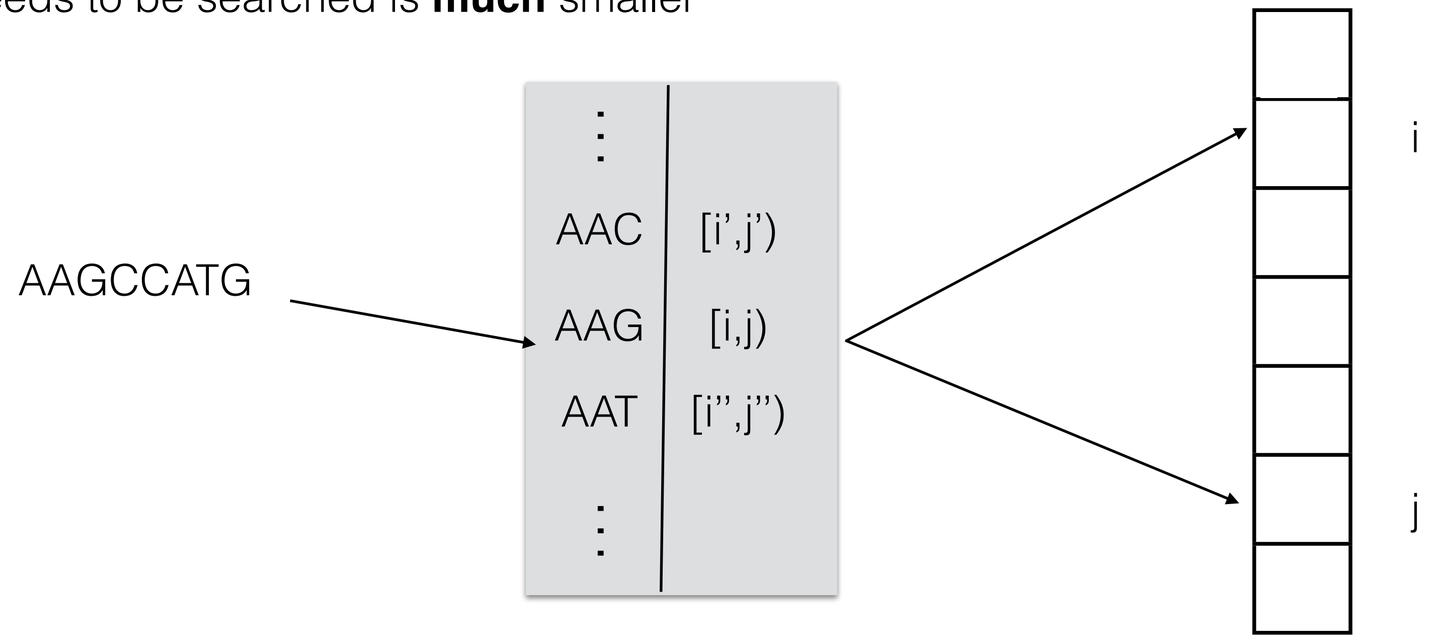
	Super accelerant	Simple accelerant	No accelerant
python -O	68.78 s	69.80 s	102.71 s
pypy -O	5.37 s	5.21 s	8.74 s
# character comparisons	99.5 M	117 M	235 M

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

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 **α**=0.5

	dna	english	proteins	sources	xml
$m = 16$					
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
$m = 64$					
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

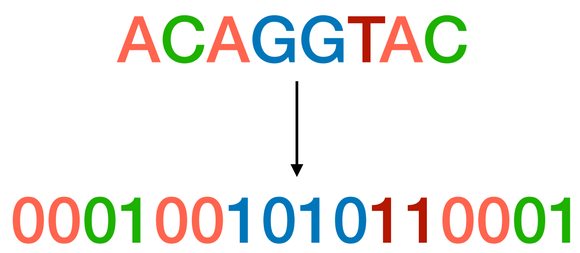
Kowalski, Tomasz, et al. "Suffix arrays with a twist." *arXiv preprint arXiv:1607.08176* (2016).

*

Encoding nucleotide strings as integers

Alphabet of size 4: can encode each nucleotide in 2-bits

Can encode a nucleotide string of length k in 2k bits! (Up to a 32-mer in 1 machine word)



There are actually a lot of interesting choices to make. Should you left-pad or right-pad, how should you encode nucleotides as integers (here we use A:0, C:1, G:2, T:3, but there are other choices that are better in some regards — think about reverse complementation, or speed of encoding).

What if we are allowed to use > 2 bits / char? There are 3-bit encodings that let us compute Hamming distance using word-level operations!

```rust
use hashbrown::HashMap;

fn test_lookup_string(m: &HashMap<String, u32>, q: &[String]) -> u64 {
    let mut res = 0_u64;
    for query in q {
        if let Some(v) = m.get(query) { res += *v as u64; }
    }
    res
}

fn test_lookup_int(m: &HashMap<u64, u32>, q: &[u64]) -> u64 {
    let mut res = 0_u64;
    for query in q {
        if let Some(v) = m.get(query) { res += *v as u64; }
    }
    res
}

// The output is wrapped in a Result to allow matching on errors.
// Returns an Iterator to the Reader of the lines of the file.
fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where P: AsRef<Path>, {
    let file = File::open(filename)?;
    Ok(io::BufReader::new(file).lines())
}
```

```rust
fn main() -> Result<(), std::io::Error> { ▶ Run | Debug
    let mut shash = HashMap::<String, u32>::new();
    let mut skeys = Vec::<String>::new();
    // File hosts.txt must exist in the current path
    if let Ok(lines) = read_lines("string_keys.txt") {
        // Consumes the iterator, returns an (Optional) String
        for (i, line) in lines.flatten().enumerate() {
            shash.insert(line.to_string(), i as u32);
            skeys.push(line.to_string());
        }
    }

    let mut ihash = HashMap::<u64, u32>::new();
    let mut ikeys = Vec::<u64>::new();
    if let Ok(lines) = read_lines("int_keys.txt") {
        // Consumes the iterator, returns an (Optional) String
        for (i, line) in lines.flatten().enumerate() {
            let n = line.parse().unwrap();
            ihash.insert(n, i as u32);
            ikeys.push(n);
        }
    }

    let mut rng = rand::thread_rng();
    skeys.shuffle(&mut rng);
    ikeys.shuffle(&mut rng);

    use std::time::Instant;
    let now = Instant::now();
    let mut sr = 0;
    for _ in 0..10 {
        sr += test_lookup_string(&shash, &skeys);
    }
    println!("string map took : {}, res : {sr}", now.elapsed().as_millis());

    let now = Instant::now();
    let mut sr = 0;
    for _ in 0..10 {
        sr += test_lookup_int(&ihash, &ikeys);
    }
    println!("int map took : {}, res : {sr}", now.elapsed().as_millis());


    Ok(())
}
```

```rust
use hashbrown::HashMap;

fn test_lookup_string(m: &HashMap<String, u32>, q: &[String]) -> u64 {
    let mut res = 0_u64;
    for query in q {
        if let Some(v) = m.get(query) { res += *v as u64; }
    }
    res
}

fn test_lookup_int(m: &HashMap<u64, u32>, q: &[u64]) -> u64 {
    let mut res = 0_u64;
    for query in q {
        if let Some(v) = m.get(query) { res += *v as u64; }
    }
    res
}

// The output is wrapped in a Result to allow matching on errors.
// Returns an Iterator to the Reader of the lines of the file.
fn read_lines<P>(filename: P) -> io::Result<io::Lines<io::BufReader<File>>>
where P: AsRef<Path>, {
    let file = File::open(filename)?;
    Ok(io::BufReader::new(file).lines())
}
```

```
hash_test on  master [?] is 📦 v0.1.0 via 🦀 v1.75.0
✦ ❯ cargo run --release
    Finished release [optimized] target(s) in 0.50s
     Running `target/release/hash_test`
string map took : 2526, res : 4999995000000
int map took : 206, res : 4999995000000
```

Lookup for the encoded 31-mers is about an order of magnitude faster than for the strings!

```rust
fn main() -> Result<(), std::io::Error> { ▶ Run | Debug
    let mut shash = HashMap::<String, u32>::new();
    let mut skeys = Vec::<String>::new();
    // File hosts.txt must exist in the current path
    if let Ok(lines) = read_lines("string_keys.txt") {
        // Consumes the iterator, returns an (Optional) String
        for (i, line) in lines.flatten().enumerate() {
            shash.insert(line.to_string(), i as u32);
            skeys.push(line.to_string());
        }
    }

    let mut ihash = HashMap::<u64, u32>::new();
    let mut ikeys = Vec::<u64>::new();
    if let Ok(lines) = read_lines("int_keys.txt") {
        // Consumes the iterator, returns an (Optional) String
        for (i, line) in lines.flatten().enumerate() {
            let n = line.parse().unwrap();
            ihash.insert(n, i as u32);
            ikeys.push(n);
        }
    }

    let mut rng = rand::thread_rng();
    skeys.shuffle(&mut rng);
    ikeys.shuffle(&mut rng);

    use std::time::Instant;
    let now = Instant::now();
    let mut sr = 0;
    for _ in 0..10 {
        sr += test_lookup_string(&shash, &skeys);
    }
    println!("string map took : {}, res : {sr}", now.elapsed().as_millis());

    let now = Instant::now();
    let mut sr = 0;
    for _ in 0..10 {
        sr += test_lookup_int(&ihash, &ikeys);
    }
    println!("int map took : {}, res : {sr}", now.elapsed().as_millis());


    Ok(())
}
```

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

The Skew Algorithm (aka DC3)

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.

$$T[0,n) = \begin{array}{cccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \texttt{y} & \texttt{a} & \texttt{b} & \texttt{b} & \texttt{a} & \texttt{d} & \texttt{a} & \texttt{b} & \texttt{b} & \texttt{a} & \texttt{d} & \texttt{o} \end{array}$$

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

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.

*slide courtesy of Carl Kingsford

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., $C = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1 — Sorting the sample

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

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

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: $[t_i t_{i+1} t_{i+2}][t_{i+3} t_{i+4} t_{i+5}] \ldots$ 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. $R =$ [abb][ada][bba][do0][bba][dab][bad][o00].

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1.5 — Sorting the sample

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

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

Note: In this example, we won't be able to get all of these ranks until the algorithm has recursed a second time.

R' is ranks after first round / recursion.

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

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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.

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

1 mod 3 triplets | 2 mod 3 triplets

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

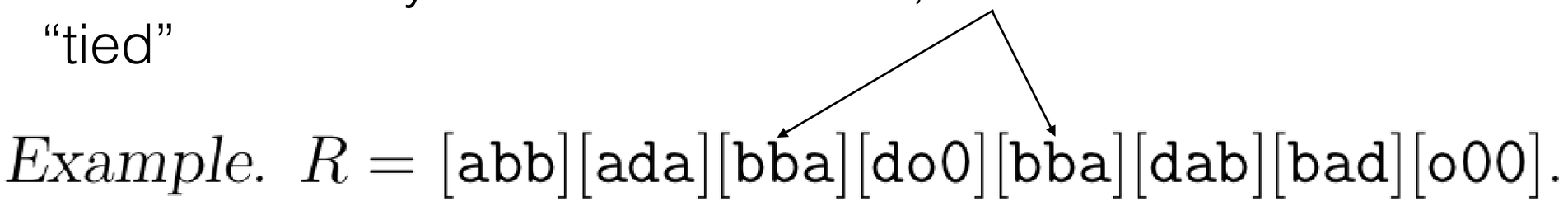
i	0	1	2	3	4	5	6	7	8	9	10	11	12
Eg.rank(Si)	⊥	1	4	⊥	2	5	⊥	4	3	⊥	6	7	⊥

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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.

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

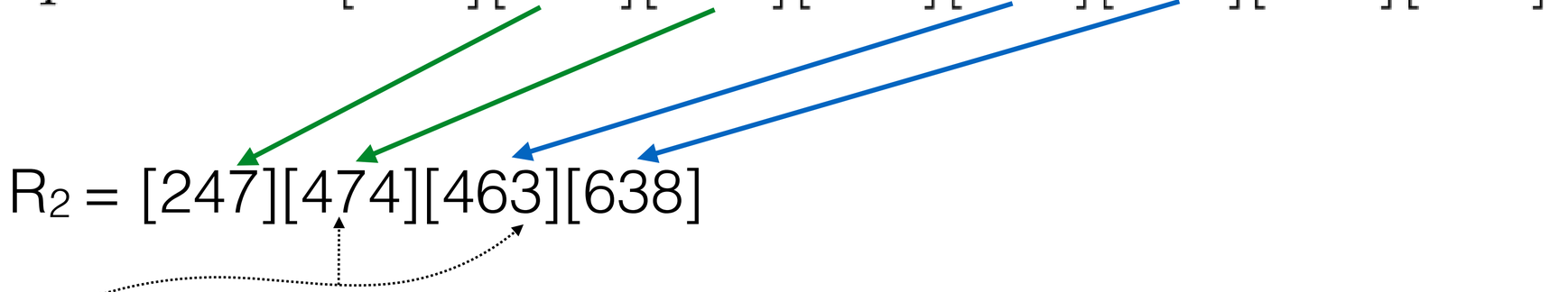


Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

Step 1.5 — Sorting the sample

$$T[0,n) = \begin{array}{cccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ \mathtt{y} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{o} \end{array}$$

$$\textit{Example. } R = \overset{1}{[\mathtt{abb}]}\overset{2}{[\mathtt{ada}]}\overset{4}{[\mathtt{bba}]}\overset{7}{[\mathtt{do0}]}\overset{4}{[\mathtt{bba}]}\overset{6}{[\mathtt{dab}]}\overset{3}{[\mathtt{bad}]}\overset{8}{[\mathtt{o00}]}.$$



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.

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

The ranks for original B_1 B_2 samples

	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

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

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 $(t_i, rank(S_{i+1}))$. Note that $rank(S_{i+1})$ is always defined for $i \in B_0$. Clearly we have, for all $i, j \in B_0$,

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

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

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

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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:

$$i \in B_1 : \quad S_i \leq S_j \iff (t_i, rank(S_{i+1})) \leq (t_j, rank(S_{j+1}))$$

$$i \in B_2 : \quad S_i \leq S_j \iff (t_i, t_{i+1}, rank(S_{i+2})) \leq (t_j, t_{j+1}, rank(S_{j+2}))$$

S_{B1} S_{B2} S_{B0} S_{B1}

S_{B2} S_{B0} S_{B1} S_{B0} S_{B1} S_{B2}

Note that the ranks are defined in all cases.

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

Taken from: Kärkkäinen, J., Sanders, P., & Burkhardt, S. (2006). Linear work suffix array construction. Journal of the ACM (JACM), 53(6), 918-936.

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 $i < n$.
- $T(n) \leq cn + 3c(2n/3) = cn + 2cn = 3cn$ □

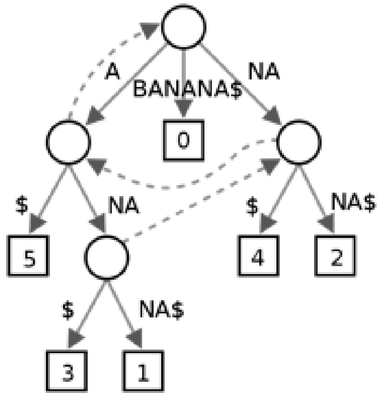
*slide courtesy of Carl Kingsford

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

Enhanced Suffix Arrays

Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch. **"Replacing suffix trees with enhanced suffix arrays."** Journal of Discrete Algorithms 2.1 (2004): 53-86.

Can restore the **full** asymptotic efficiency of suffix trees with a small number of auxiliary tables.

Application	Enhanced suffix array					
	suftab $4n$ bytes	lcptab n bytes	childtab n bytes	suflink $2n$ bytes	S $n \log \lvert\Sigma\rvert$ bits	bwttab $n \log \lvert\Sigma\rvert$ bits
esasupermax	√	√				√
esamum	√	√				√
esarep	√	√				√
Ziv–Lempel	√	√				
esamatch	√	√	√		√	
shortest unique sub.	√	√	√			
esams	√	√	√	√	√	

The operations that can be done optimally in an enhanced suffix array (esa), and the aux. tables required for them.

*

Many Suffix Array Variants

Compressed suffix arrays[+] — require even less space

Compressed enhanced suffix arrays[✢] — strive for the best of both worlds and allow interesting query times like O(n log|Σ|+k) for finding k occurrences of a pattern, where |Σ| is the size of the alphabet (independent of m).

+R. Grossi and J. S. Vitter, Compressed Suffix Arrays and Suffix Trees, with Applications to Text Indexing and String Matching, SIAM Journal on Computing, 35(2), 2005, 378-407

✢Ohlebusch, Enno, and Simon Gog. "A compressed enhanced suffix array supporting fast string matching." String Processing and Information Retrieval. Springer Berlin Heidelberg, 2009.

*