Week 7

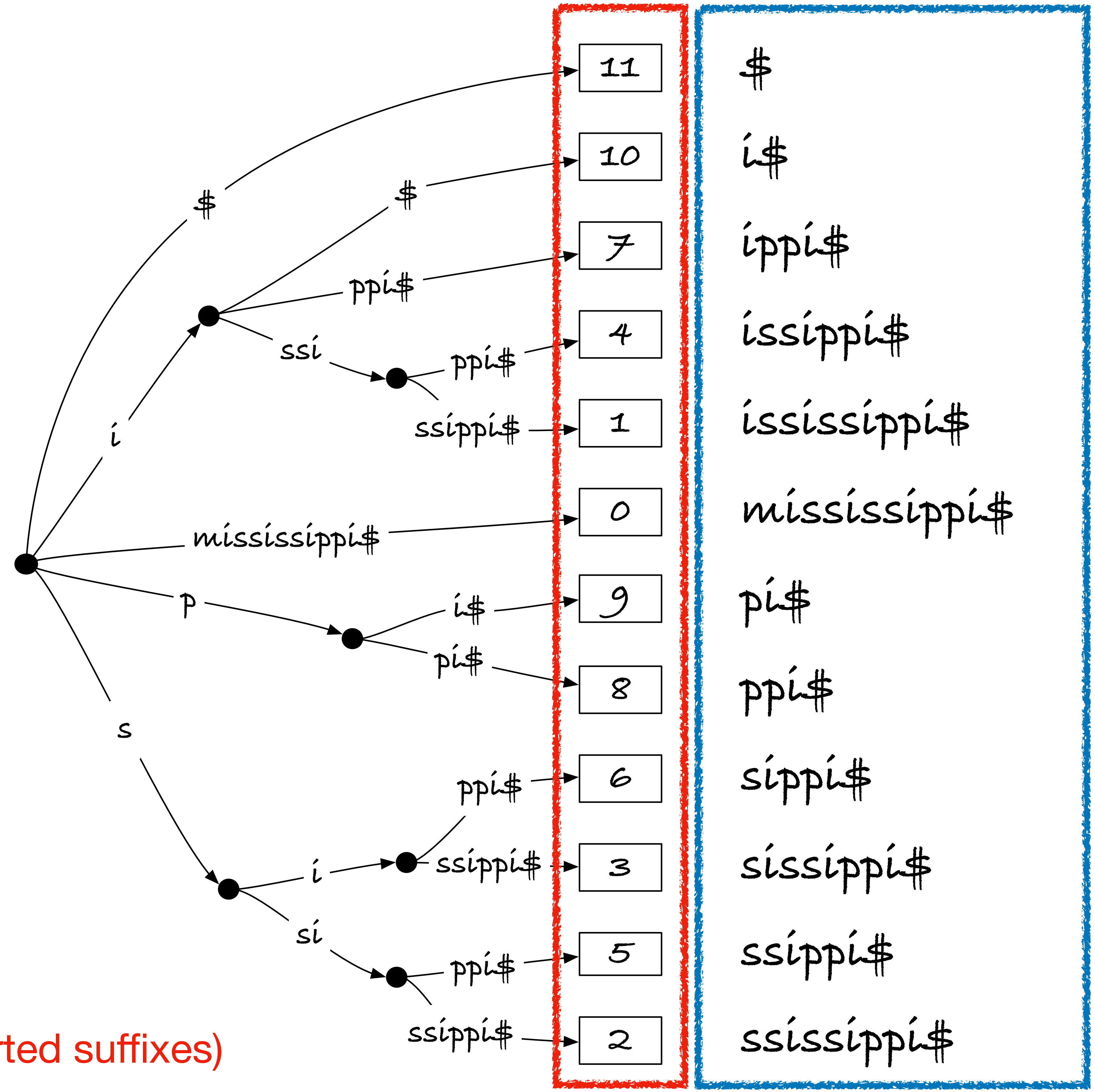
Suffix & LCP arrays + suffix tree construction

Introduction to two important arrays: suffix and lcp arrays

- Suffix and lcp arrays use less memory than suffix trees, but can do most of the same things
- We will explore how they relate to suffix trees
    - how we can construct them from a suffix tree
    - how we can construct a suffix tree from them
- We never do that circular construction, but it is the same technique you need to simulate one structure using the other (normally: simulating a suffix tree from a suffix array)

Suffix arrays

Sorted suffixes



Suffix array
(indices of sorted suffixes)

Searching

	$p \in x?$	locate $p \in x$
KMP or boder array:	$O(n+m) \quad [O(k(n+m))]$	$O(n+m) \quad [O(k(n+m))]$
Suffix tree:	$O(n+m) \quad [O(n+km)]$	$O(n+m) \quad [O(n+km+z)]$
Suffix array:	$O(n+???) \quad [O(n+???)]$	$O(n+???) \quad [O(n+???)]$

Binary search

$p \in x?$

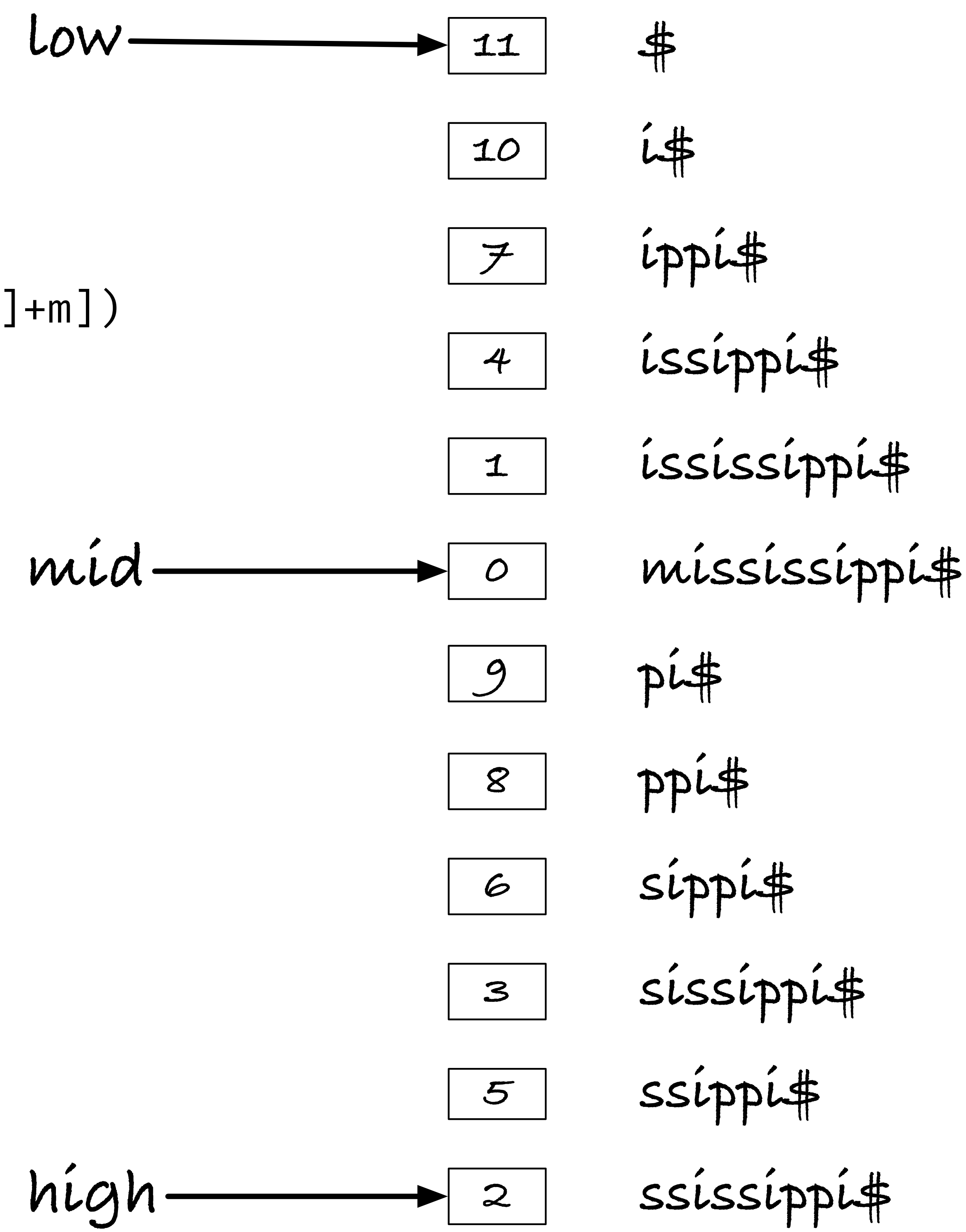
```
cmp = strcmp(p, x[sa[mid]:sa[mid]+m])
if cmp == 0: return True
if cmp < 0: high = mid
else: low = mid + 1
```

What is the running time?

O(m log n) for the search

$\log_2$ of a billion is only 30, so
it is not a massive cost compared to
other overhead we might incur.

(but still diff btw a day and a month)

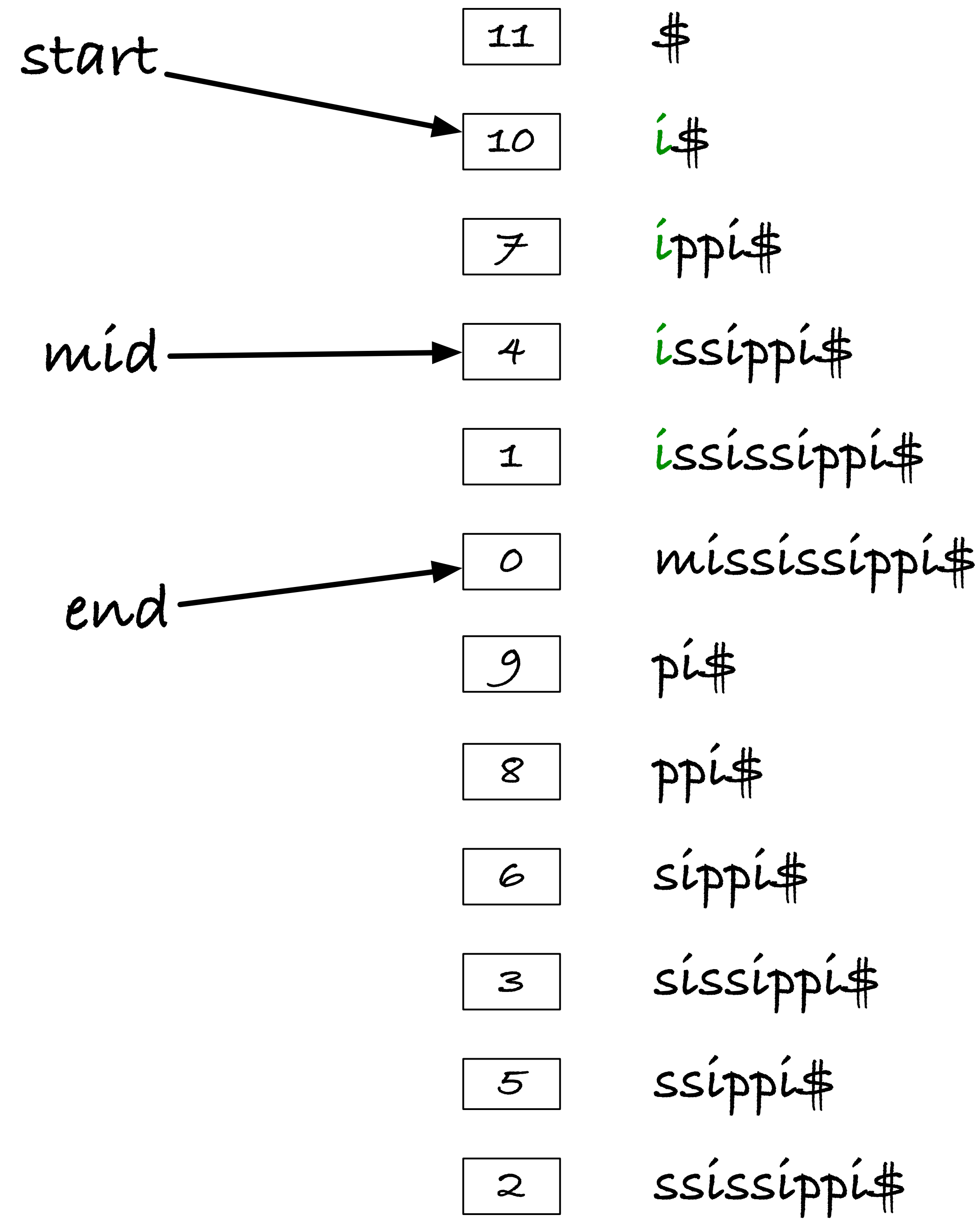


Binary search

```
p=“i”
x[sa[mid]:sa[mid]+m] == “i”

find start and end so start points to the
first occurrence and end one past the last

occ = [sa[i] for i in range(start, end)]
```



Finding [start,end] costs O(mz)

Binary search

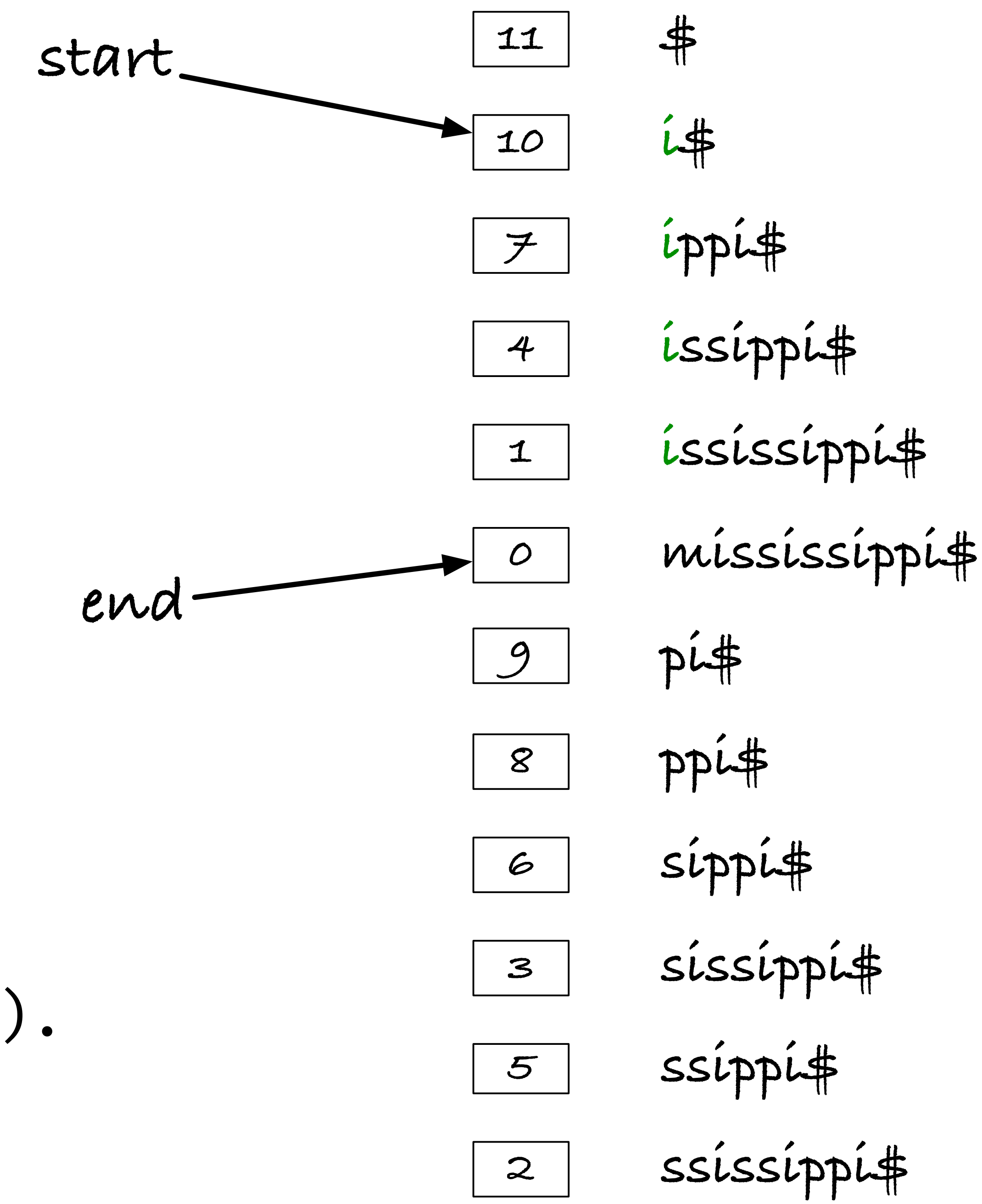
```
start = lower_bound(p, x, sa)
end = upper_bound(p, x, sa)
occ = [sa[i] for i in range(start, end)]
```

```
lower_bound(p, x, sa) finds the
smallest index i such
that x[sa[i]:sa[i]+m] ⩾ p

upper_bound(p, x, sa) finds the largest
index i such that x[sa[i-1]:sa[i-1]+m] ⩽ p

both use binary search and run in O(m log n).
```

Finding [start,end] costs O(m log n)



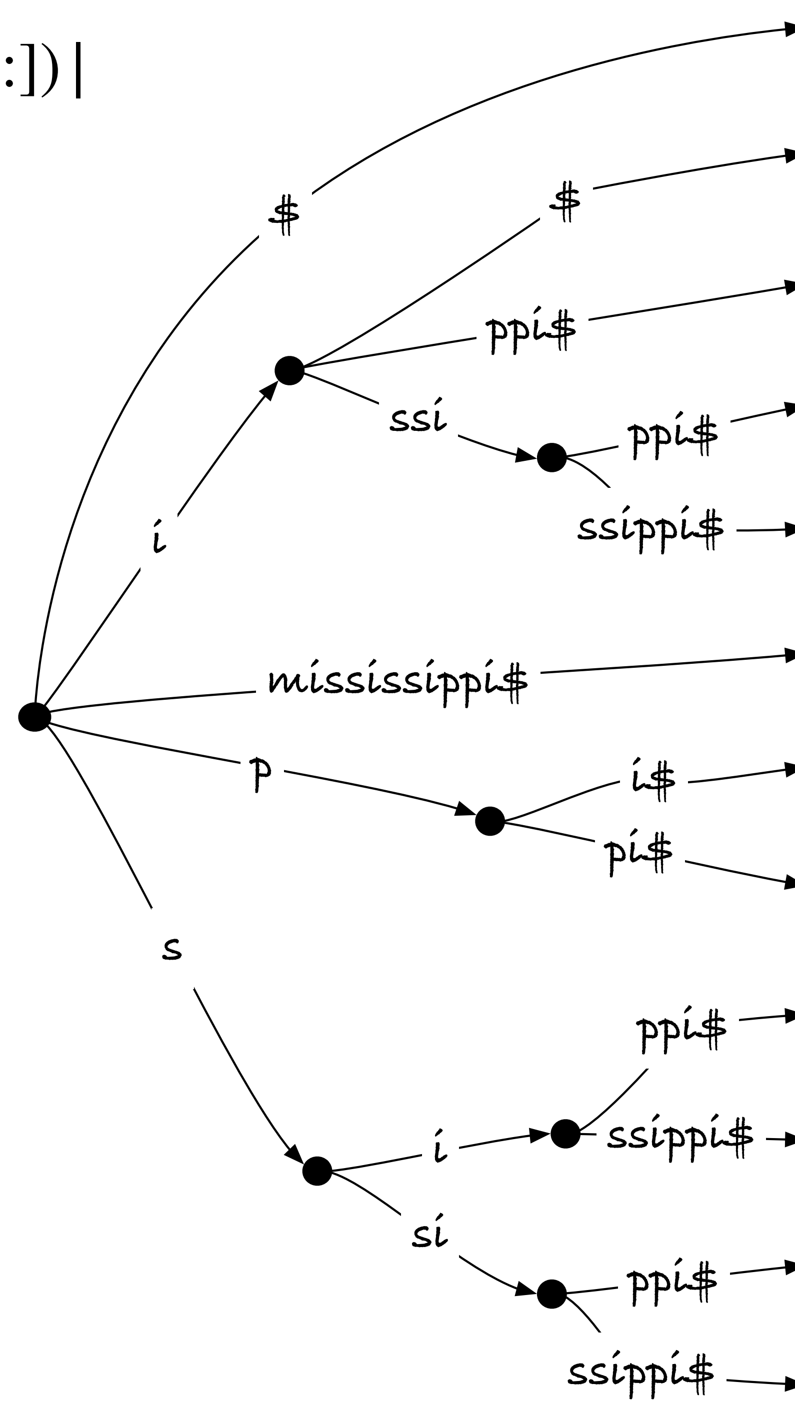
Binary search

- lower_bound and upper_bound are left as exercises
- It is possible to speed up the binary search by keeping track of prefixes in the interval that you know you match; that is also left as an exercise
- Even with the speed-up we have O(m log n + z) search time.
- This can be improved to O(m + log n + z) using another array
    - We will see the other array in a minute, but not the search algorithm
    - We can get rid of the log n completely in other ways…

lcp arrays

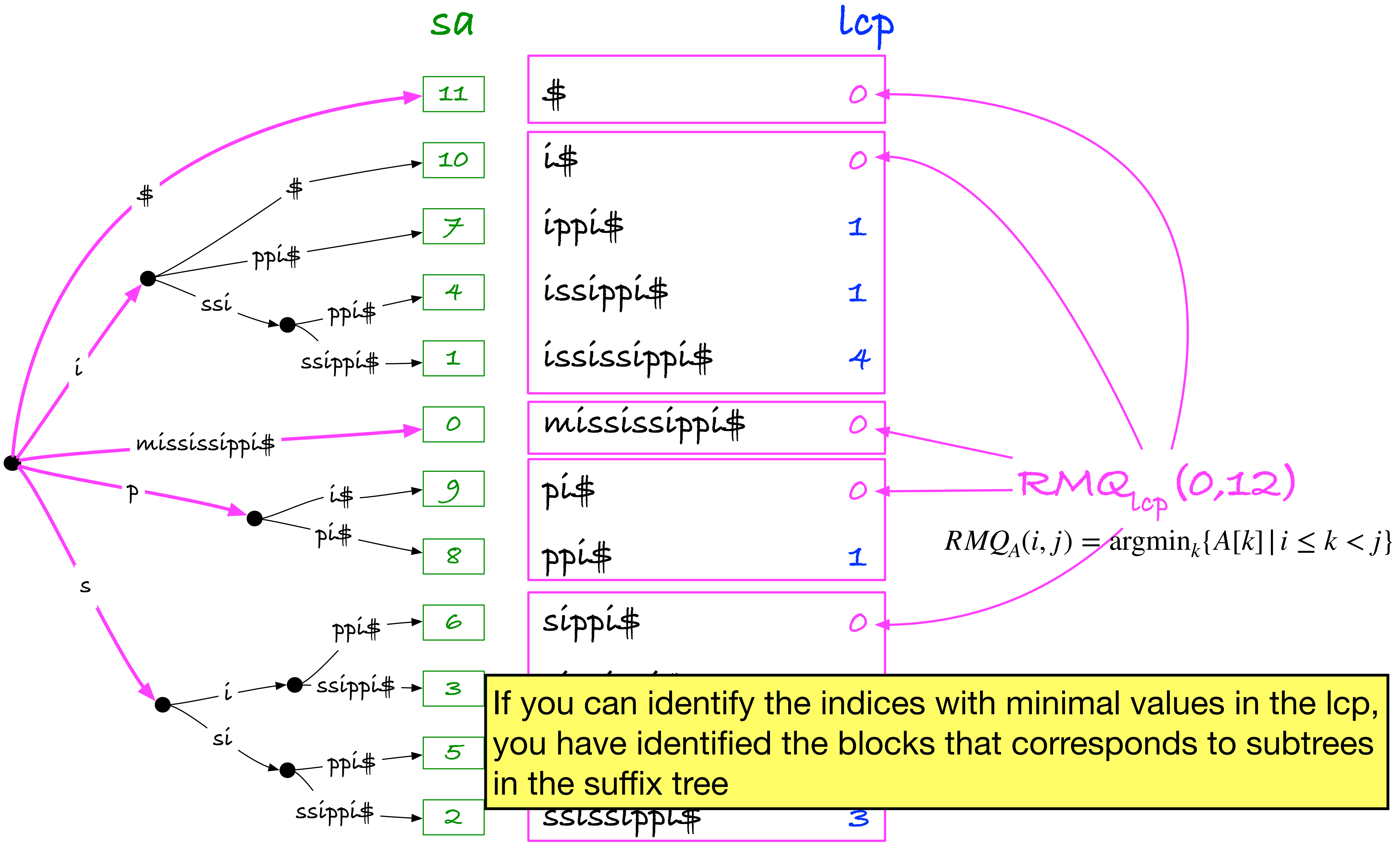
$\text{lcp}[0] = 0$

$\text{lcp}[i] = |\text{LCP}(x[\text{sa}[i]:], x[\text{sa}[i-1]:])|$



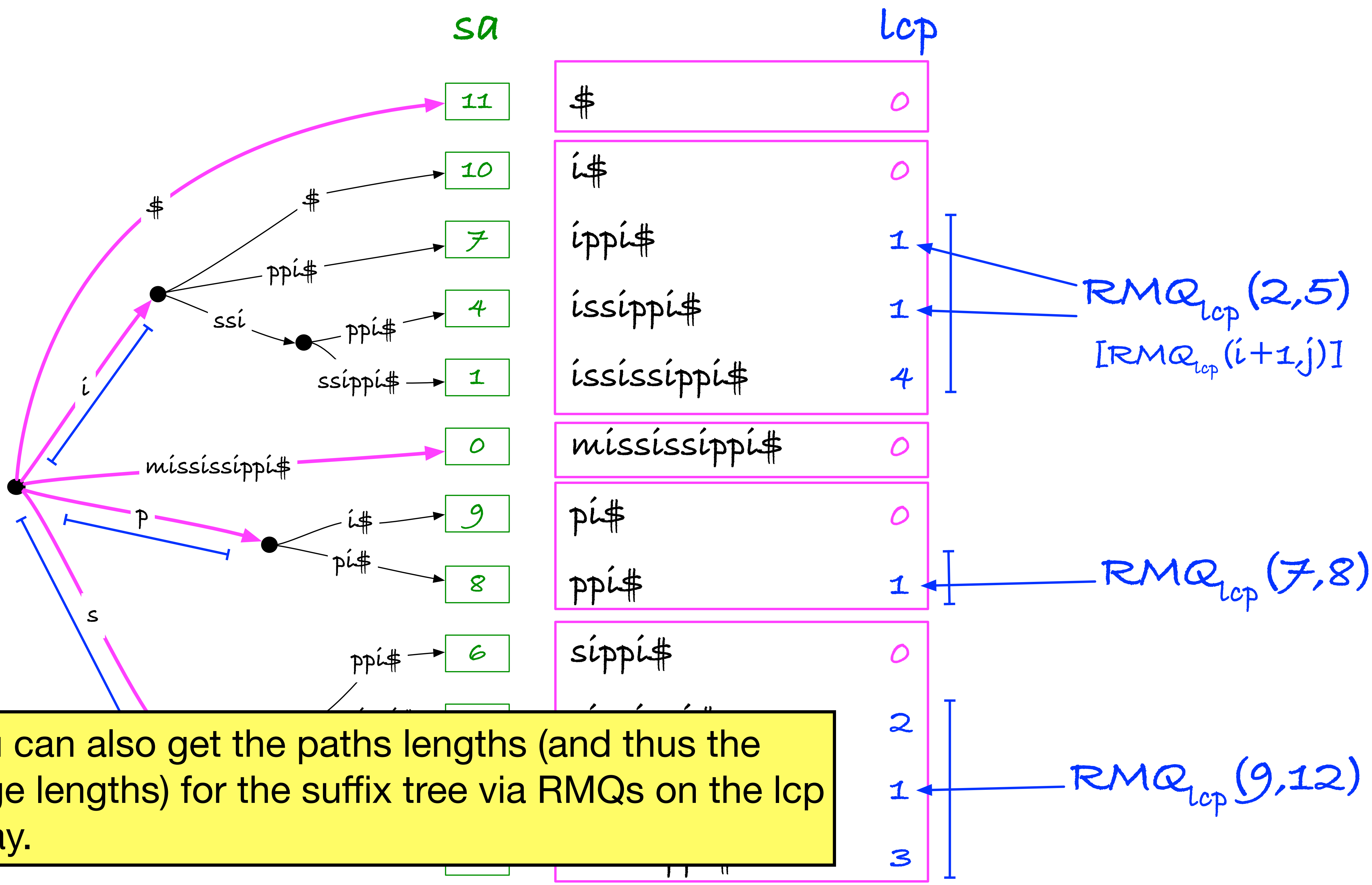
sa		lcp
11	$	0
10	i$	0
7	ippi$	1
4	issippi$	1
1	ississippi$	4
0	mississippi$	0
9	pi$	0
8	ppi$	1
6	sippi$	0
3	sissippi$	2
5	ssippi$	1
2	ssissippi$	3

lcp arrays and interval trees



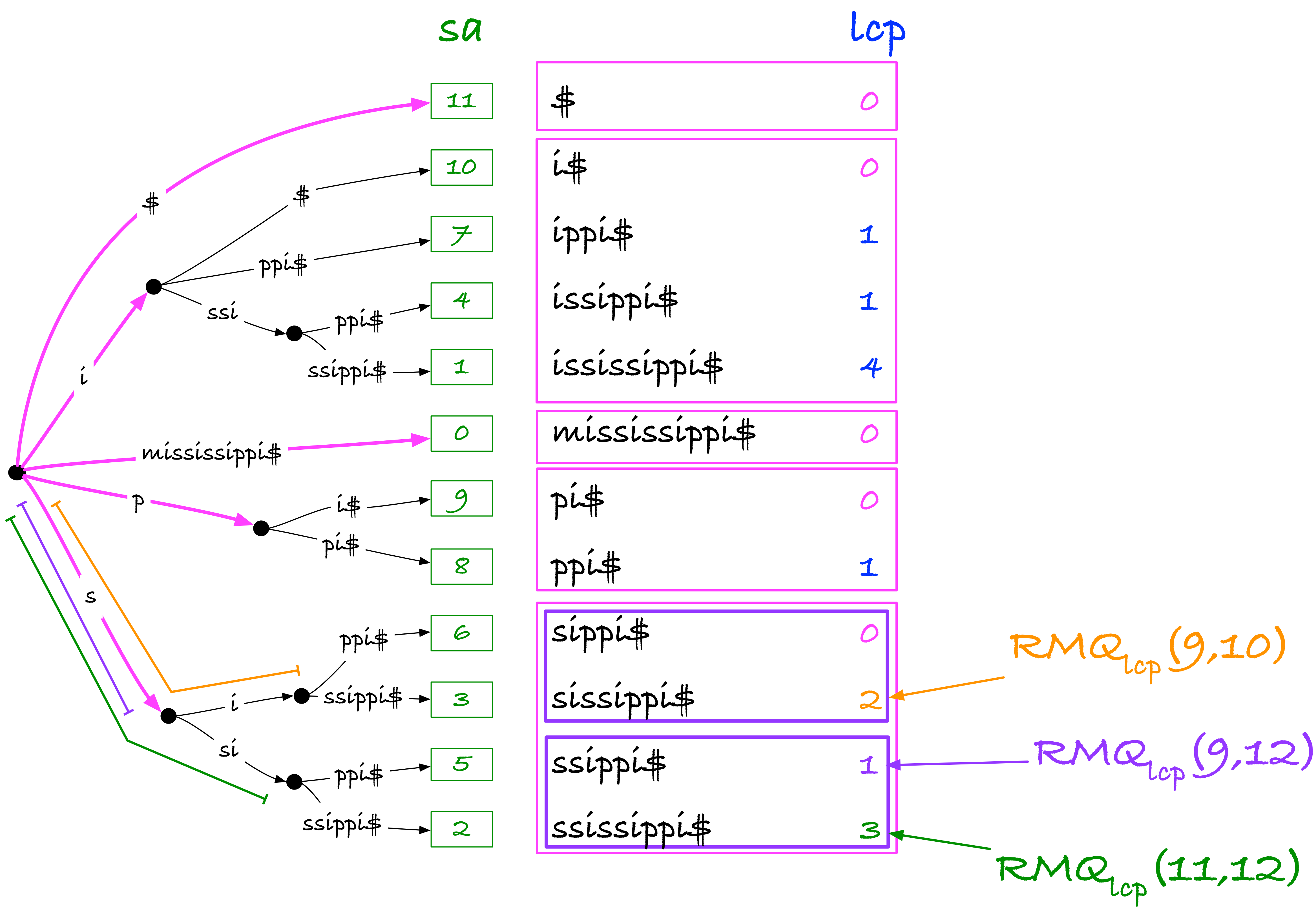
If you can identify the indices with minimal values in the lcp, you have identified the blocks that corresponds to subtrees in the suffix tree

lcp arrays and interval trees

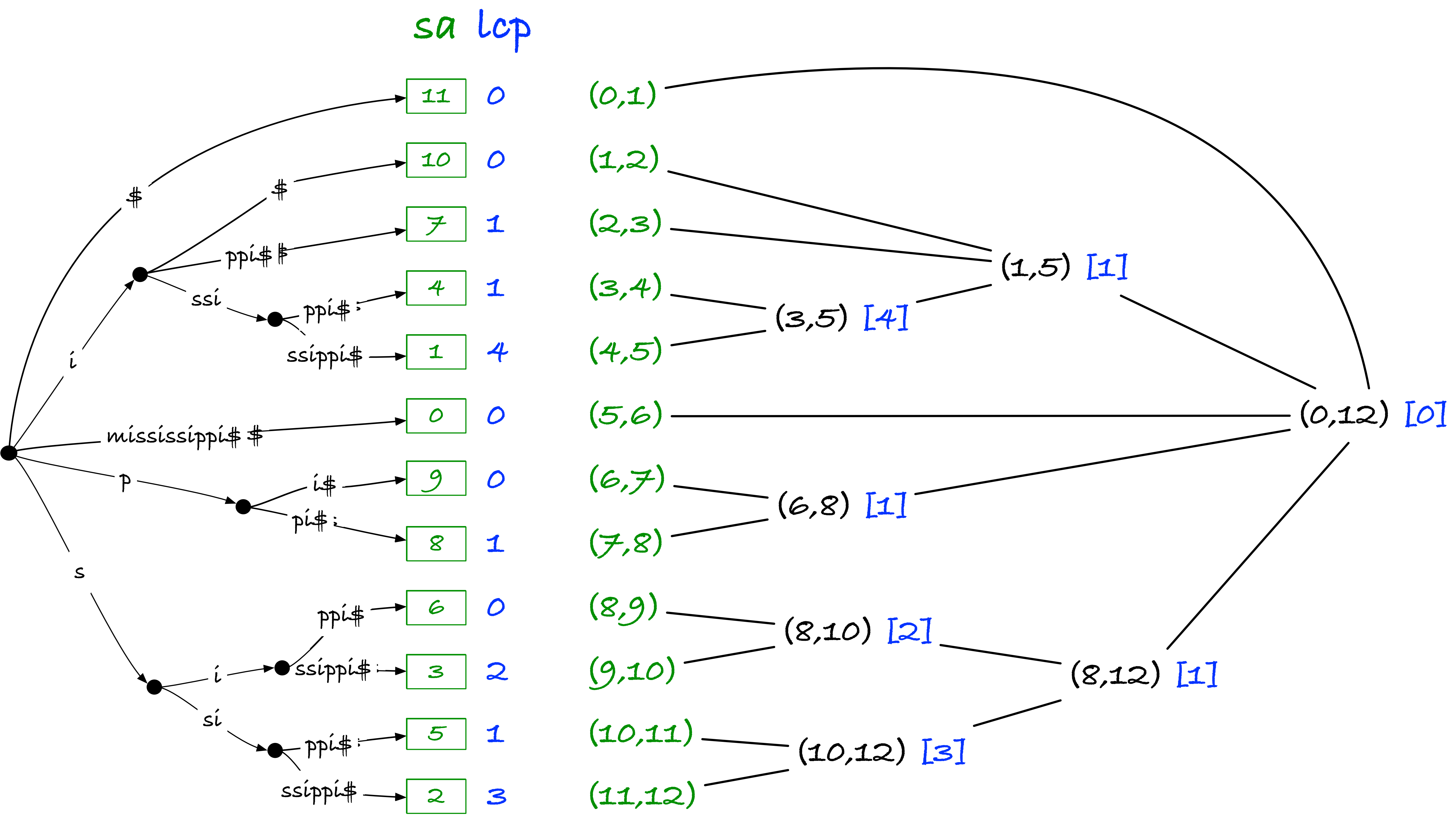


You can also get the paths lengths (and thus the edge lengths) for the suffix tree via RMQs on the lcp array.

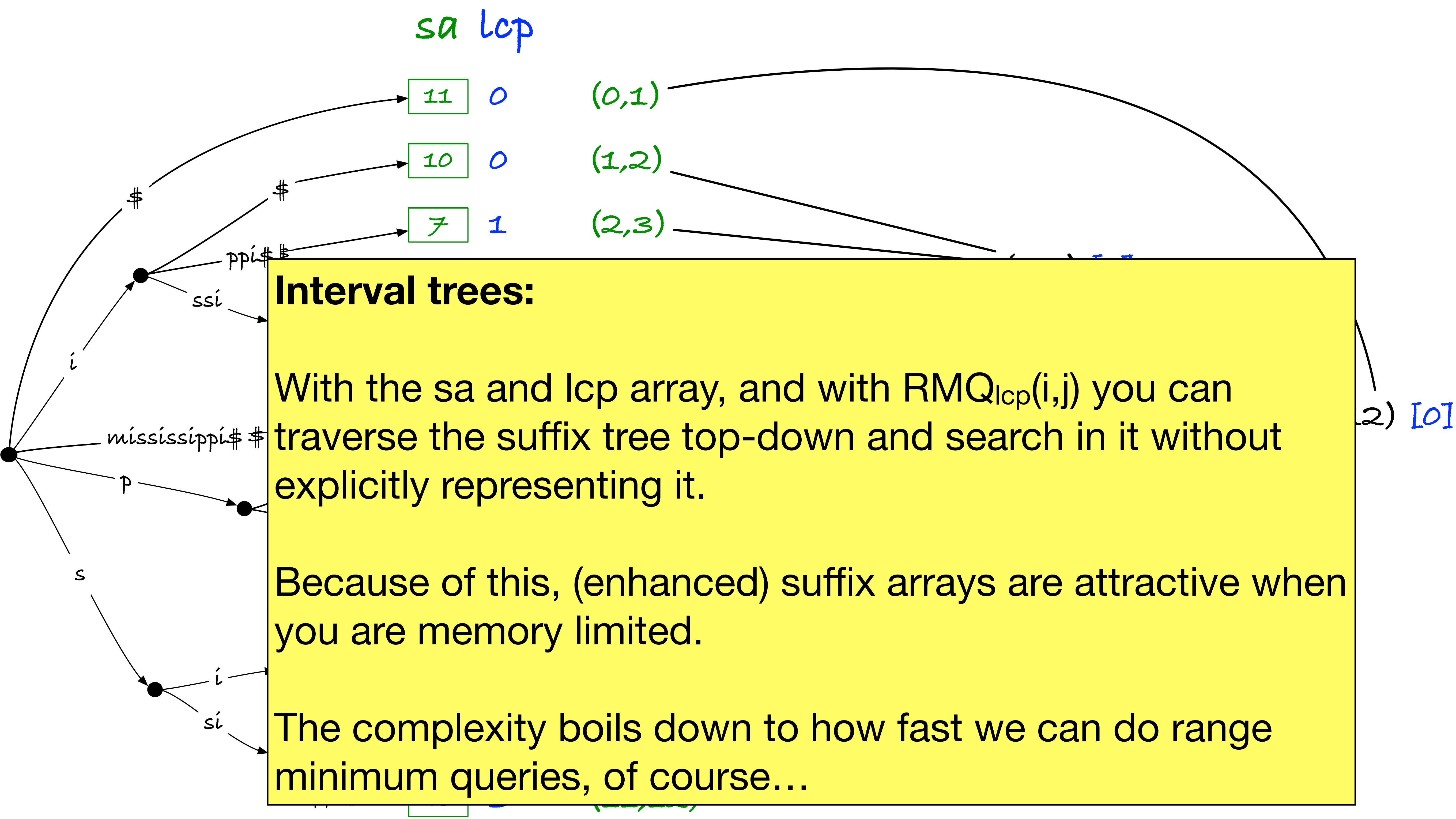
lcp arrays and interval trees



Interval trees



Interval trees



Interval trees:

With the sa and lcp array, and with $RMQ_{lcp}(i,j)$ you can traverse the suffix tree top-down and search in it without explicitly representing it.

Because of this, (enhanced) suffix arrays are attractive when you are memory limited.

The complexity boils down to how fast we can do range minimum queries, of course…

Range Minimum Queries

	Preprocessing	**Lookup**	
Search for each interval	O(1)	O(n)	
Trivial table	$O(n^3)$	O(1)	Precompute T[i,j] for all (i,j)
Dynamic programming	$O(n^2)$	O(1)	Precompute T[i,j] for all (i,j) using dynamic programming
Power-of-two tables	O(n log n)	O(1)	Precompute $T[i,i+2^k]$ tables (using dynamic programming). With lookup, combine $T[i,i+2^k]$ and $T[j-2^k,j]$
Reduce table	O(n)	O(log n)	Split input in bins of size log n and use previous trick; search in bins when you don't have the table
Use two tables (tricky solution)	O(n)	O(1)	Combine the previous solution with precomputed tables for the blocks.

Range Minimum Queries

	Preprocessing	Lookup	

Need to know/nice to know?

You do not need to know how to compute RMQ, but you should probably know that it can be done in <O(n),O(1)> as it is useful for many other algorithms on both strings and trees (e.g. LCA computations).

The tricks are not harder than what we see in class, we just don't have time to cover everything. We see one of the main tricks next week when we compute the lcp from the sa. The Four Russians from algo in bioinformatics is another of them. I am not trying to cheat you, I think you would be able to implement this with sufficient time, but we only have 14 weeks, and we have a lot more to cover.

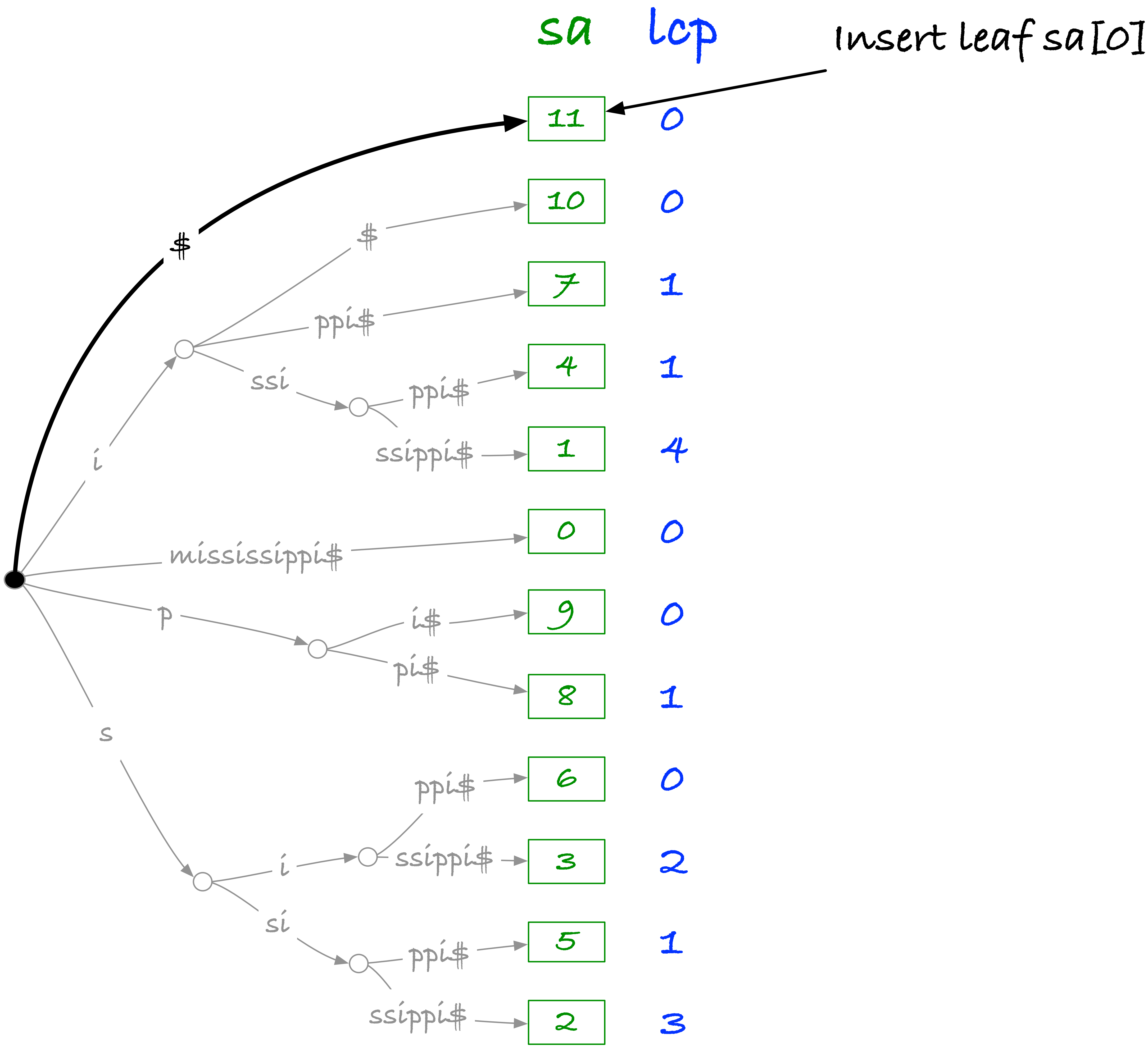
It is an excellent topic for a project or thesis, but we won't explore it further here.

A new suffix tree construction algorithm

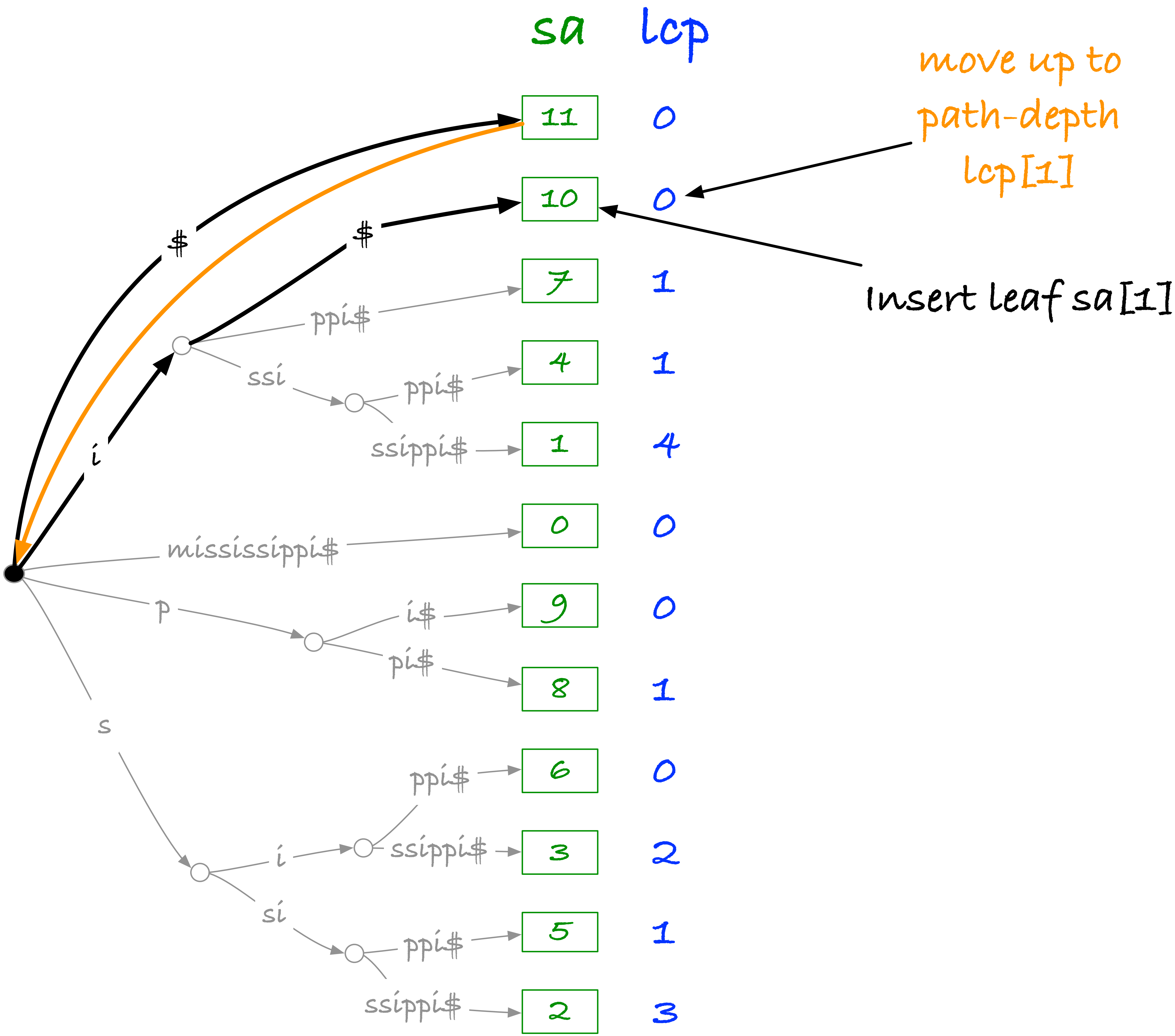
- We are not leaving the interval trees / sa+lcp <=> suffix trees world behind yet, though…
- We will see an algorithm for constructing a suffix tree from the two arrays
- Its main application isn't to *build* suffix trees, though, but to traverse them (without explicitly building them)

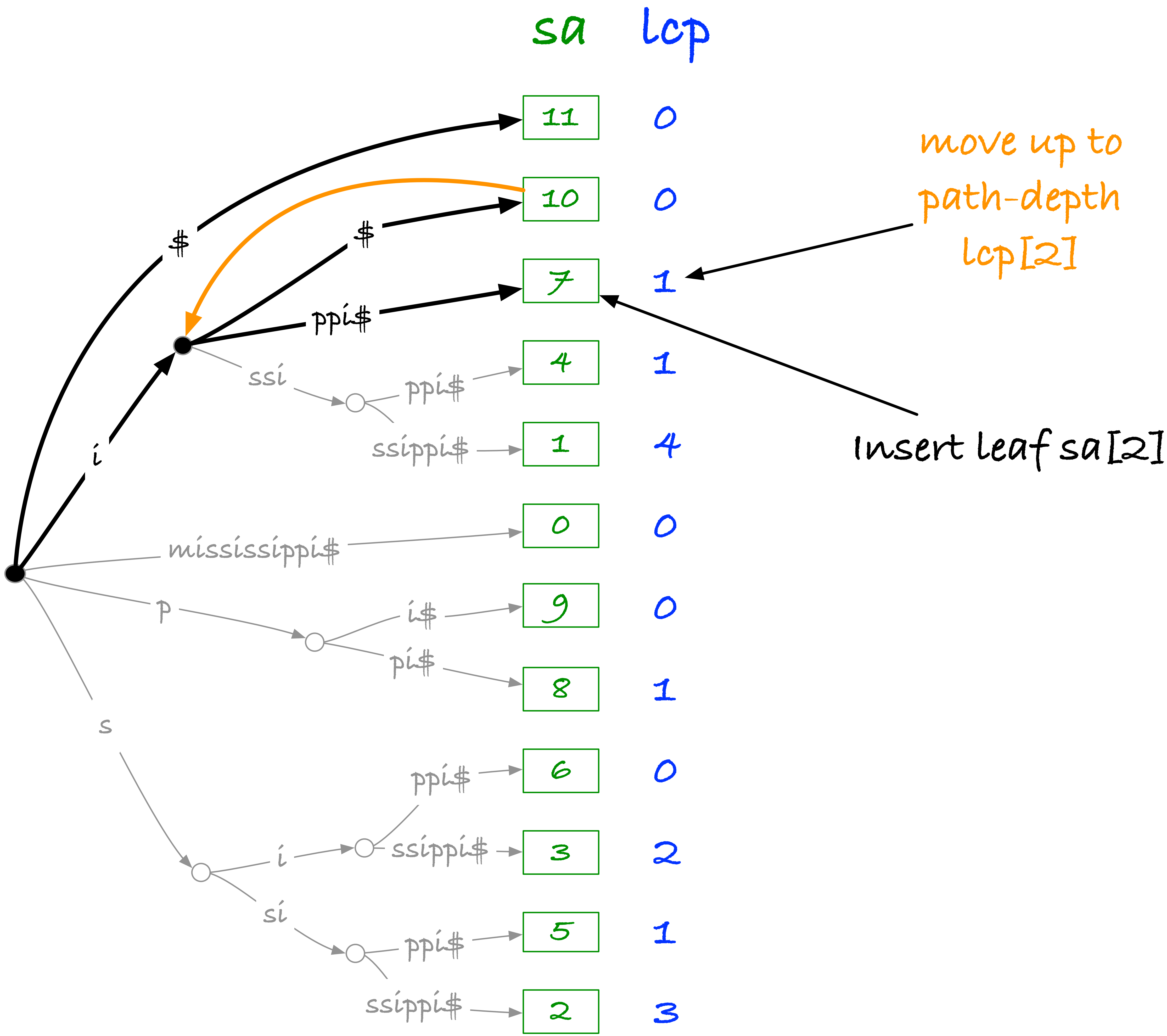
Algorithm overview

- Insert leaves in the order they appear in the suffix array, sa[i] for i = 0, …, n
- Start with leaf sa[i-1] and go up to depth lcp[i], then insert a new leaf there

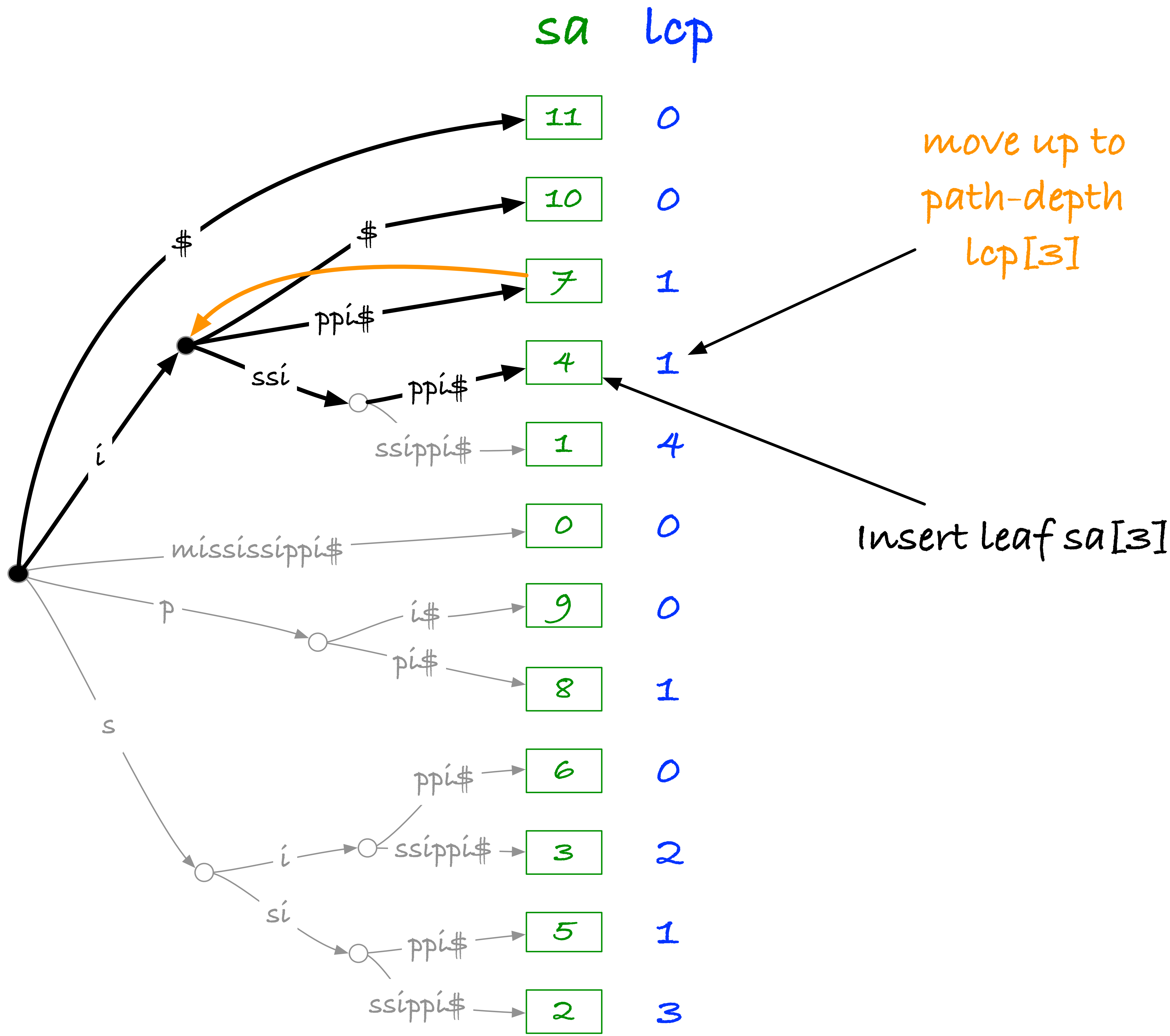


sa
lcp
Insert leaf sa[0]
11
0
10
0
7
1
4
1
1
4
0
0
9
0
8
1
6
0
3
2
5
1
2
3
$
i
$
ppi$
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
i
ppi$
ssippi$
si
ppi$
ssippi$

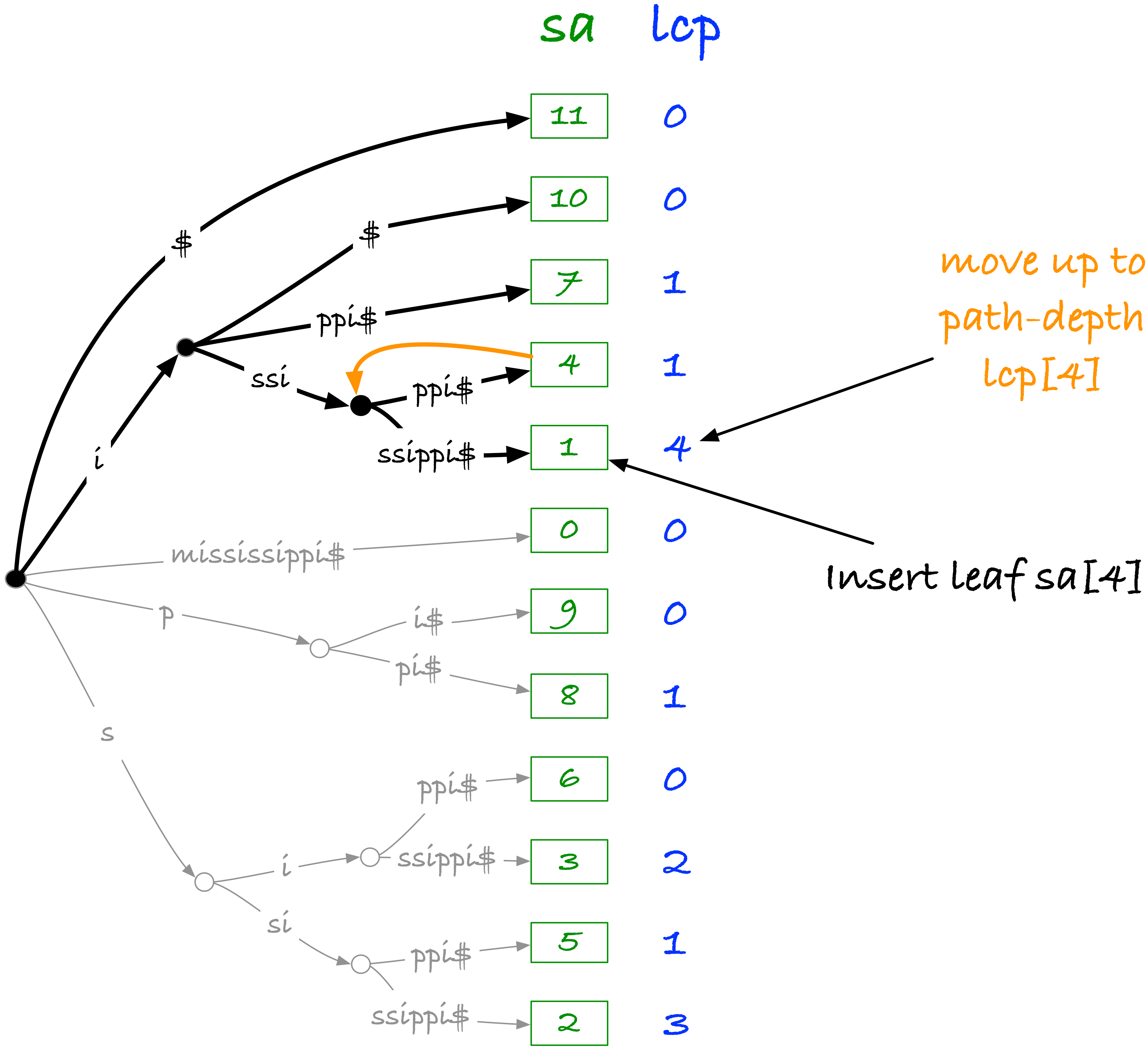

sa
lcp
11
0
10
0
7
1
4
1
1
4
0
0
9
0
8
1
6
0
3
2
5
1
2
3
$
$
i
ppi$
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
ppi$
i
ssippi$
si
ppi$
ssippi$
move up to
path-depth
lcp[1]
Insert leaf sa[1]

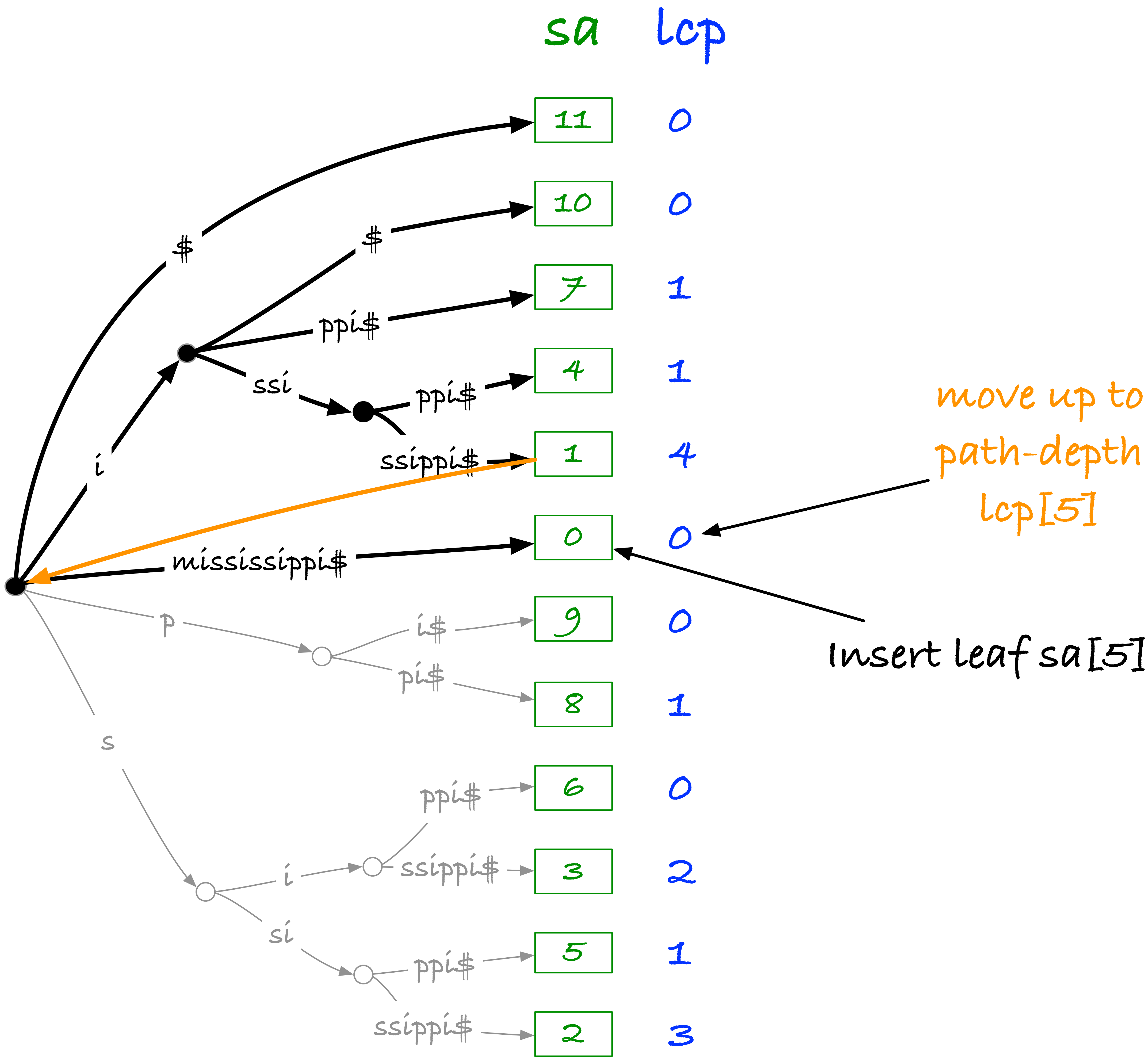


sa
lcp
11
0
10
0
7
1
4
1
1
4
0
0
9
0
8
1
6
0
3
2
5
1
2
3
move up to
path-depth
lcp[2]
Insert leaf sa[2]
$
$
ppi$
i
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
ppi$
i
ssippi$
si
ppi$
ssippi$



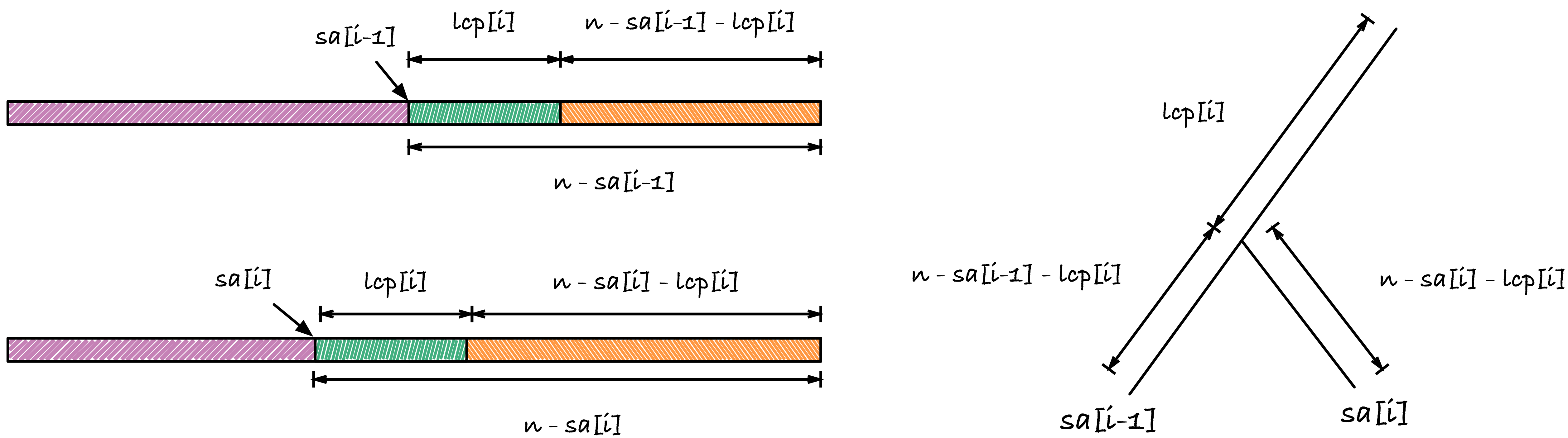
sa
lcp
11
10
7
4
1
0
9
8
6
3
5
2
0
0
1
1
4
0
0
1
0
2
1
3
$
$
ppi$
i
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
ppi$
i
ssippi$
si
ppi$
ssippi$
move up to
path-depth
lcp[3]
Insert leaf sa[3]


sa
lcp
11
0
10
0
7
1
4
1
1
4
0
0
9
0
8
1
6
0
3
2
5
1
2
3
$
$
ppi$
i
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
ppi$
i
ssippi$
si
ppi$
ssippi$
move up to
path-depth
lcp[4]
insert leaf sa[4]



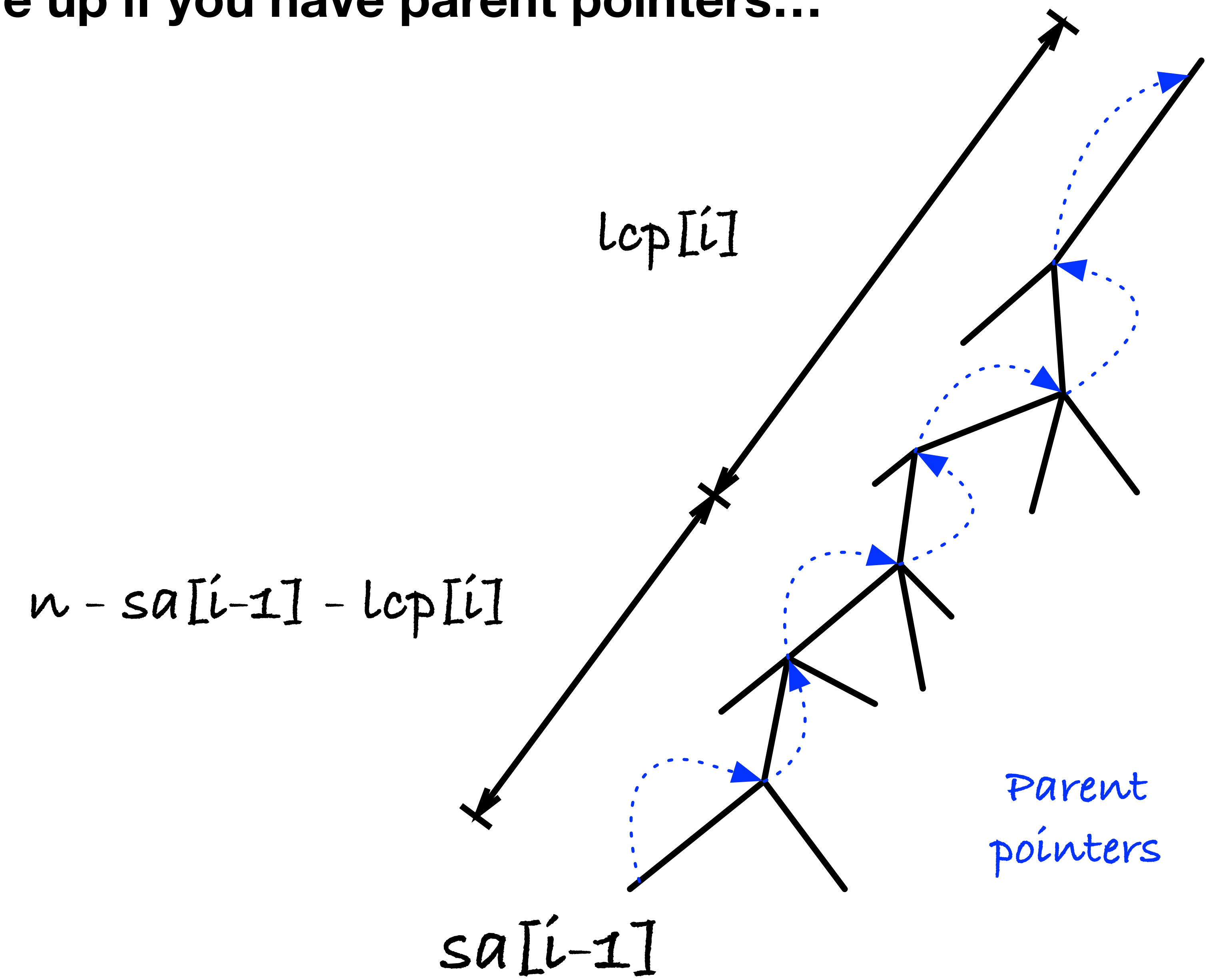
sa
lcp
11 0
10 0
7 1
4 1
1 4
0 0
9 0
8 1
6 0
3 2
5 1
2 3
$
$
ppi$
i
ssi
ppi$
ssippi$
mississippi$
p
i$
pi$
s
ppi$
i
ssippi$
si
ppi$
ssippi$
move up to
path-depth
lcp[5]
Insert leaf sa[5]

Observation

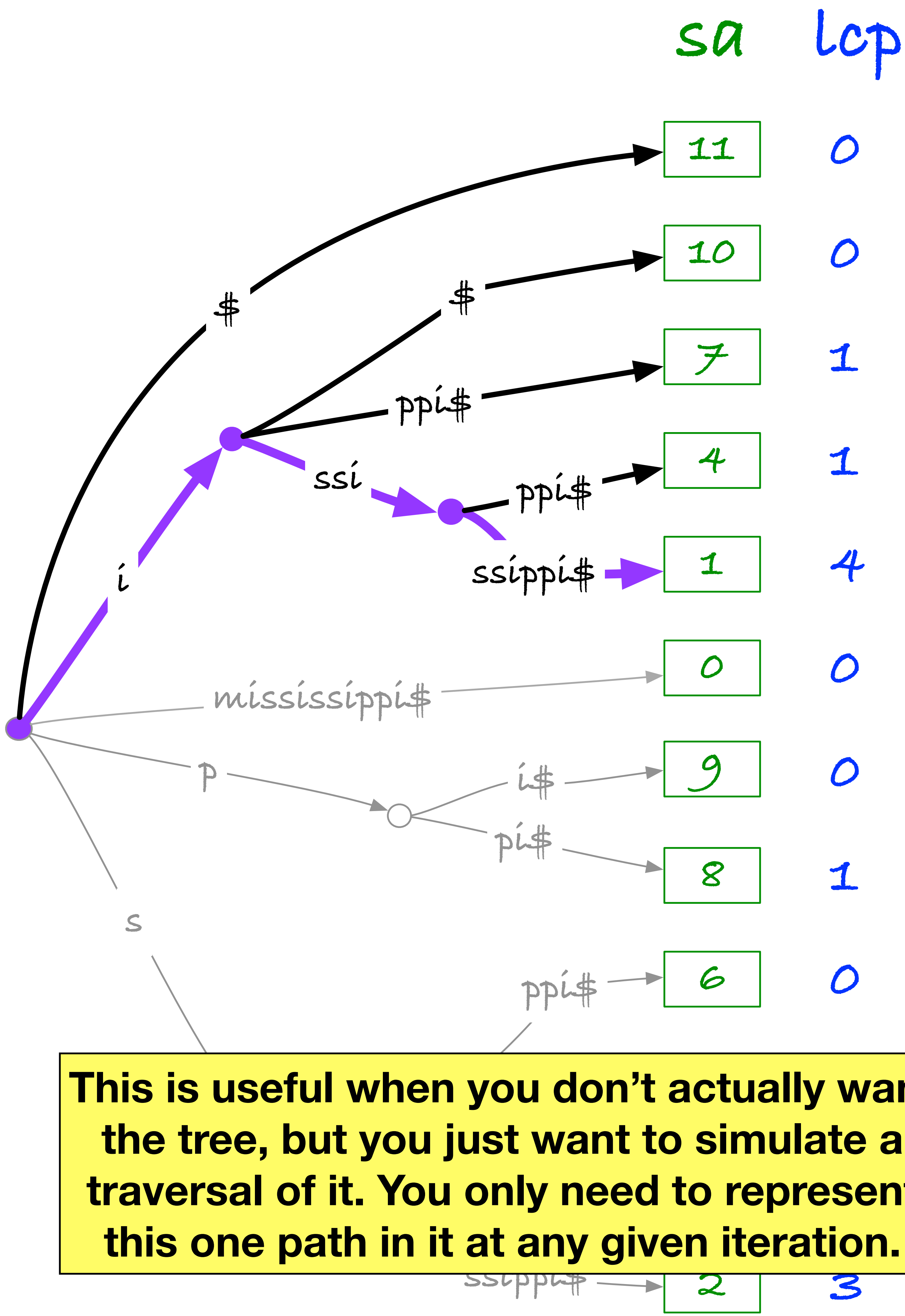


You need to move n - sa[i-1] - lcp[i] up in the tree from sa[i-1] to find where sa[i] should be added

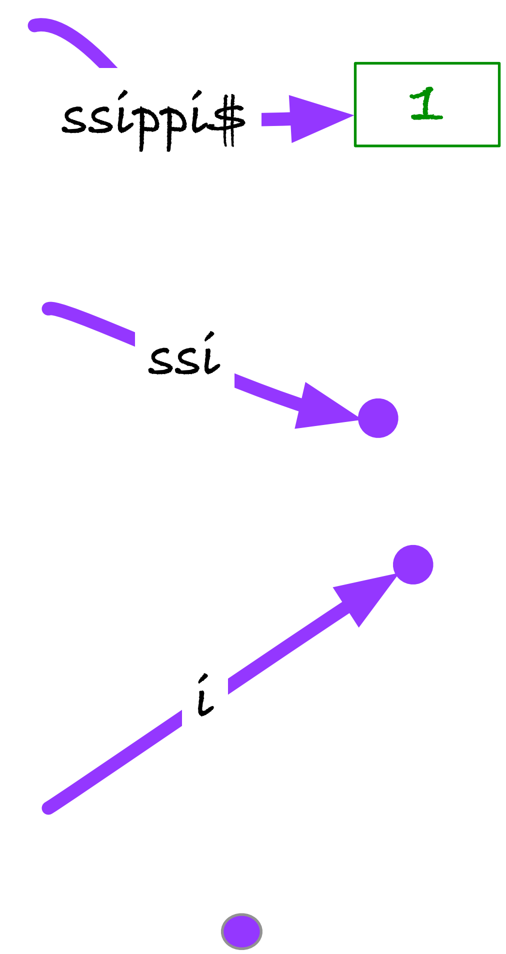
You can move up if you have parent pointers…



You only need to keep track of the distance,
we don't need to look at the edges
(we know they won't match before depth lcp[i])



You can keep the "right path" / "bottom path" in a list (that works like a stack)



This is useful when you don't actually want the tree, but you just want to simulate a traversal of it. You only need to represent this one path in it at any given iteration.

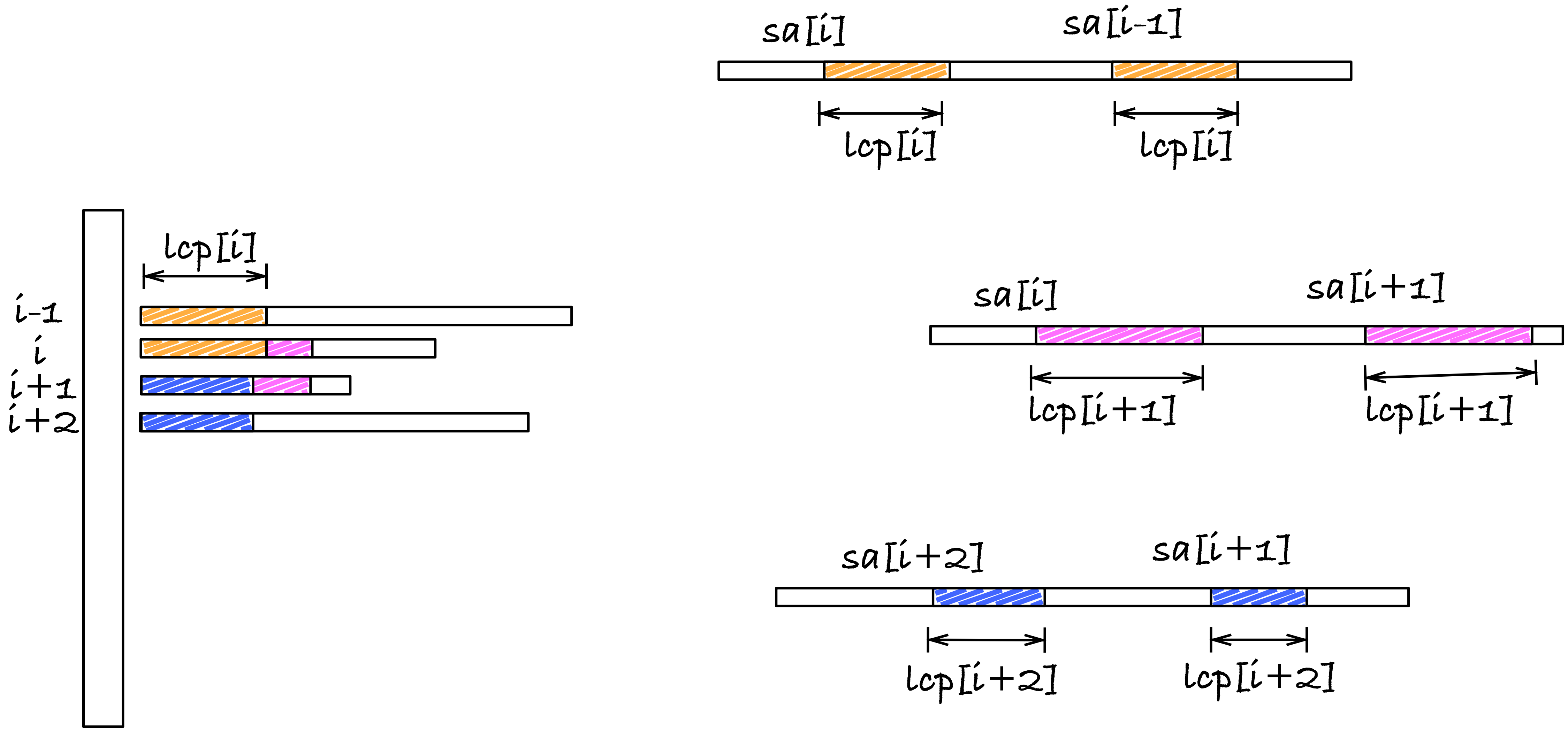
Running time

- We are doing a (virtual) depth first traversal. That takes linear time.
- Once we have moved up past a node we never see it again, so we only move past a node once.
- We can’t decrease the node-depth more than we increase it, and we only increase it by one each iteration.

Constructing the lcp-array

Constructing the suffix array next week…

Constructing lcp



Just comparing x[sa[i-1]:] with x[sa[i]:] for all i takes $O(n^2)$
Can we exploit overlapping comparisons?

Constructing lcp

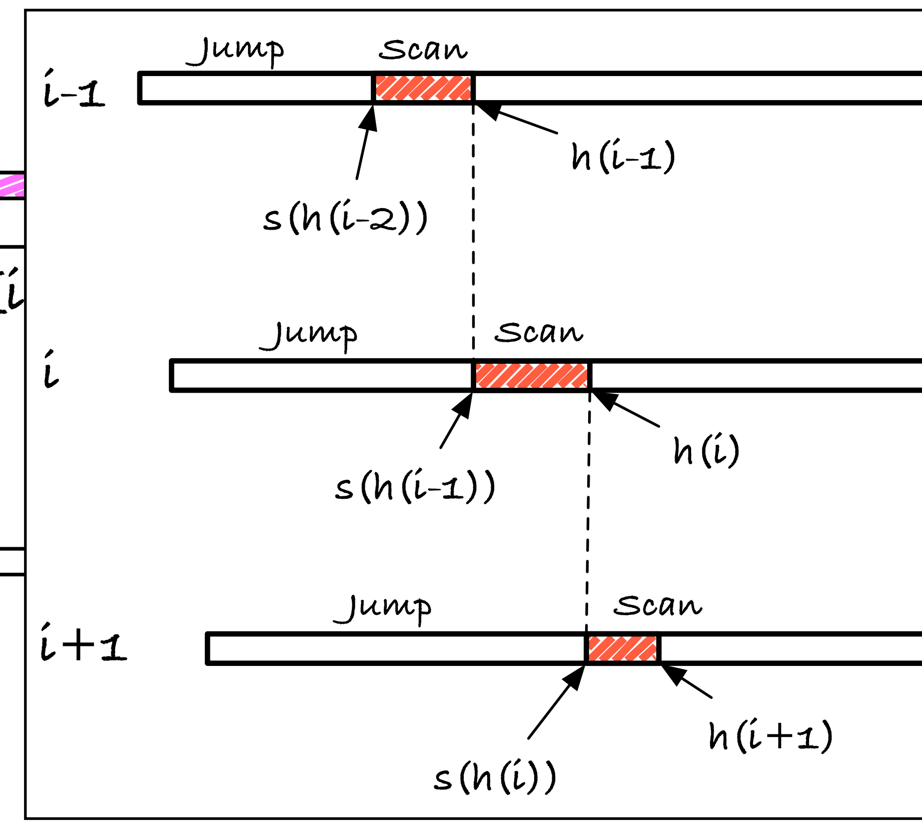
Exploiting already "matched" is the idea behind KMP, border arrays, McCreight, etc. We skip passed it, speeding up the computation.

When we compute lcp[i] we have already looked at x[sa[i-1]:]. Can we exploit that?

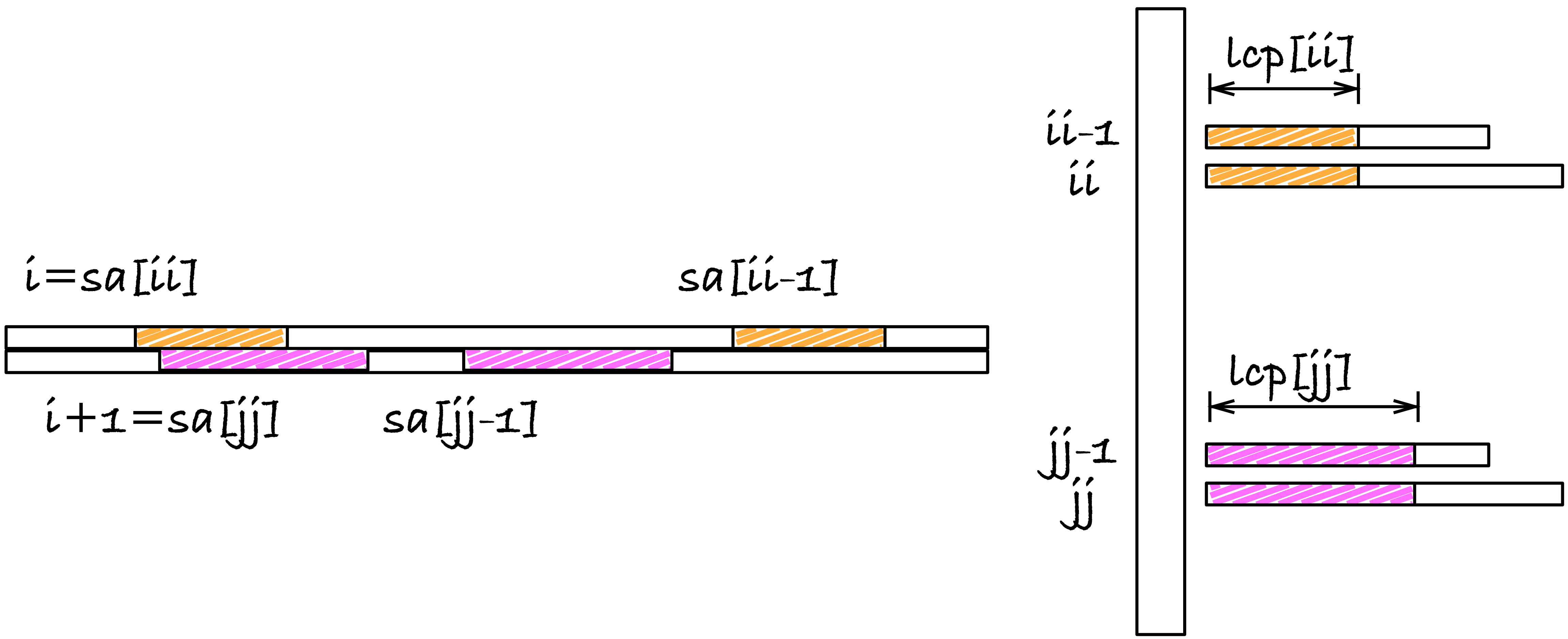
Not really, because we don't know anything about x[sa[i]:]…

BUMMER!!!

Can we exploit overlapping comparisons?

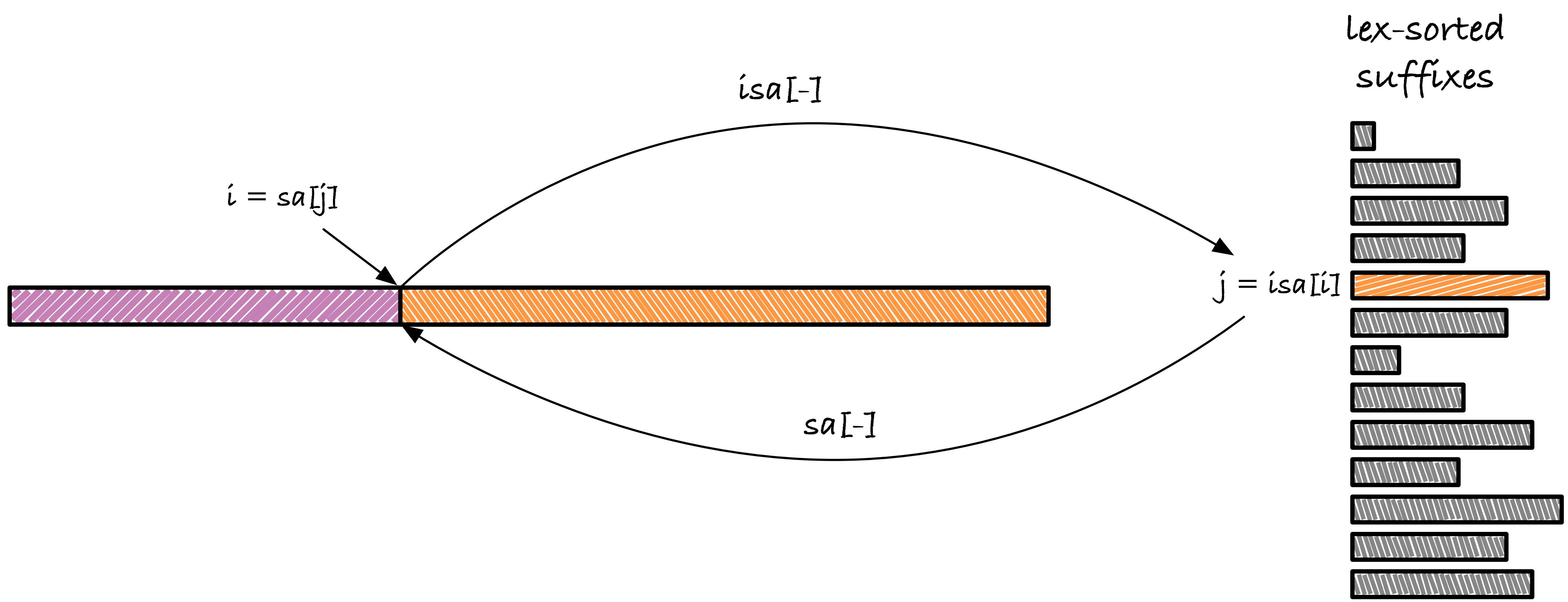


Exploiting prior scans



If we can't go sa[i] to sa[i+1], maybe we can go x[i:] to x[i+1:]…

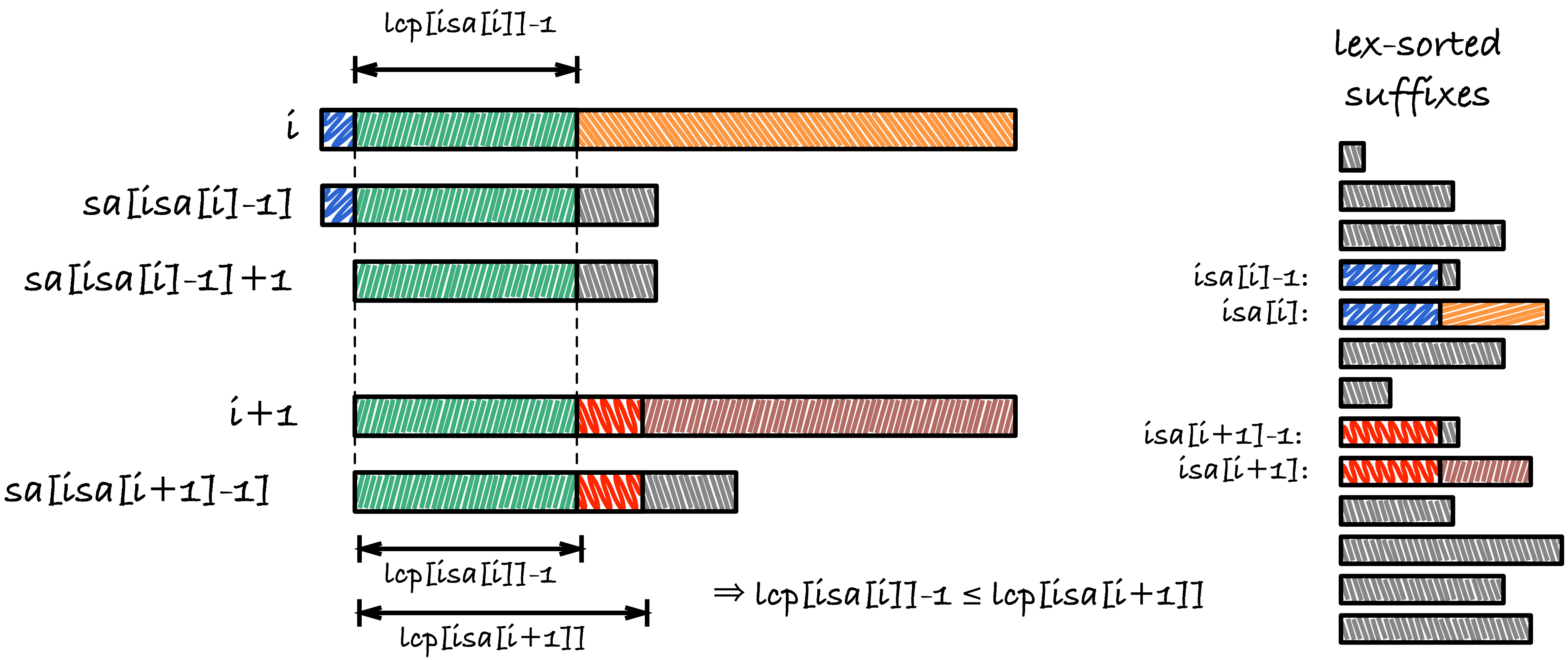
Inverse suffix array



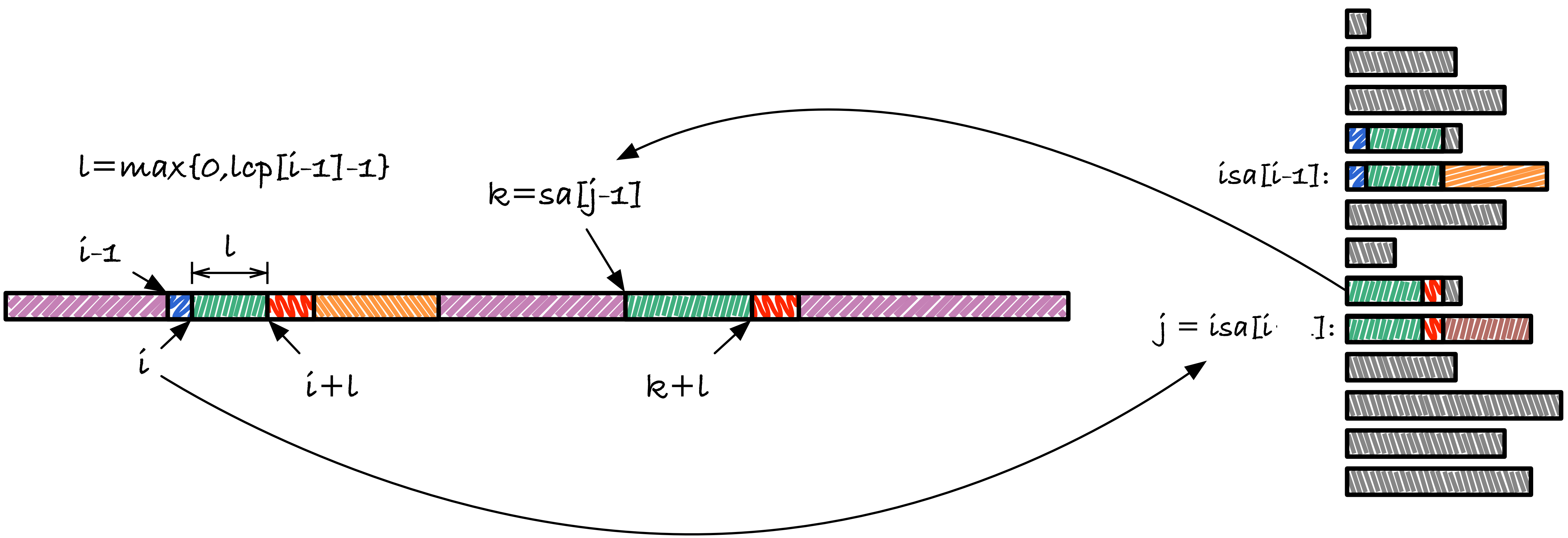
Also useful for comparing strings:

`x[i:]` $\iff$ `x[j:]`

`isa[i]` $\iff$ `isa[j]`

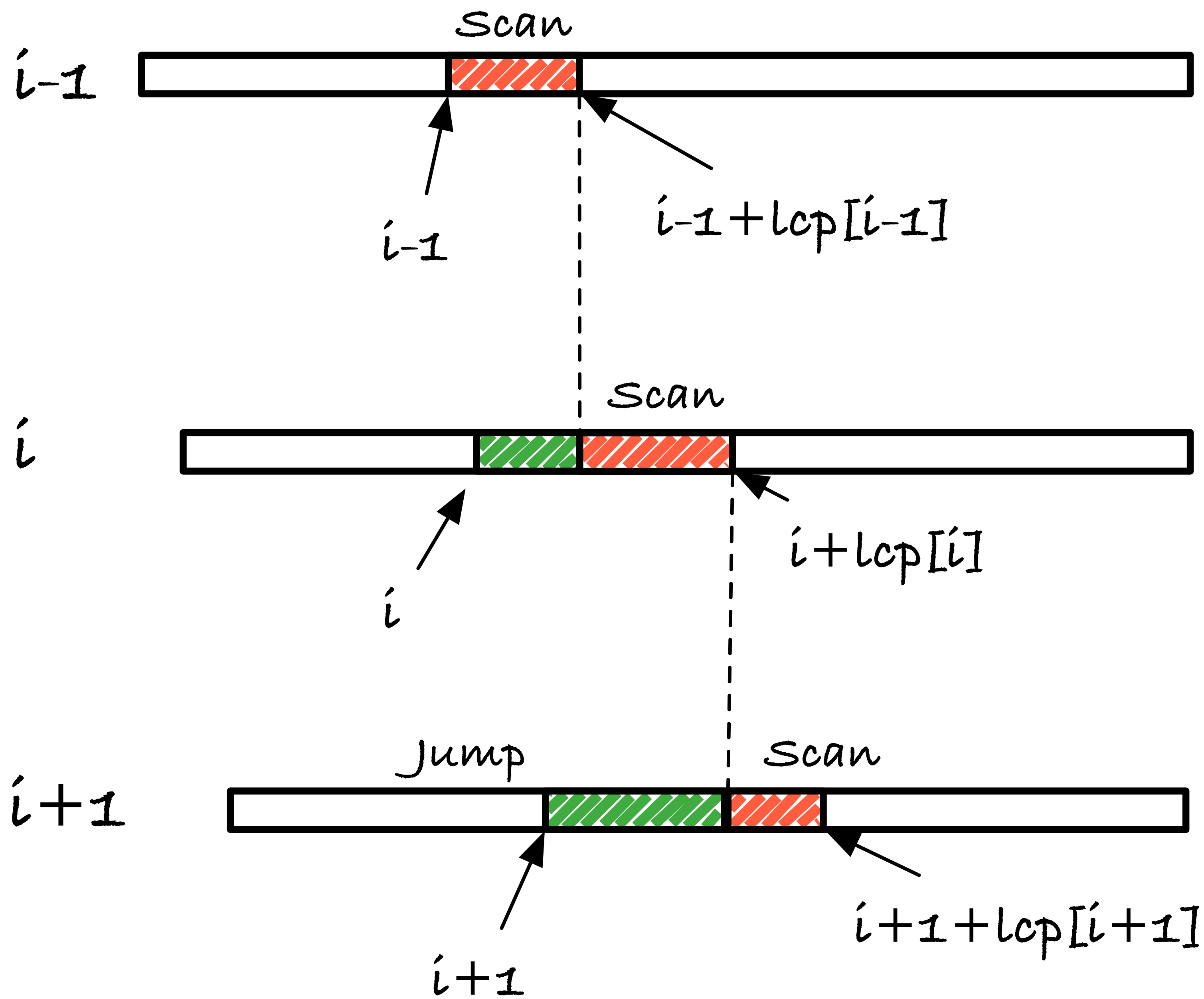


lcp[isa[i]]-1
i
sa[isa[i]-1]
sa[isa[i]-1]+1
i+1
sa[isa[i+1]-1]
lcp[isa[i]]-1
lcp[isa[i+1]]
⇒ lcp[isa[i]]-1 ≤ lcp[isa[i+1]]
lex-sorted suffixes
isa[i]-1:
isa[i]:
isa[i+1]-1:
isa[i+1]:



l=max{0,lcp[i-1]-1}
k=sa[j-1]
isa[i-1]:
i-1
l
i
i+l
k+l
j = isa[i]:

Running time



Running time

- Variable l can never exceed n.
- In each iteration it is at most decreased by one.
- Amortisation gets you the rest of the way.

That's all Folks!