SUFFIX ARRAYS
& CS BASICS

Asymptotic Analysis

- Big O notation "Big oh"
  - aka Landau notations / Bachmann-Landau notation
  - Concerned with the "order" of the function (i.e. how fast or slow do the dominant terms grow)
  - Concerned with the growth as the problem size $\to \infty$
    - not concerned with behavior on small fixed values of the input
  - Measures the growth (usually in time or memory) as a function of the size of the input to the procedure or algorithm.

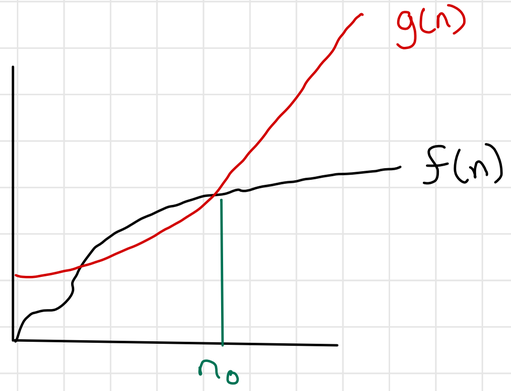
Asymptotic Analysis

Definition: We say that $f(n) \in O(g(n))$ or $f(n) = O(g(n))$ if, as $n \to \infty$, $\exists\ c \in \mathbb{R}^+$, n_0 such that

$$\forall\ n \geq n_0, \quad f(n) \leq c \cdot g(n)$$

In words, there is some constant c and smallest n_0 such that for all $n \geq n_0$, $f(n)$ is no larger than $c \cdot g(n)$.

here's a picture



Asymptotic Analysis

Let's look at some Examples

- $f(n) = 4n + 8$
- $g(n) = n$

→ does this work? Yes, let $c = 5$ $n_0 = 8$

$f(n_0) = 4 \cdot 8 + 8 = 40$, $g(n_0) = 5 \cdot 8 = 40$

and $\forall\ n > n_0$ $5 \cdot n > 4n + 8$ because

$5n = 4n + n > 4n + 8$ because $n > 8$ ✓

→ what about $g(n) = n^2$? Yes, let $c = 1$, $n_0 = 6$

$f(n_0) = 4 \cdot 6 + 8 = 32$, $g(n_0) = 6^2 = 36$

$\forall\ n > n_0$ $n^2 > 4n + 8$ because $\frac{n^2}{n} = n > \frac{4n}{n} + \frac{8}{n}$ ✓

Asymptotic Analysis

- So, we can find many $g(n)$ such that $f(n) = O(g(n))$.
  - → in general, we are interested in the smallest
  - → some rules of thumb; with polynomials, it is easy

    $f(n) = a \cdot n + b$ for constant a, b is $O(n)$

    $f(n) = a \cdot n^2 + bn + c$ for constant a, b, c is $O(n^2)$

    etc. if we take the highest-order term in $f(n)$, say n^p, then

    $f(n) = O(n^p)$

- Lower order terms "don't matter" because we can choose our c, n_0 in the definition.

Asymptotic Analysis

- Some common "complexities"

  $O(1)$ - algorithm completes in same order of time, independent of input

  - what might take this long?

  $O(\log(n))$ - " " in order of time proportional to log of input size

  - what might take this long?

  $O(n)$ - " " in order of time proportional to input size

  - what might take this long?

  $O(n^2)$ - " "

  $O(n^3)$ - " "

  } What are potential examples of these?

Basic Search

Consider a list of numbers:

Does the number 42 occur in this list?

How many “steps” will it take to search this list for 42?

How many values will we have to look at *in the worst case*, before we either find 42 or can **guarantee** it doesn’t exist?

Basic Search

```python
def find_first_occurrence(query, l):
    '''
    search the list `l` from the start until we find the first
    occurrence of `query`. If we find `query`, return the offset
    of the first occurrence. Otherwise, return -1
    '''
    for i, e in enumerate(l):
        if e == query:
            return i
    return -1
```

Basic Search

- In our Big-O notation, this algorithm is O(n)
- In fact, it scales exactly with n — looking at n items in the worst case
- Interestingly, without some more ***structure***, it will be hard to do better than this!

Basic Search of a Sorted List

Consider our list from before, except assume that I give you the values *in order*:

Now, how do the answers to the following questions change:

Does the number 42 occur in this list?

How many “steps” will it take to search this list for 42?

How many values will we have to look at *in the worst case*, before we either find 42 or can **guarantee** it doesn’t exist?

Can we do better than O(n)?

Binary Search

Let’s take advantage of the structure of this list.

Does 42 occur in this list?

How do we know?

Binary Search

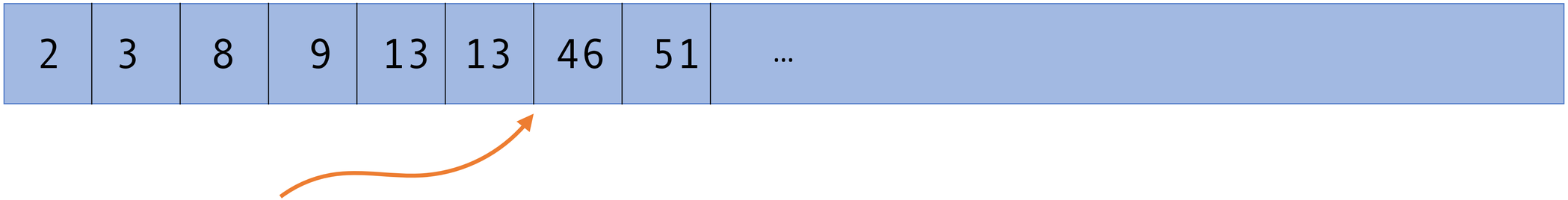
Let’s take advantage of the structure of this list.

If 42 *did* occur in this list, where would it have to be?

Binary Search

Let’s take advantage of the structure of this list.

If 42 *did* occur in this list, where would it have to be?

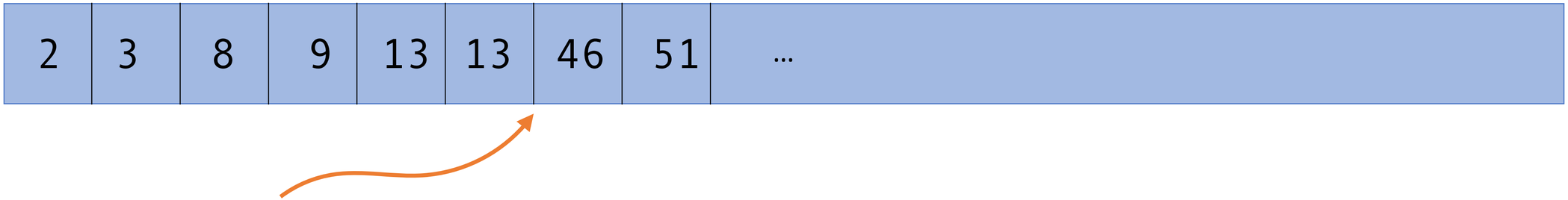


Right here — why?

Binary Search

Let’s take advantage of the structure of this list.

If 42 *did* occur in this list, where would it have to be?



Right here — why?

Otherwise; the list wouldn’t be in order! **How can we use this principle in general?**

Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

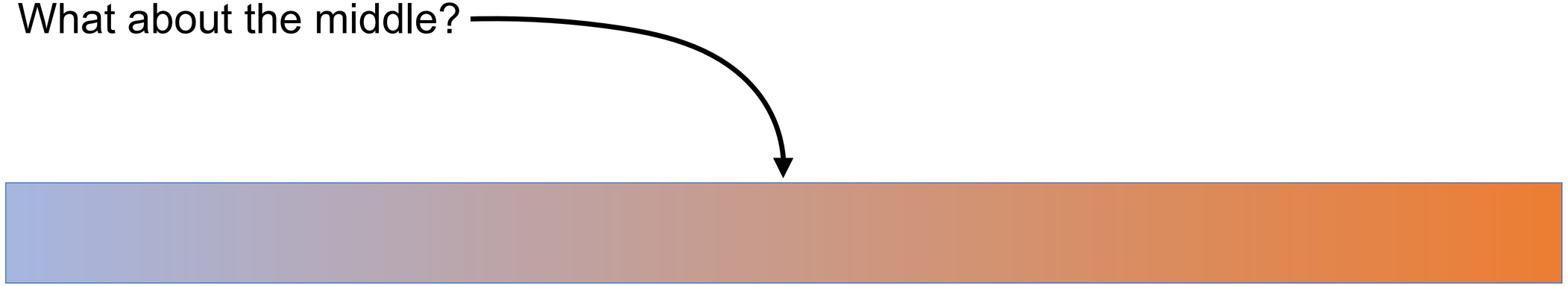
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



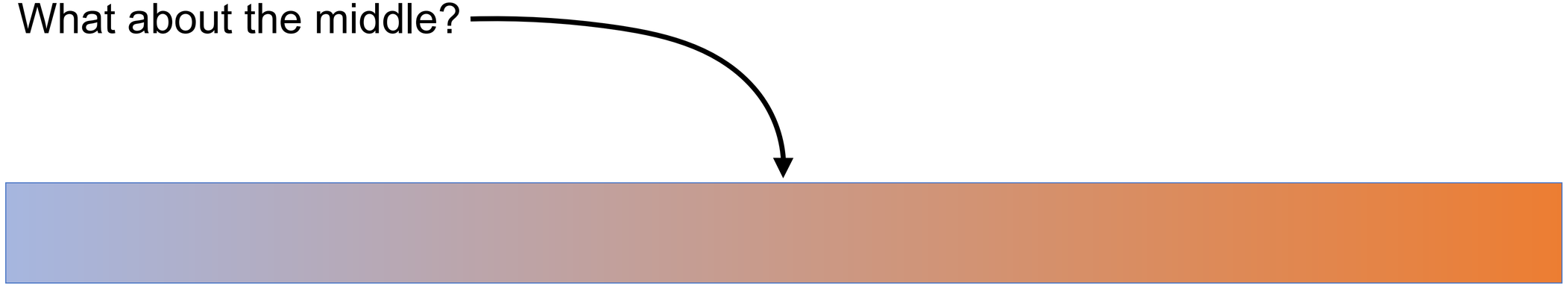
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Call the element we find there **m**. How might **m** relate to **q**?

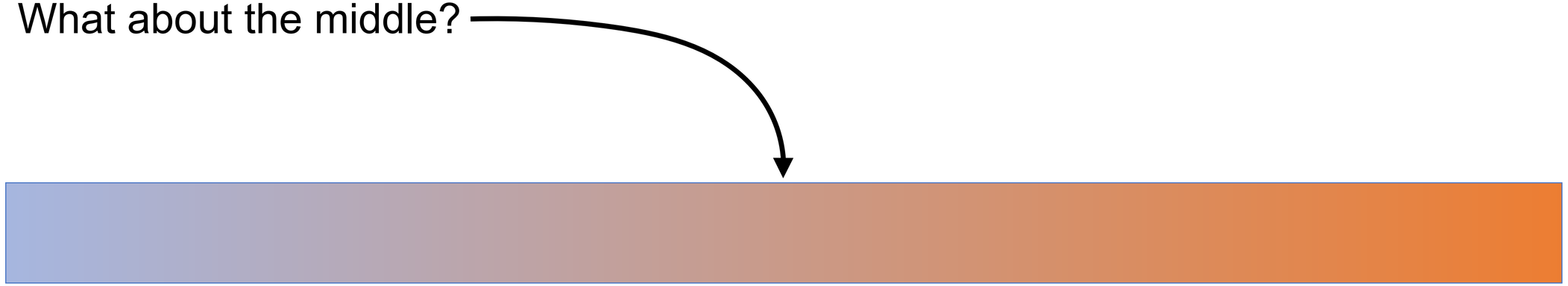
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Call the element we find there **m**. How might **m** relate to **q**?

m < q, m = q, m > q

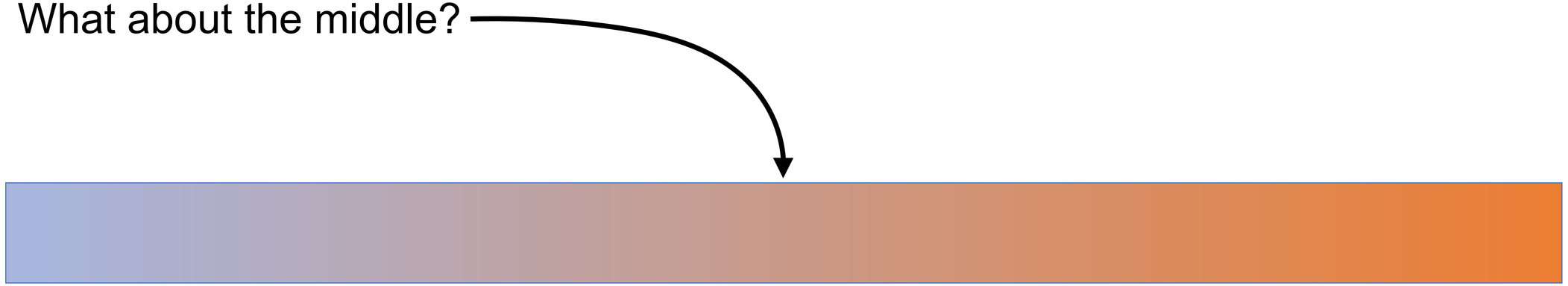
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Call the element we find there **m**. How might **m** relate to **q**?

m < q, m = q, m > q

Yea; then we're done!

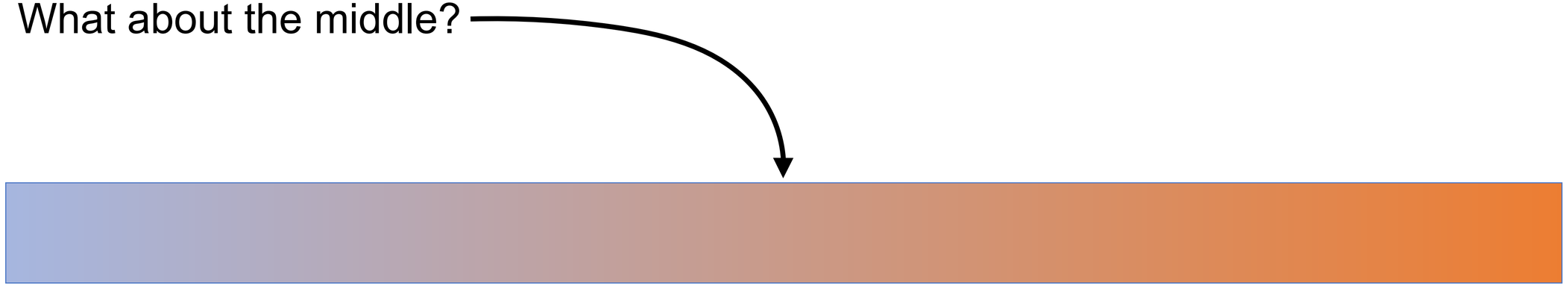
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Call the element we find there **m**. How might **m** relate to **q**?

m < q, m = q, m > q

Ok; then where do we look next for q?

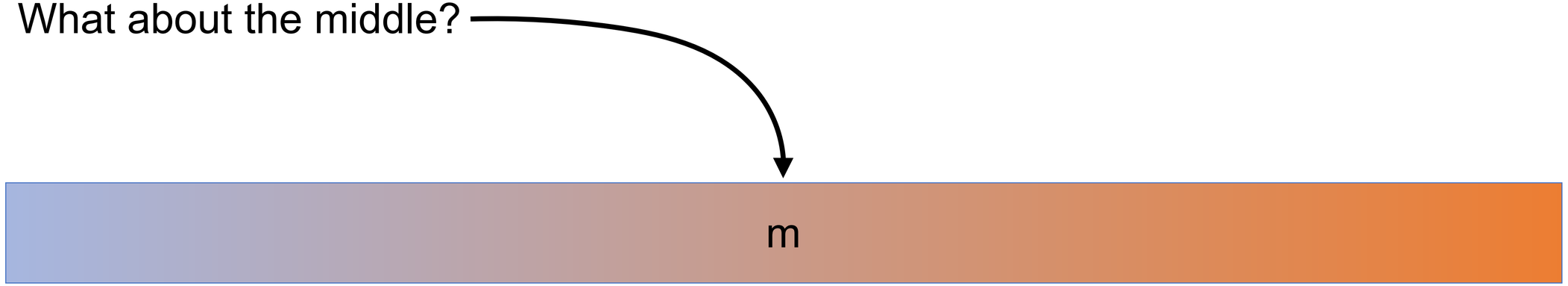
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Consider: m < q

If q exists in our list, where must it "live"?

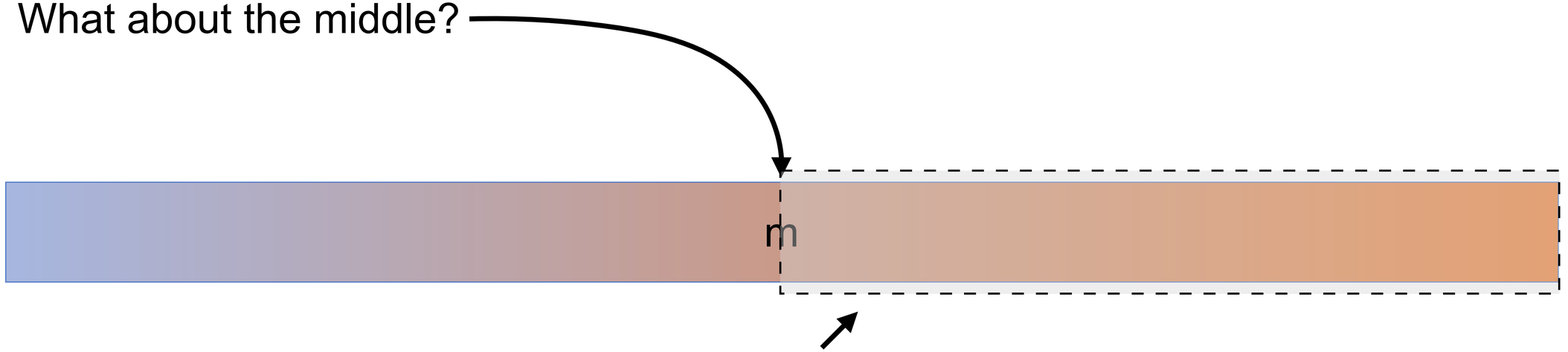
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Consider: m < q

If q exists in our list, it must "live" in the half > m!

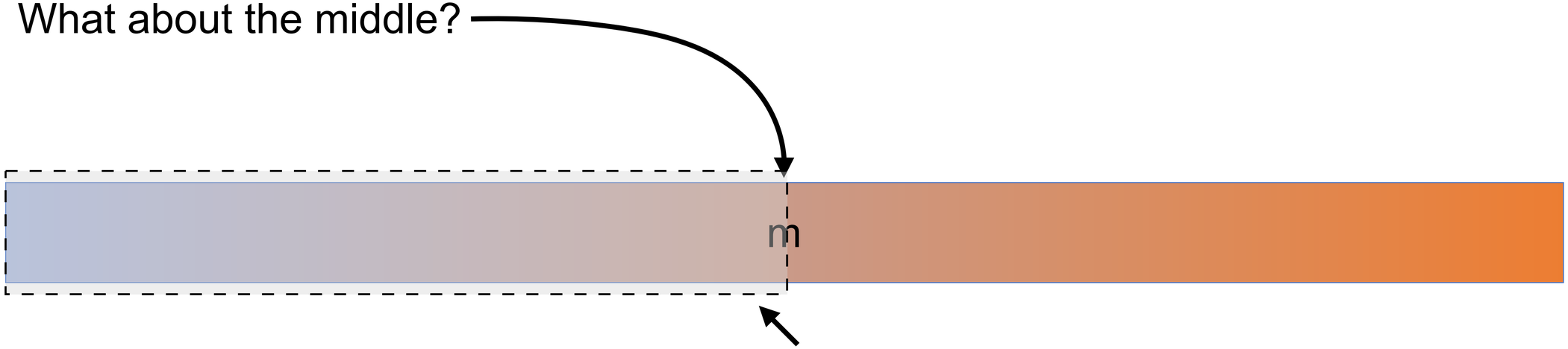
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

We want to find out where **q** occurs in our list. Where should we look first?

What about the middle?



Consider: m > q

If q exists in our list, it must "live" in the half < m!

Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

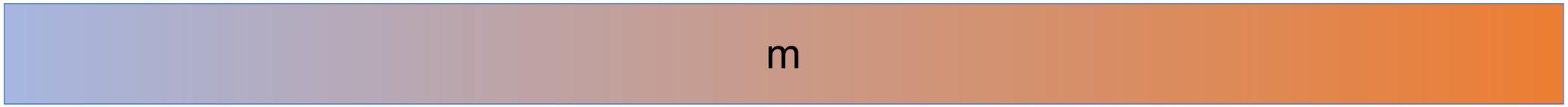
Imagine we want to search for some query value **q**

So we have a plan of action!

If m < q, search in the second half of the list

If m = q, we found it (forget for a minute, having to find the first occurrence)

If m > q, search in the first half of the list



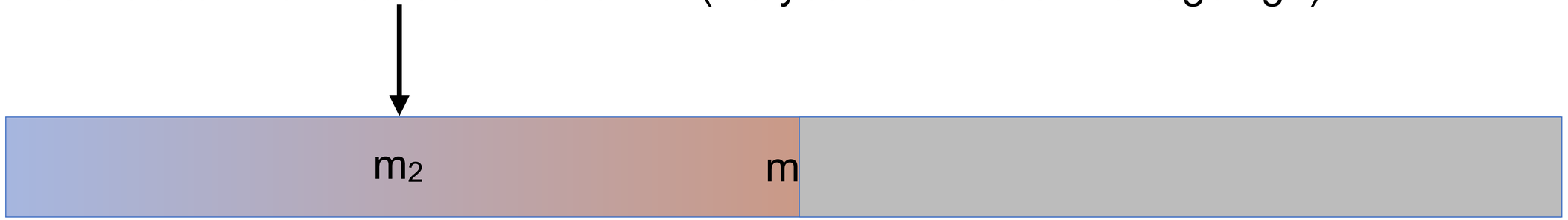
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

Imagine we want to search for some query value **q**

Consider m > q, we search in the first half of our list; where do we look?

How about the middle of this half? (Do you see where this is going?)



Call the element we find there $\mathbf{m_2}$. How might $\mathbf{m_2}$ relate to **q**?

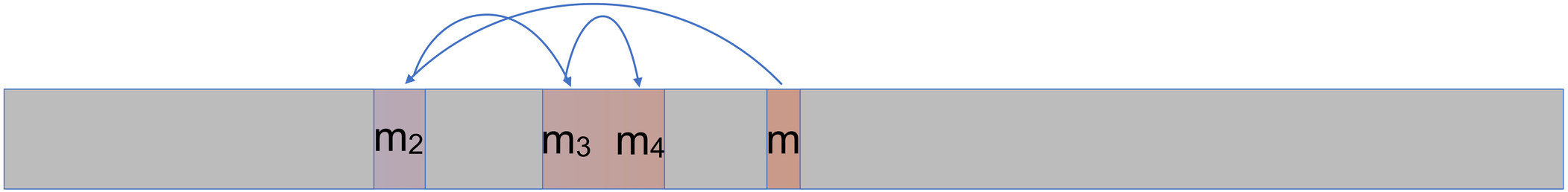
$m_2 < q$, $m_2 = q$, $m_2 > q$

Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

General approach — compare **q** to the middle of the *remaining feasible interval* **m'**

If q = m' (done), if q < m' (search left half), if q > m' (search right half)



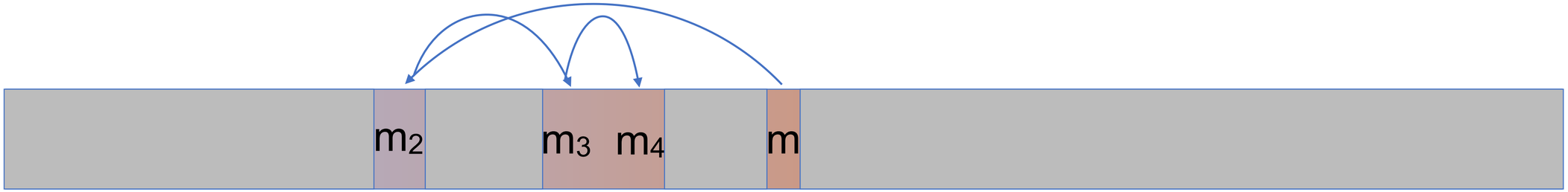
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

General approach — compare **q** to the middle of the *remaining feasible interval* **m'**

If q = m' (done), if q < m' (search left half), if q > m' (search right half)

How many steps can this take in the worst case?



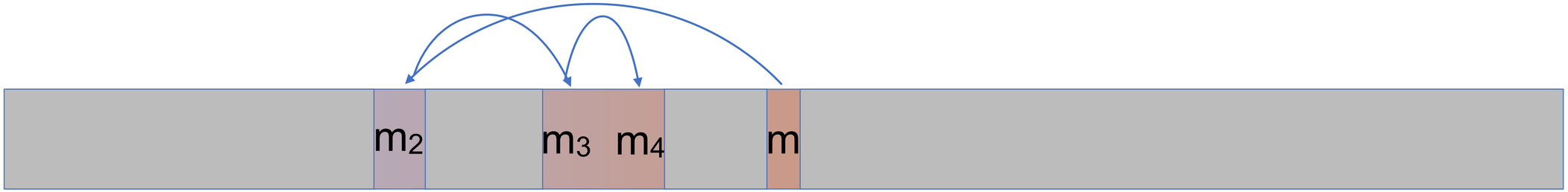
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

General approach — compare **q** to the middle of the *remaining feasible interval* **m'**

If q = m' (done), if q < m' (search left half), if q > m' (search right half)

How many steps can this take in the worst case?



At worst, we divide the remaining list in half each time. If we have a list of length N, now many times can we divide it in half before it's empty?

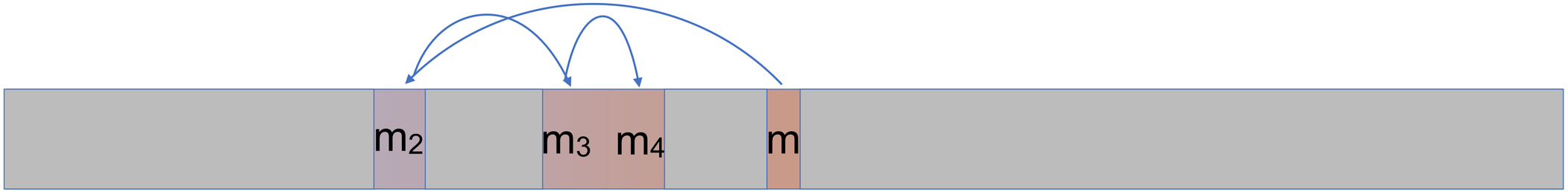
Binary Search

Consider a sorted list of integers (let's just assume positive, but it doesn't matter):

General approach — compare **q** to the middle of the *remaining feasible interval* **m'**

If q = m' (done), if q < m' (search left half), if q > m' (search right half)

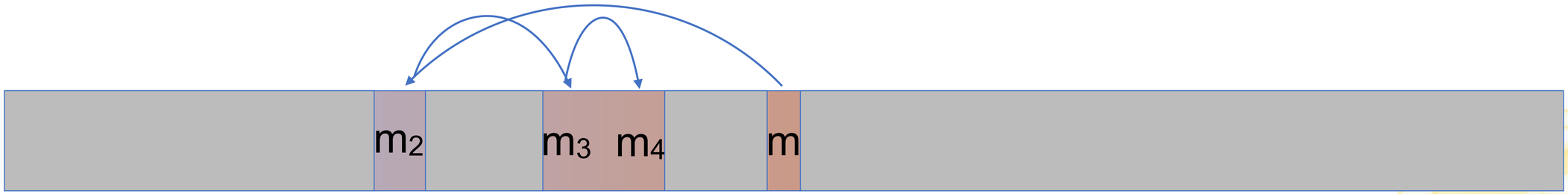
How many steps can this take in the worst case?



At worst, we divide the remaining list in half each time. If we have a list of length N, now many times can we divide it in half before it's empty? **$\log_2(N)$**

Binary Search

```python
def find_occurrence_binary_search(query, l):
    lower_bound = 0
    upper_bound = len(l) - 1

    while lower_bound <= upper_bound:

        midpoint = (upper_bound + lower_bound) // 2

        if query < l[midpoint]:
            upper_bound = midpoint - 1

        elif query > l[midpoint]:
            lower_bound = midpoint + 1

        else:
            return midpoint

    return -1
```



Binary Search

If this seems like a useful primitive, that's because it is! In python, you'll find this in the bisect module.

`bisect.`**`bisect_left`**`(a, x, lo=0, hi=len(a), *, key=None)`

Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem < x for elem in a[lo : ip])` is true for the left slice and `all(elem >= x for elem in a[ip : hi])` is true for the right slice.

key specifies a key function of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

Changed in version 3.10: Added the *key* parameter.

Binary Search

If this seems like a useful primitive, that's because it is! In python, you'll find this in the bisect module.

`bisect.`**`bisect_left`**`(a, x, lo=0, hi=len(a), *, key=None)`

Locate the insertion point for *x* in *a* to maintain sorted order. The parameters *lo* and *hi* may be used to specify a subset of the list which should be considered; by default the entire list is used. If *x* is already present in *a*, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to `list.insert()` assuming that *a* is already sorted.

The returned insertion point *ip* partitions the array *a* into two slices such that `all(elem < x for elem in a[lo : ip])` is true for the left slice and `all(elem >= x for elem in a[ip : hi])` is true for the right slice.

key specifies a key function of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

Changed in version 3.10: Added the *key* parameter.

Even better than our example version, returns the leftmost position where x could be inserted to keep a in sorted order (the first occ. of x if it exists, or the position before the next largest element otherwise).

Binary Search

How do we *get* our list in sorted order in the first place?

Well, Python has a `sort` function, of course. Under the hood, this is using a comparison based sort (an efficient implementation).

The *asymptotic complexity* of comparison based sort is O(n log(n)) — so, like doing n binary searches.

To get on to the topic of interest (the suffix array), we won't learn the sort algorithms in detail here, but it's important to remember their complexity!

I suggest you read up more. You can search for "mergesort", and "quicksort" for an example of how these algorithms work.

Suffix Arrays

FINDING STRINGS WITH BINARY SEARCH

SCIENCE
ACADEMY

Subsequent slides not marked with * are courtesy of Ben Langmead

Suffix Array

Note this “sentinel” character; compares less than all other characters in the alphabet

$T\$ = abaaba\$$ ← *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

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"

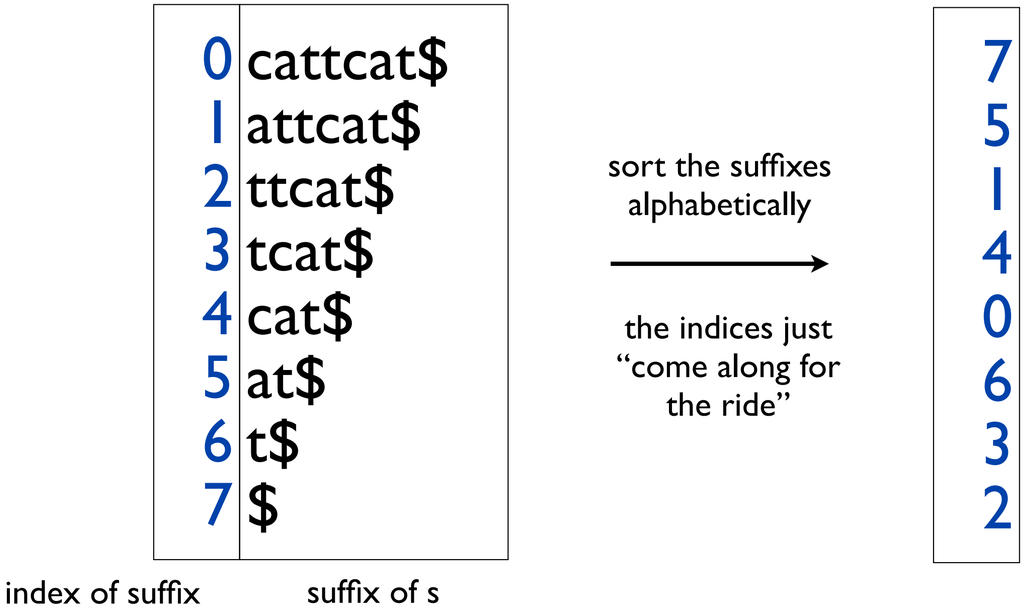
7	$
5	at$
1	attcat$
4	cat$
0	cattcat$
6	t$
3	tcat$
2	ttcat$

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.



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].

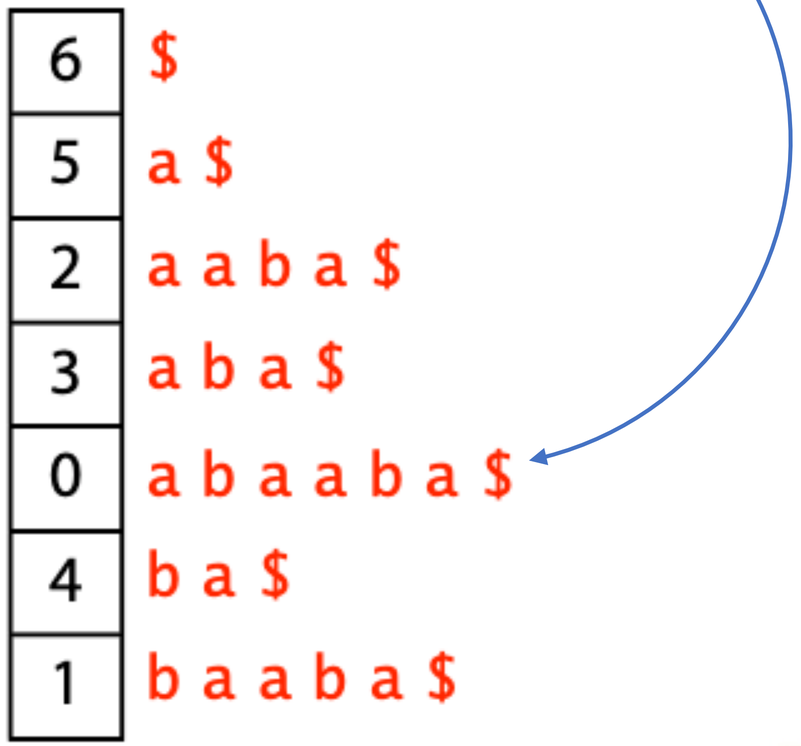
Puglisi, Smyth, Turpin. A Taxonomy of Suffix Array Construction Algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

Suffix Array: Query

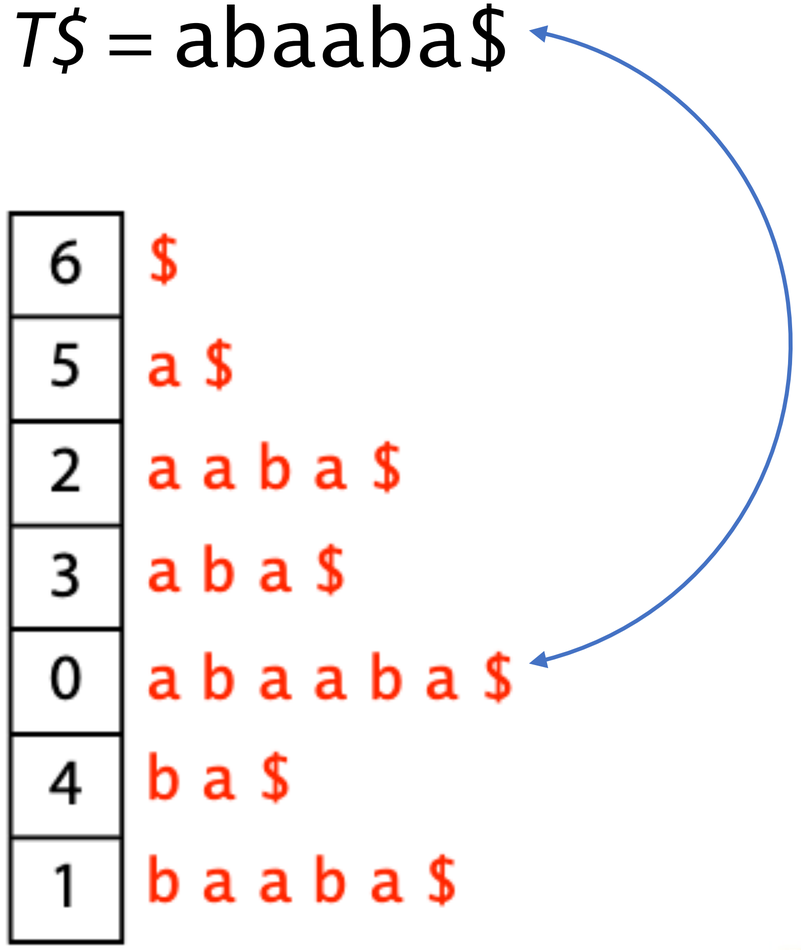
$T\$ = abaaba\$$

Check if pattern P is a substring of text T

- If P is a *substring* of T, then P must be a *prefix* of one or more *suffixes* of T.
- Since suffixes of T are sorted, all suffixes sharing a given prefix are *consecutive* in the suffix array (e.g. all a* before b*, all aa* before ab*, etc.)



Suffix Array: Query



Check if pattern P is a substring of text T

- If P is a *substring* of T, then P must be a *prefix* of one or more *suffixes* of T.
- Since suffixes of T are sorted, all suffixes sharing a given prefix are *consecutive* in the suffix array (e.g. all a* before b*, all aa* before ab*, etc.)

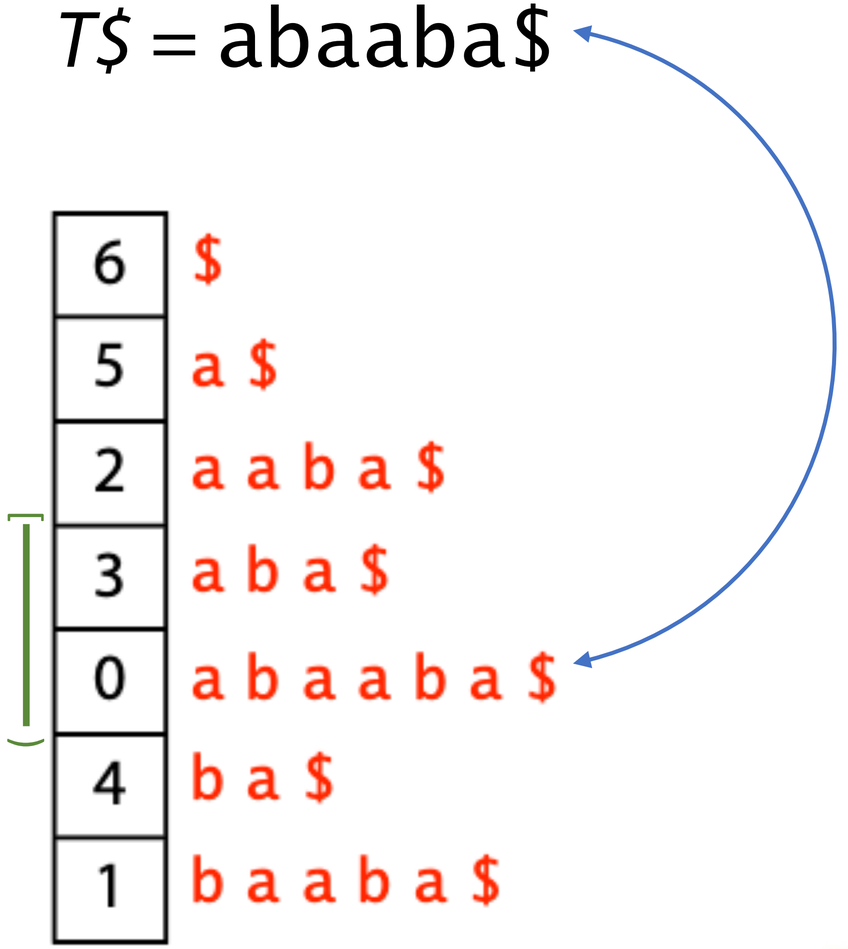
Just as with our array of numbers, we can use

binary search!

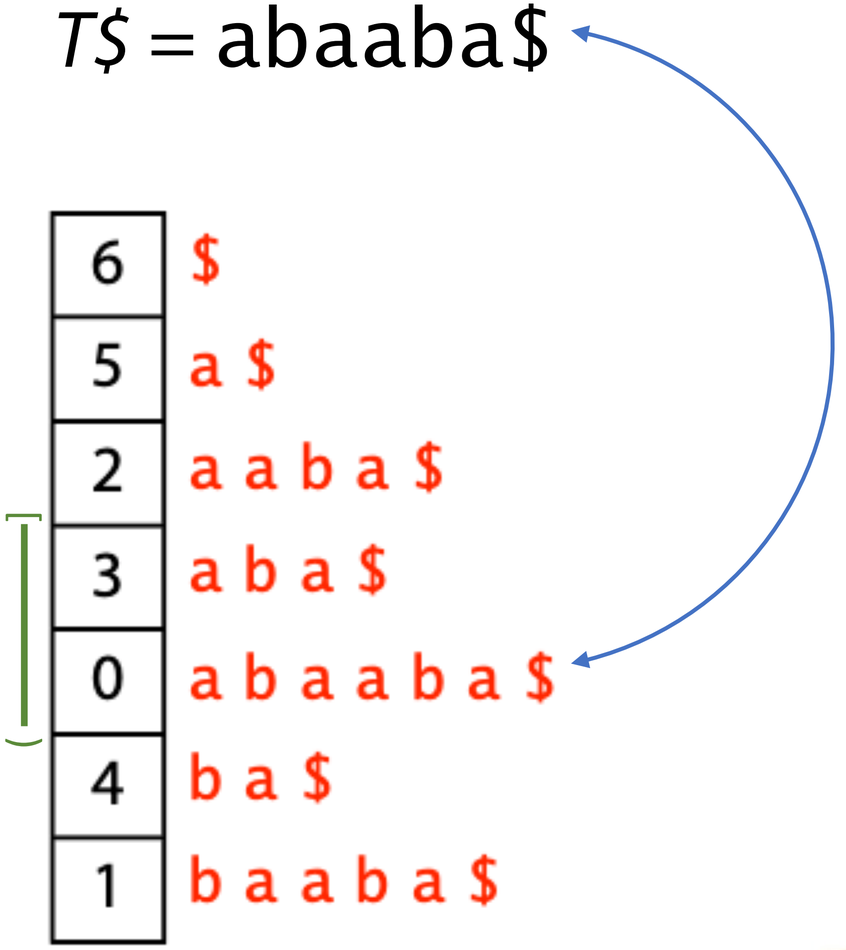
Suffix Array: Query

```
In [1]: from bisect import bisect_left, bisect_right

In [2]: t = "abaaba$"

In [3]: suffixes = list(sorted([t[i:] for i in range(len(t))]))

In [4]: st, en = bisect_left(suffixes, 'aba'), bisect_right(suffixes, 'abb')

In [5]: print(f"st: {st}, en: {en}")
st: 3, en: 5
```

[3, 5) — from index 3 (inclusive) to 5 (exclusive)



Suffix Array: Query



Can answer several questions:

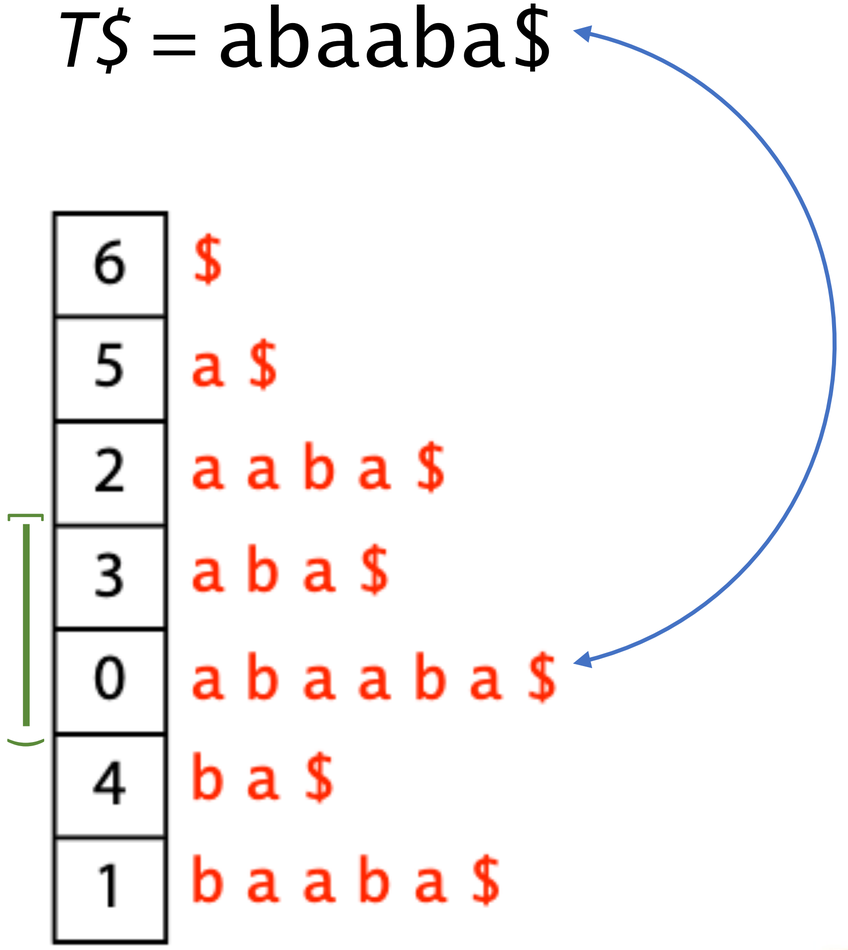
Does P occur in T?

Perform binary search, check if P is a prefix of the suffix where we end up.

How many times does P occur in T?

Search for the first place we could put P in sorted order and the first place we could put the next-largest prefix in sorted order — size of the range is # of occ.

Suffix Array: Query



Can answer several questions:

Does P occur in T?

Perform binary search, check if P is a prefix of the suffix where we end up.

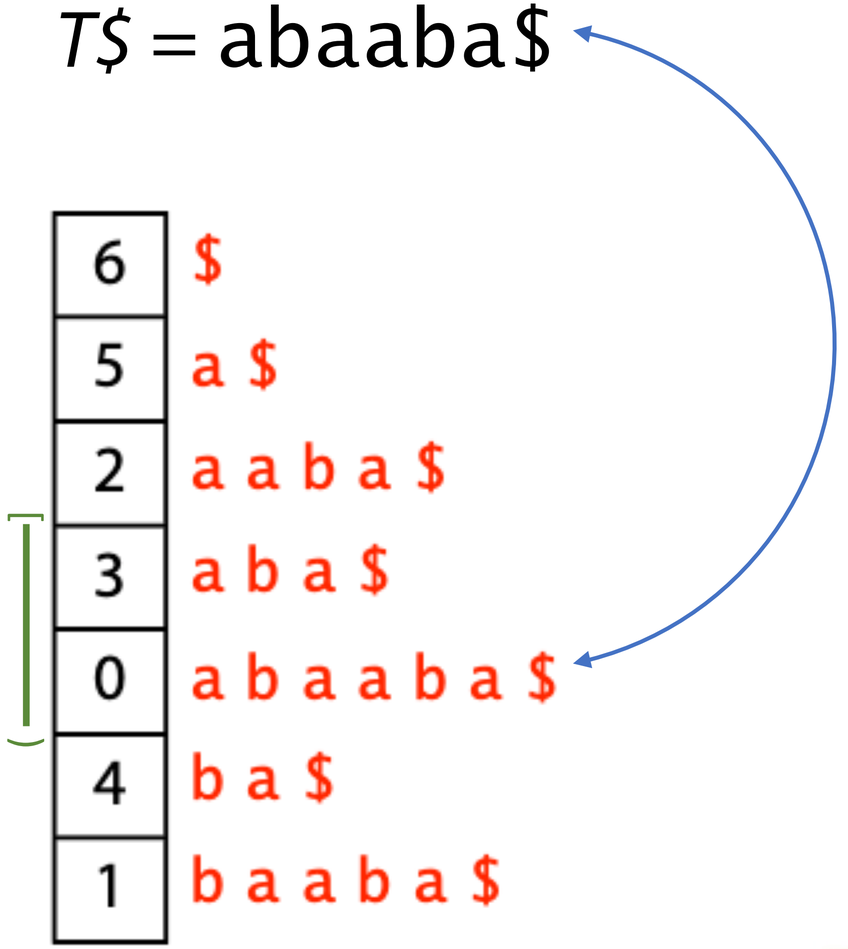
How many times does P occur in T?

Search for the first place we could put P in sorted order and the first place we could put the next-largest prefix in sorted order — size of the range is # of occ.

What is the complexity?

Say length(T) = m. Binary search takes at most O(log(m)) bisections, so O(log(m)), right?

Suffix Array: Query



Can answer several questions:

Does P occur in T?

Perform binary search, check if P is a prefix of the suffix where we end up.

How many times does P occur in T?

Search for the first place we could put P in sorted order and the first place we could put the next-largest prefix in sorted order — size of the range is # of occ.

What is the complexity?

Say length(T) = m. Binary search takes at most O(log(m)) bisections, so O(log(m)), right?

Unfortunately, not, because each comparison is not O(1), but O(n), so it takes O(n log(m))

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:

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

Suffix array: querying

Throughout search, invariant is maintained:

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

What do we do at each iteration?

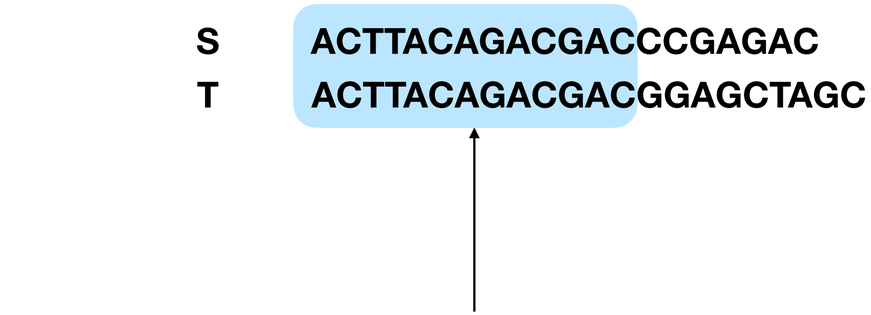
Let c = floor(($r + l$) / 2)
If $P < \mathbf{SA}[c]$, either stop or let $r = c$ and iterate
If P > $\mathbf{SA}[c]$, either stop or let $l = c$ and iterate

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

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).



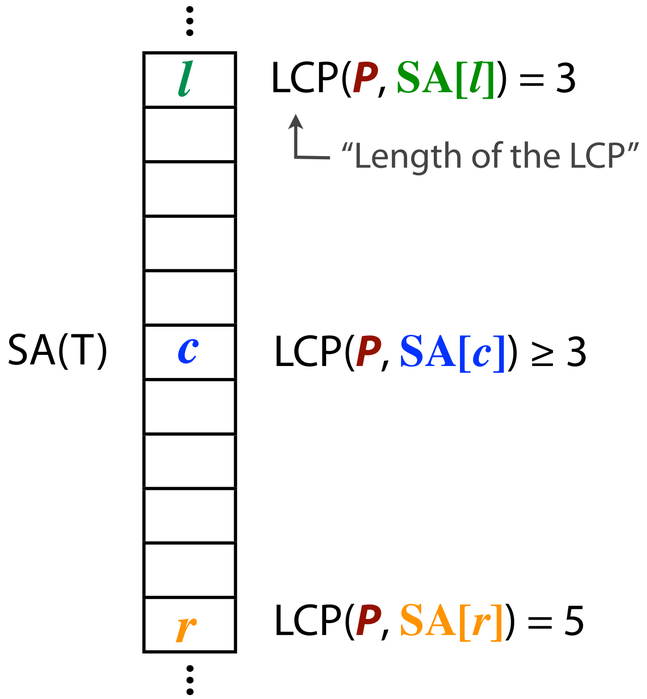
LCP(S,T) = ACTTACAGACGAC

|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:

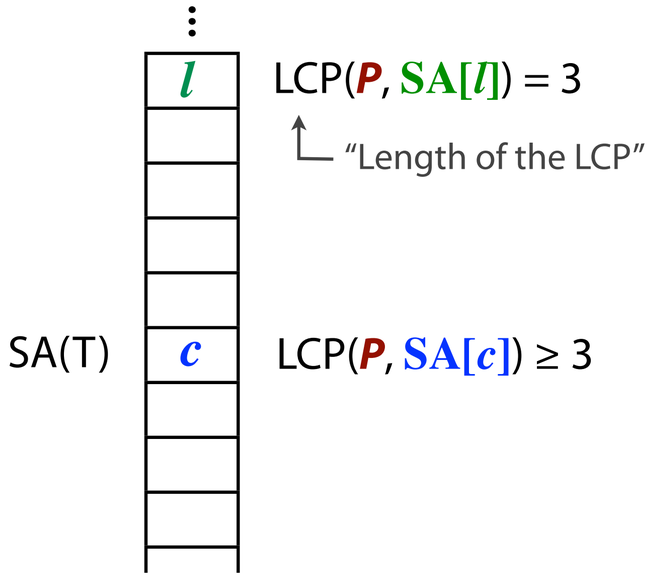
LCP(***P***, $\mathbf{SA}[c]$) ≥

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.

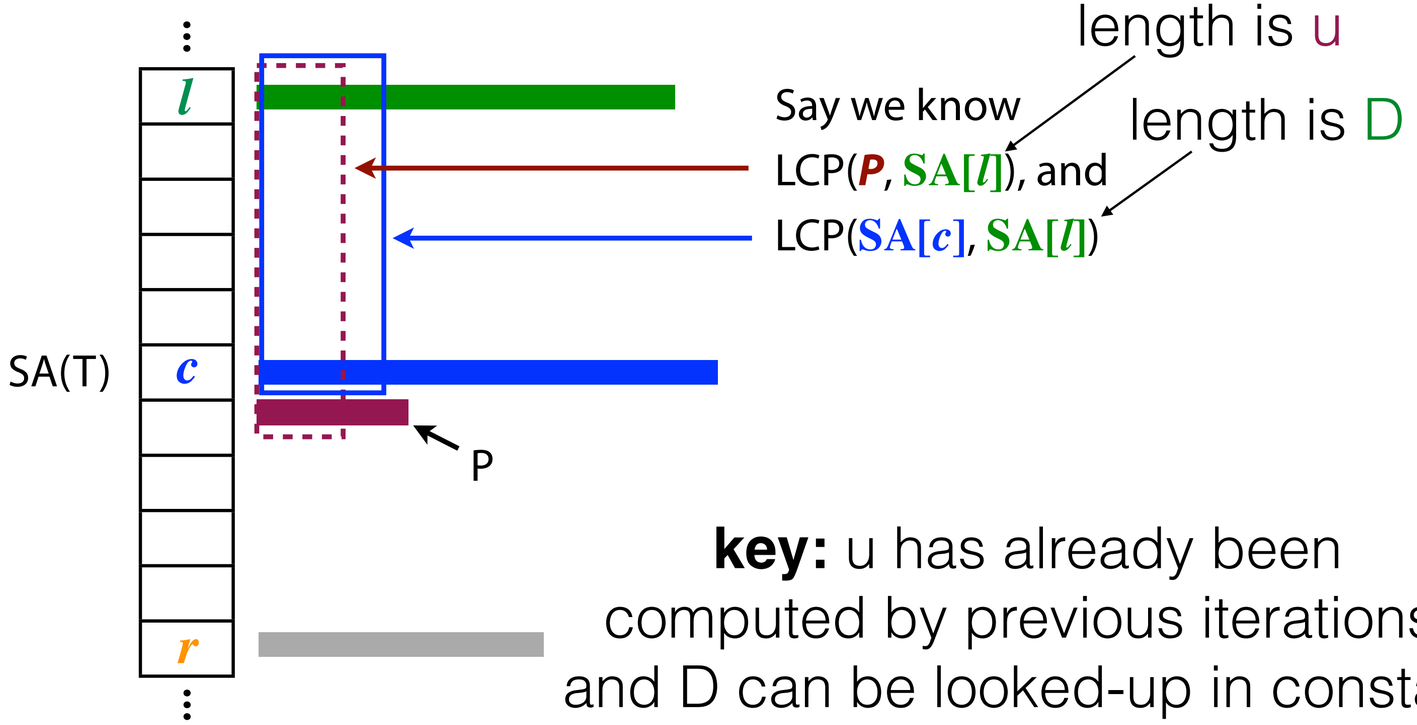
worst case example
$S=ac_{M-2}b$, P=c

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.

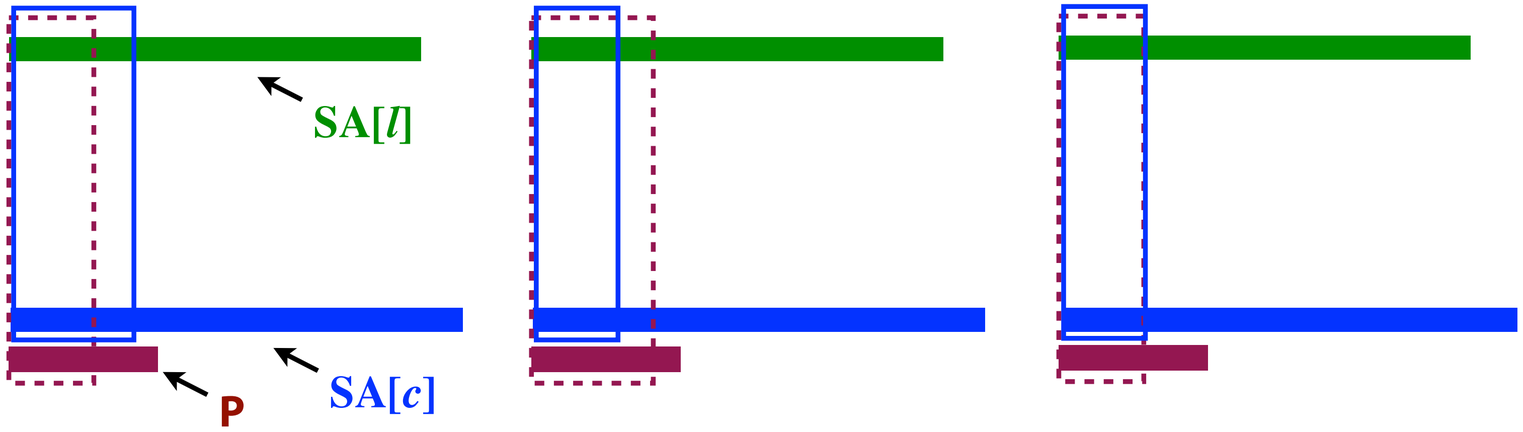
Take an iteration of binary search:



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.



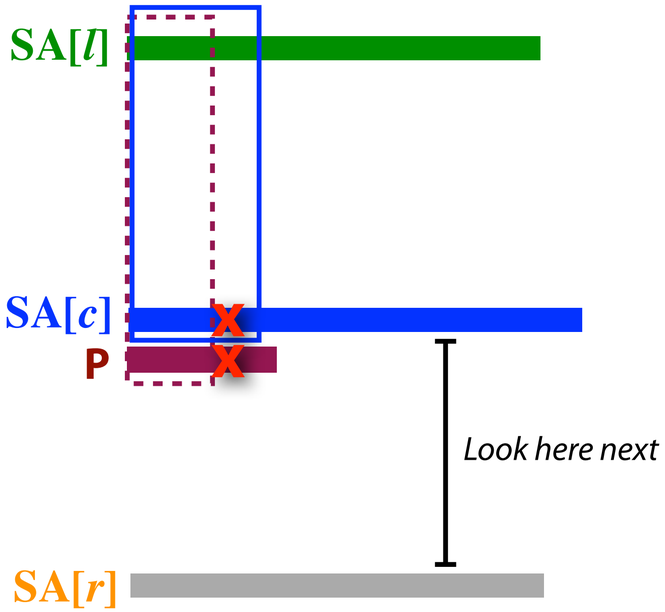
LCP(SA[c], SA[l]) > LCP(P, SA[l])

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*]**

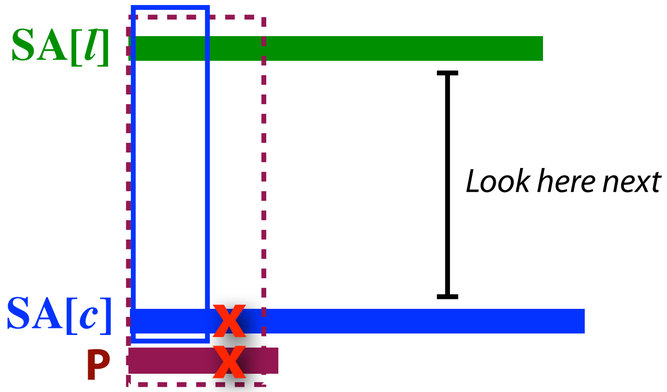
P > **SA[*c*]**

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*]**

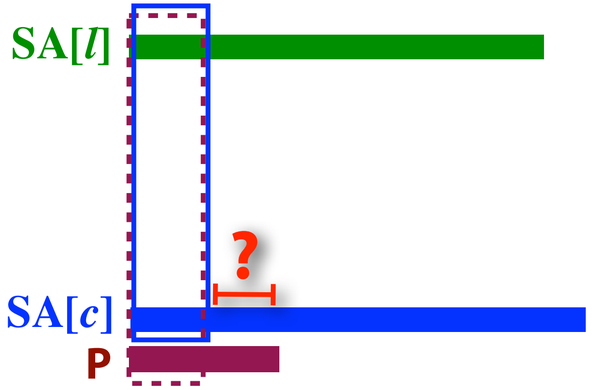
LCP(**SA[*c*]**, **SA[*l*]**) < LCP(***P***, **SA[*l*]**)

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

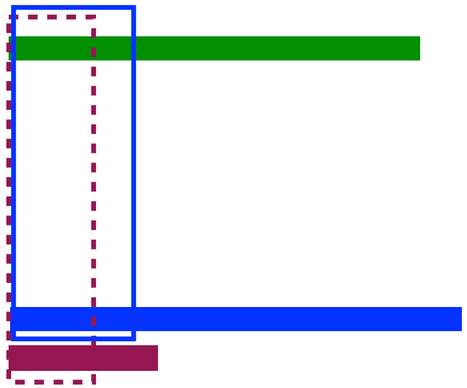
LCP(**SA[*c*]**, **SA[*l*]**) =

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

Suffix array: querying

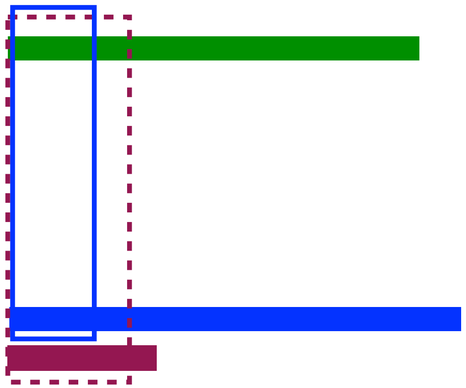
We improved binary search on suffix array from $O(n \log m)$ to $O(n + \log m)$ using information about Longest Common Prefixes (LCPs).

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



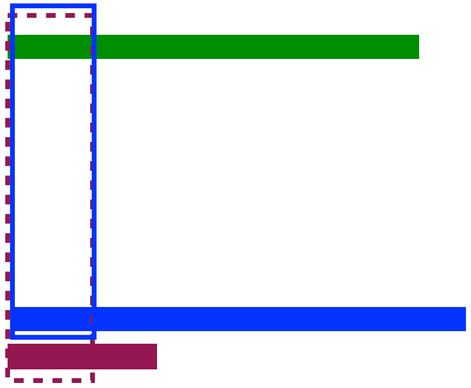
LCP(**SA[*c*]**, **SA[*l*]**) > LCP(***P***, **SA[*l*]**)

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

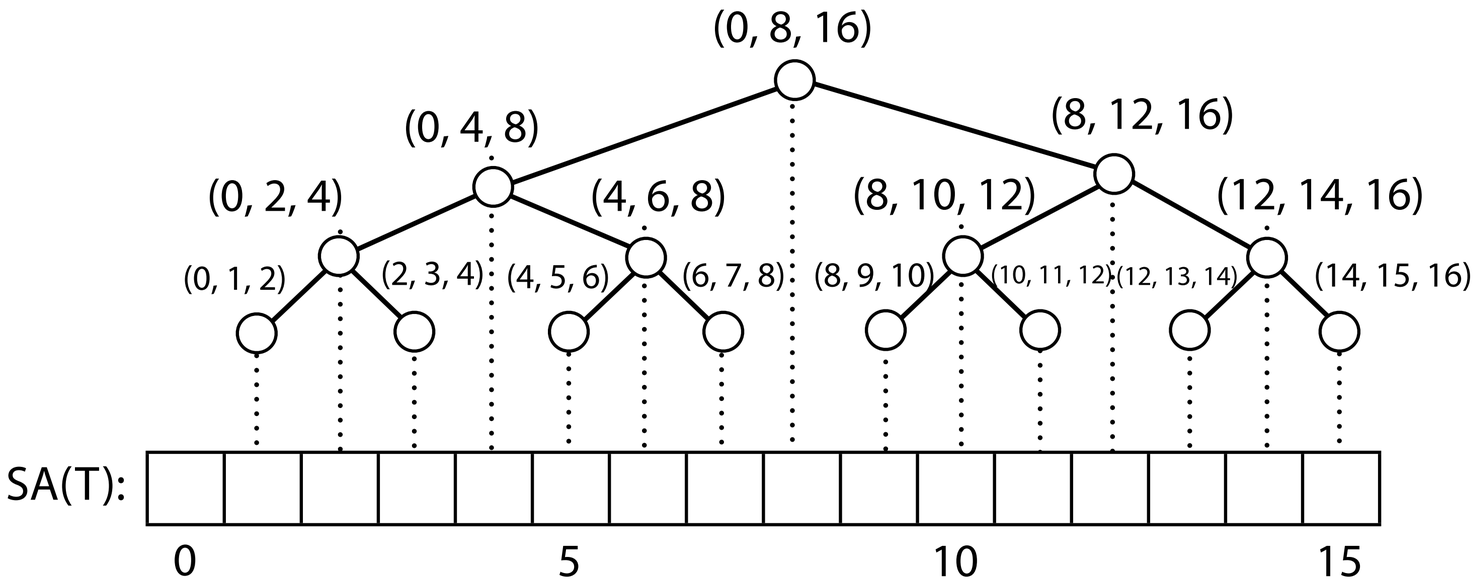
- To perform this “efficient” search, we must be able to look up LCP(SA[c], SA[l]) and LCP(SA[c], SA[r]).
- How can we pre-compute this information *efficiently?*
  - Which LCP values do we need (*hint: not all of them)*?
  - Given LCP for left and right sub-interval of a search, how can we compute LCP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

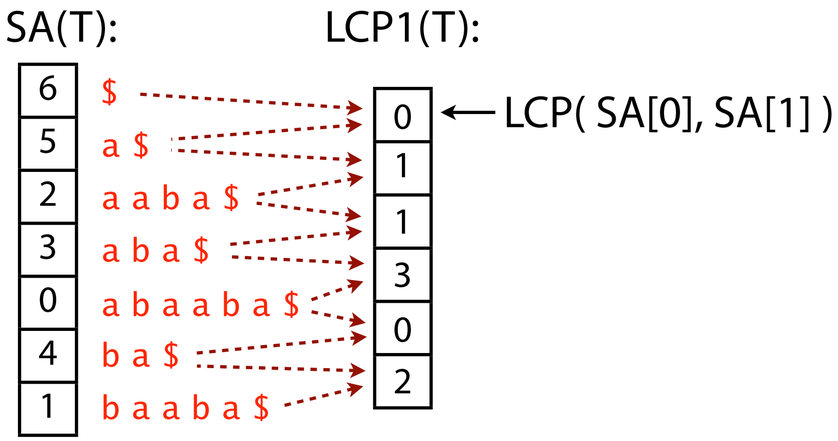


Example where *m* = 16 (incl. **$**) # search tree nodes = *m* -1

Suffix array: LCPs

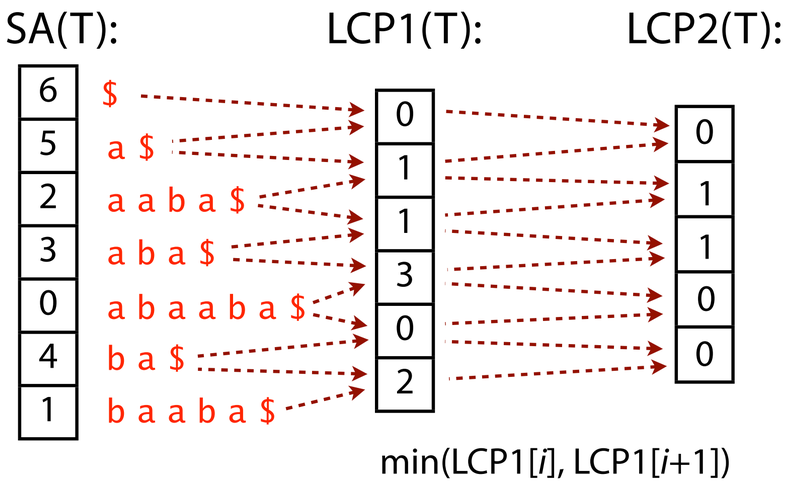
Suffix Array (SA) has *m* elements

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



Suffix array: LCPs

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

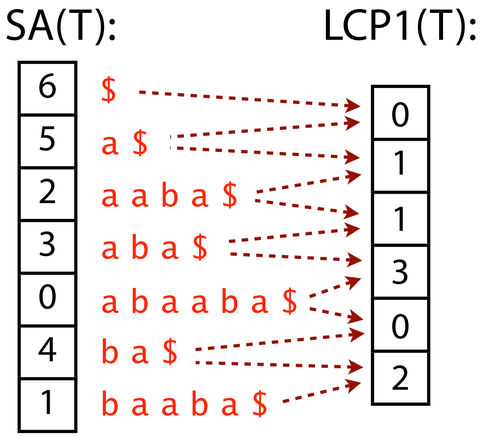


In fact, LCP of a range of consecutive suffixes in SA equals the minimum LCP1 among adjacent pairs in the range

LCP1 is a building block for other useful LCPs

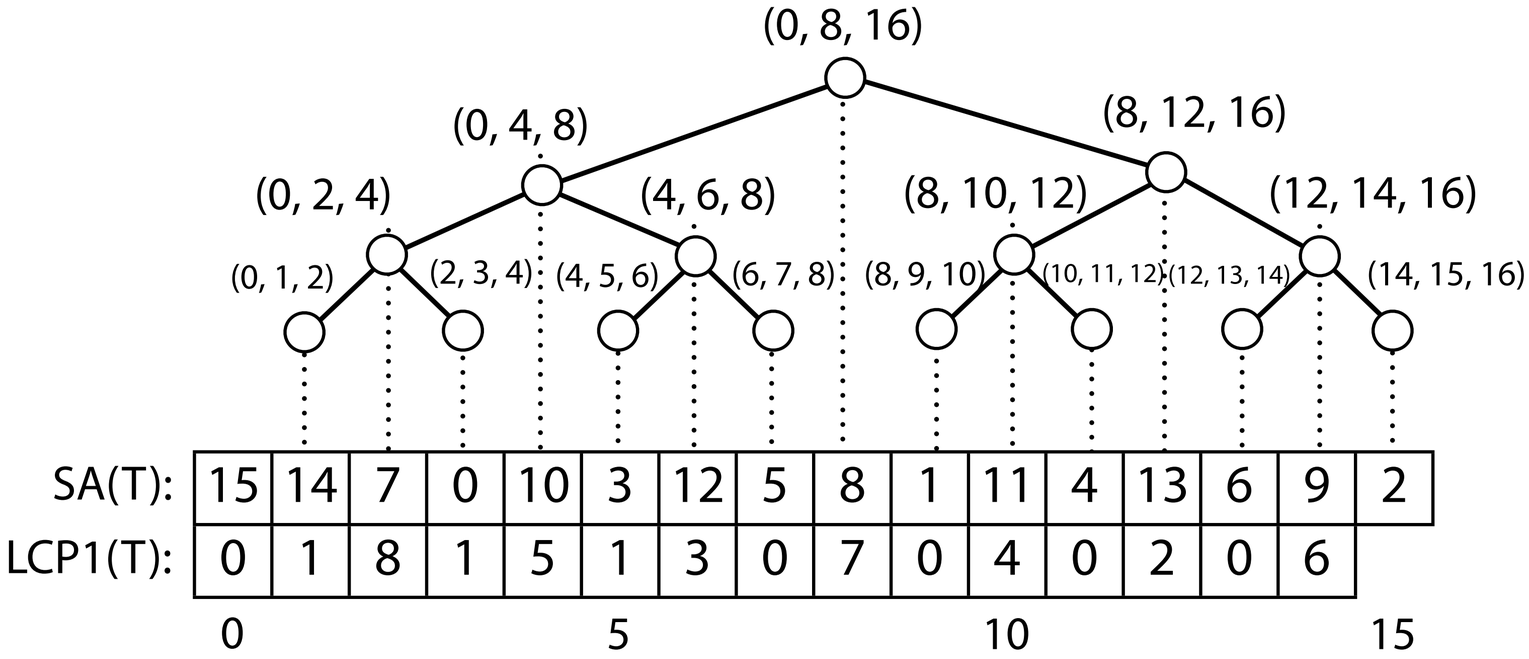
Suffix array: LCPs

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



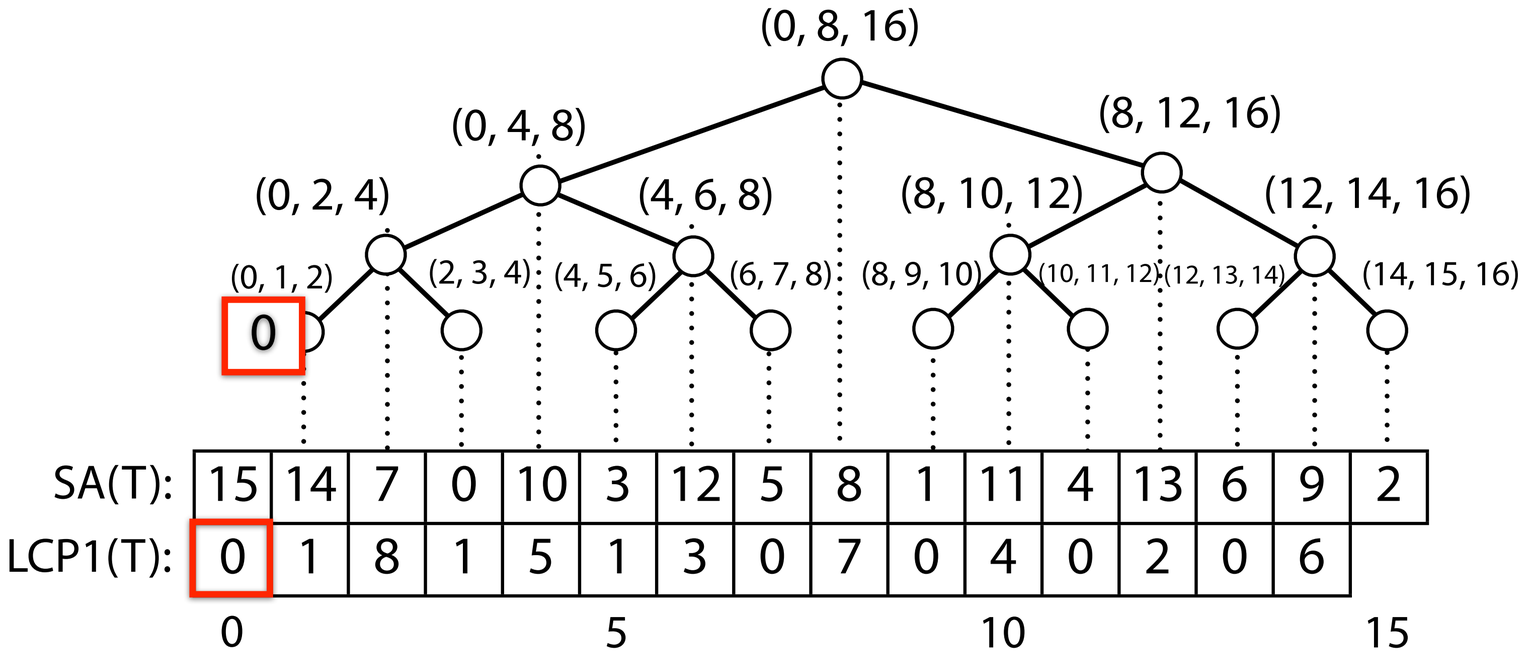
Suffix array: LCPs

T = abracadabracada



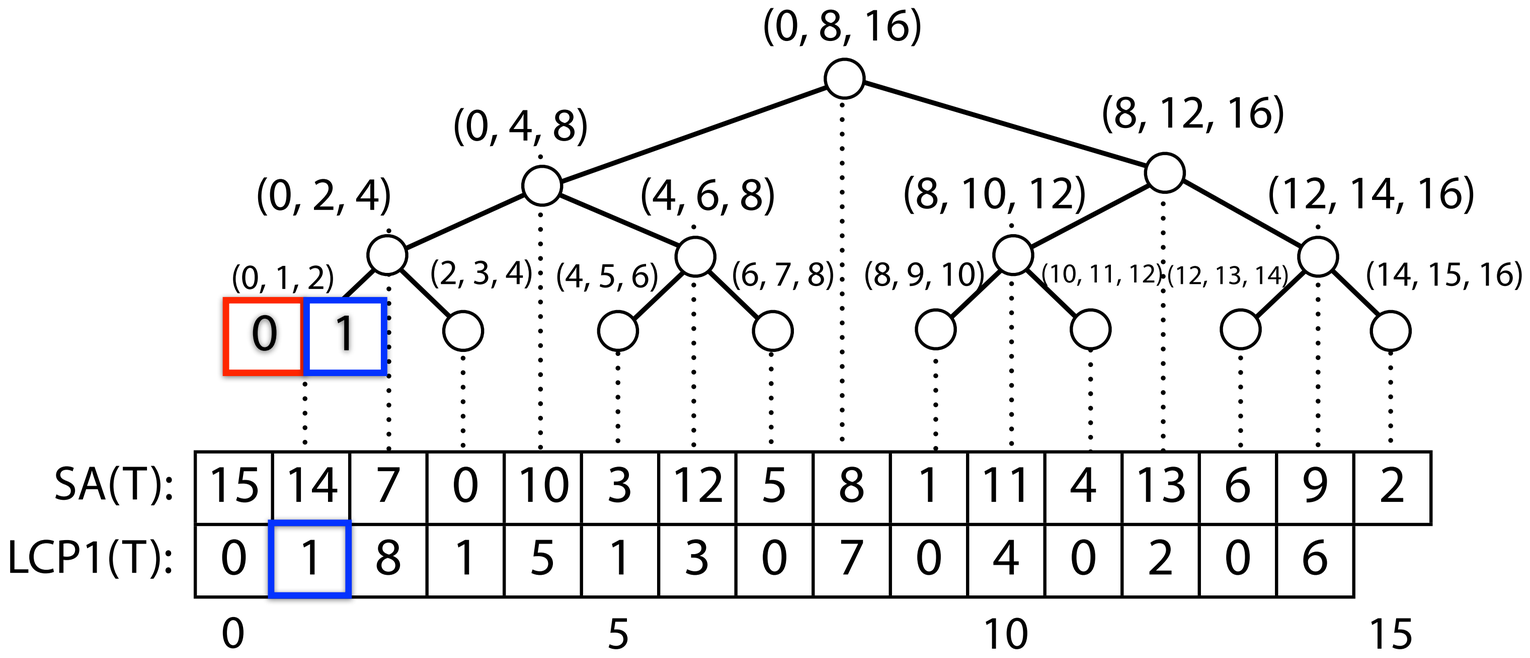
Suffix array: LCPs

T = abracadabracada$



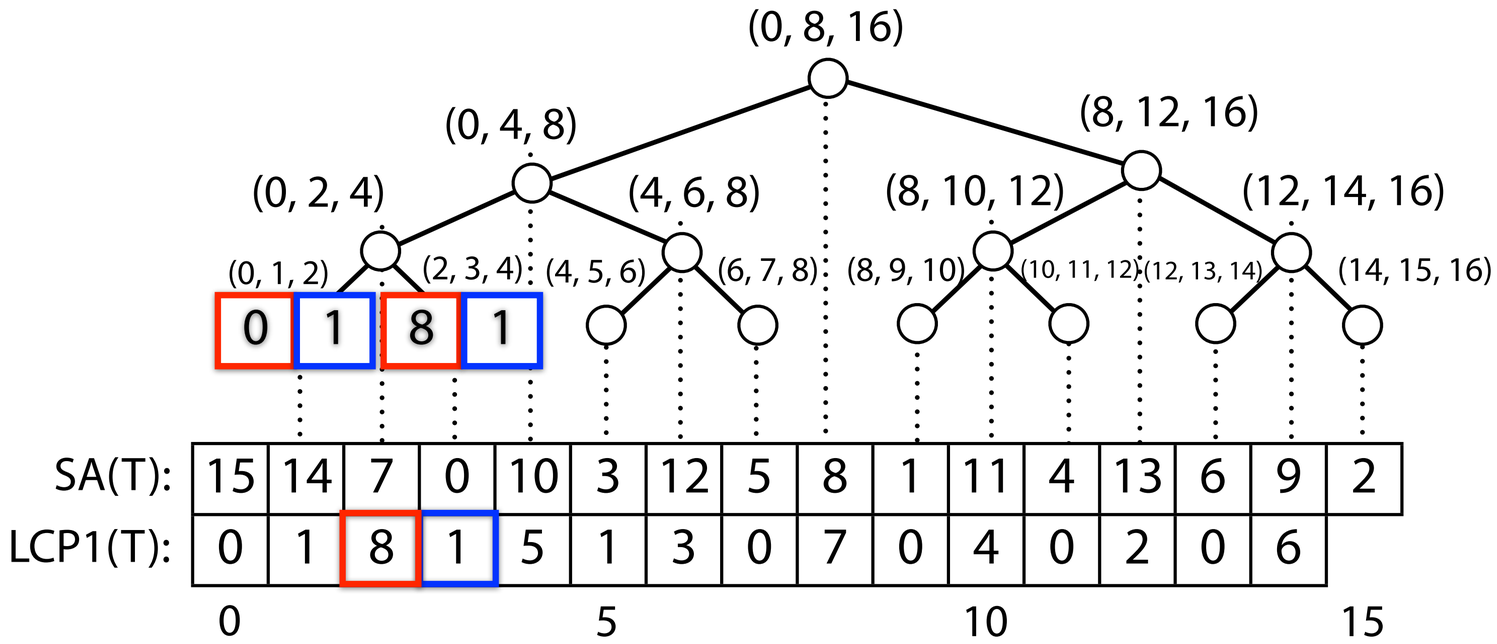
Suffix array: LCPs

T = abracadabracada$



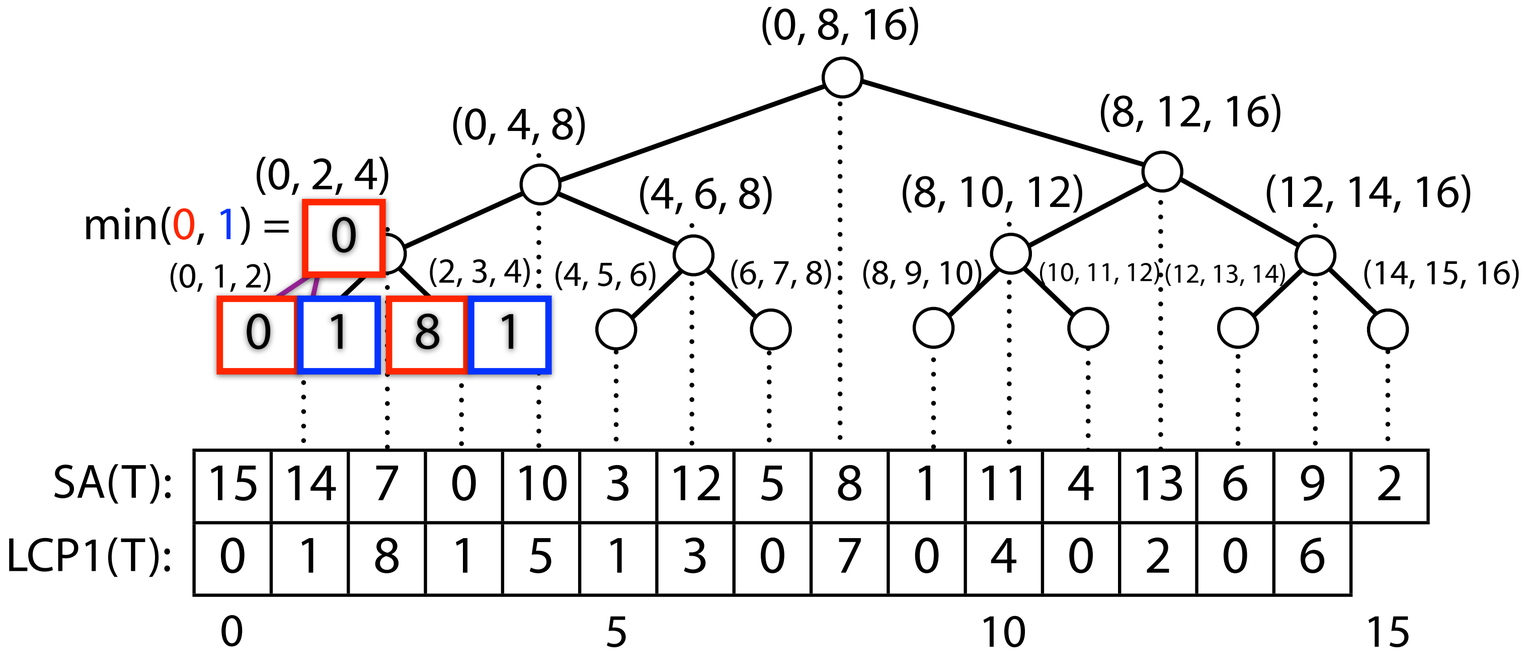
Suffix array: LCPs

T = abracadabracada$



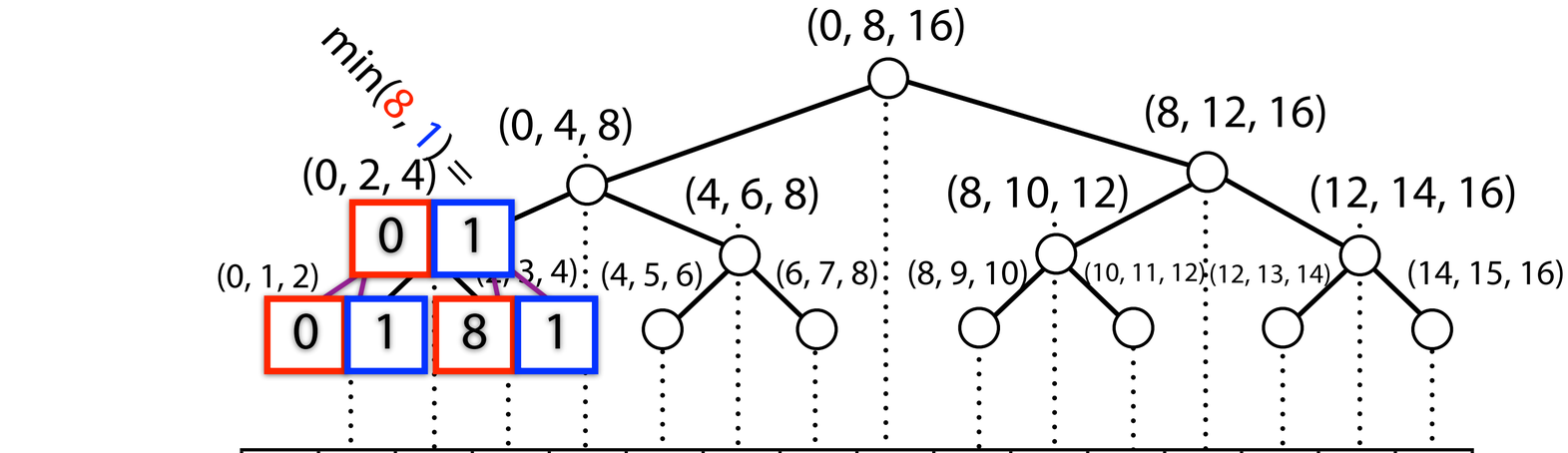
Suffix array: LCPs

T = abracadabracada$



Suffix array: LCPs

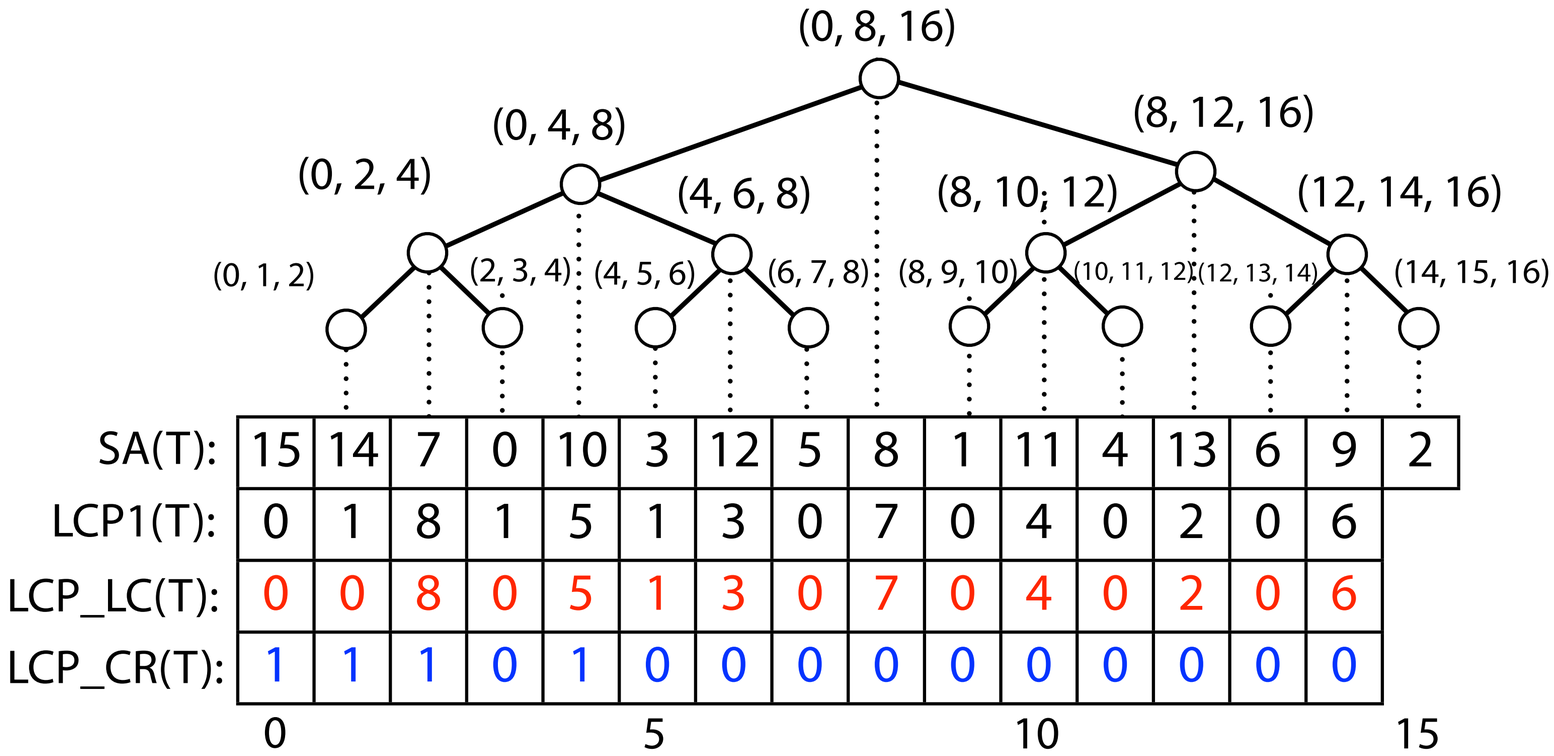
T = abracadabracada$



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

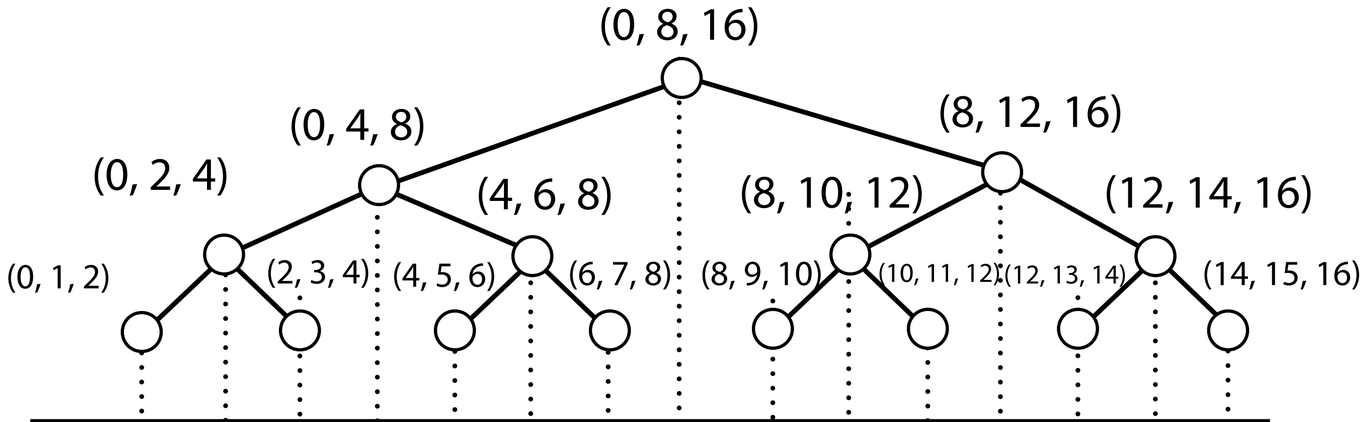
NOTE: These arrays are “shifted” by 1 — the value in LCP_LC corresponding to (0, 1, 2) is at LCP_LC[0], not LCP_LC[1]. So, to look up LCP(SA[l], SA[c]) we look at LCP_LC[c-1]

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[l], 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. ~*m* integers (SA)
2. ~*m* integers (SA)
3. ~3*m* integers (SA, LCP_LC, LCP_CR)

Suffix array: performance comparison

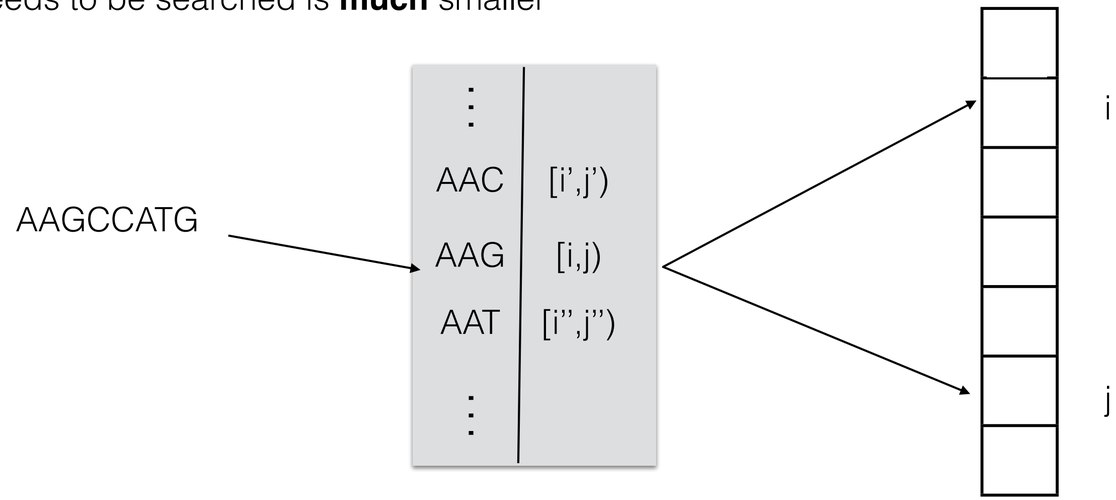
	Super accelerant	Simple accelerant	No accelerant
python -O	68.78 s	69.80 s	102.71 s
pypy -O	5.37 s	5.21 s	8.74 s
# character comparisons	99.5 M	117 M	235 M

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).

*

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