

# Exact pattern matching with multiple patterns

Aho-Corasick

# Searching for more than one pattern at a time

- If we construct all strings close to a read, wouldn't it be nice to be able to search for all of them at the same time?
- The Aho-Corasick lets us do exactly that
- We just need to put all the patterns in a *trie*

# Do we need to understand this?

- We won't use tries in class (except for a few exercises), but don't tune out now because of that. We will use suffix trees, and you won't understand those if you don't understand tries.
- This week's material is not something we will use directly, but it is here to prepare you for the slightly more complicated material that we will use later.
- You won't *need* any of it—tries do not scale—but you need to train your brain, and this is a warmup exercise.
- (Generally speaking, the entire class is for training you to think about strings, not learn individual algorithms and data structures—you can always google those later. It is all training so you can make your own if you need to later)

# Do we need to understand this?

- Tries are not useful for large strings (too memory consuming). Suffix trees are almost but not quite as bad.
- Suffix trees are at the base of many string algorithms, and you need to understand them to get much beyond an introduction level. They are the most fundamental data structure in the field.
- If you don't know suffix trees, you can't read half the literature.
- Suffix trees aren't often used directly, so you don't “need” those either. You just need to think in terms of them.

# Do we need to understand this?

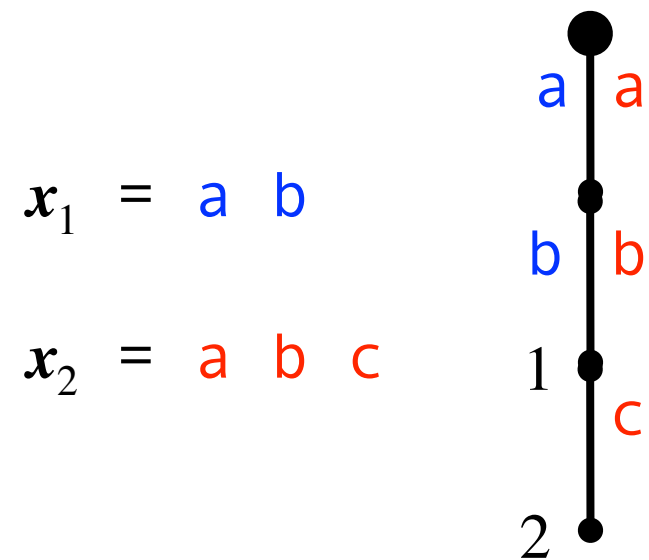
- We mainly want suffix arrays/interval trees, but those are hard to understand without understanding suffix trees, and suffix trees are hard to understand unless you understand tries (that are pretty simple in themselves).
- Interval trees are suffix arrays that pretend to be suffix trees, but we would need a few more weeks to get through those
  - I can give you 10 min explanation, but the details get complicated
- You use suffix arrays and interval trees and a handful of other data structures in implementations. You use the way of thinking you get from tries and suffix trees when you develop algorithms.

# Do we need to understand this?

- Do not think of this week as learning a data structure and an algorithm.
- You will probably never need tries or Aho-Corasick in your life.
- Tries and Aho-Corasick shows you some ways of thinking about strings, and some algorithmic tricks that are generally useful (some also outside of string algorithms).
- It is those tricks you need to learn, and learn to recognise when similar tricks can be used.
  - And that goes for the entire class, although some parts might be useful in themselves.

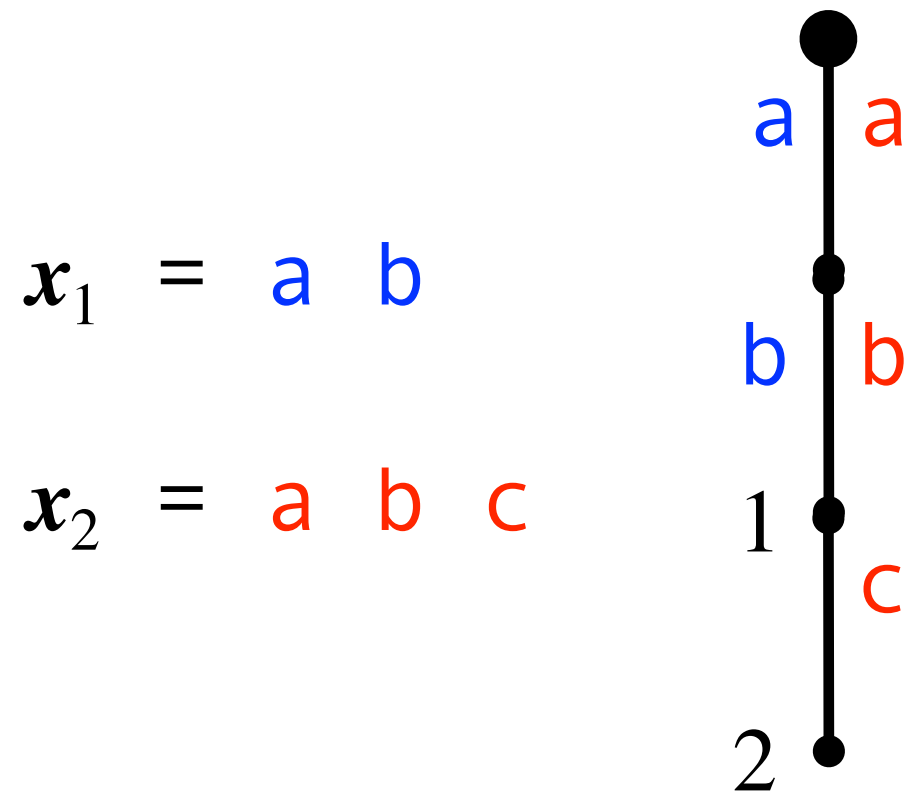
# Trie

... a *trie* is a data structure for storing and *retrieval* of strings ....



# Tries

... a *trie* is a data structure for storing and *retrieval* of strings ....



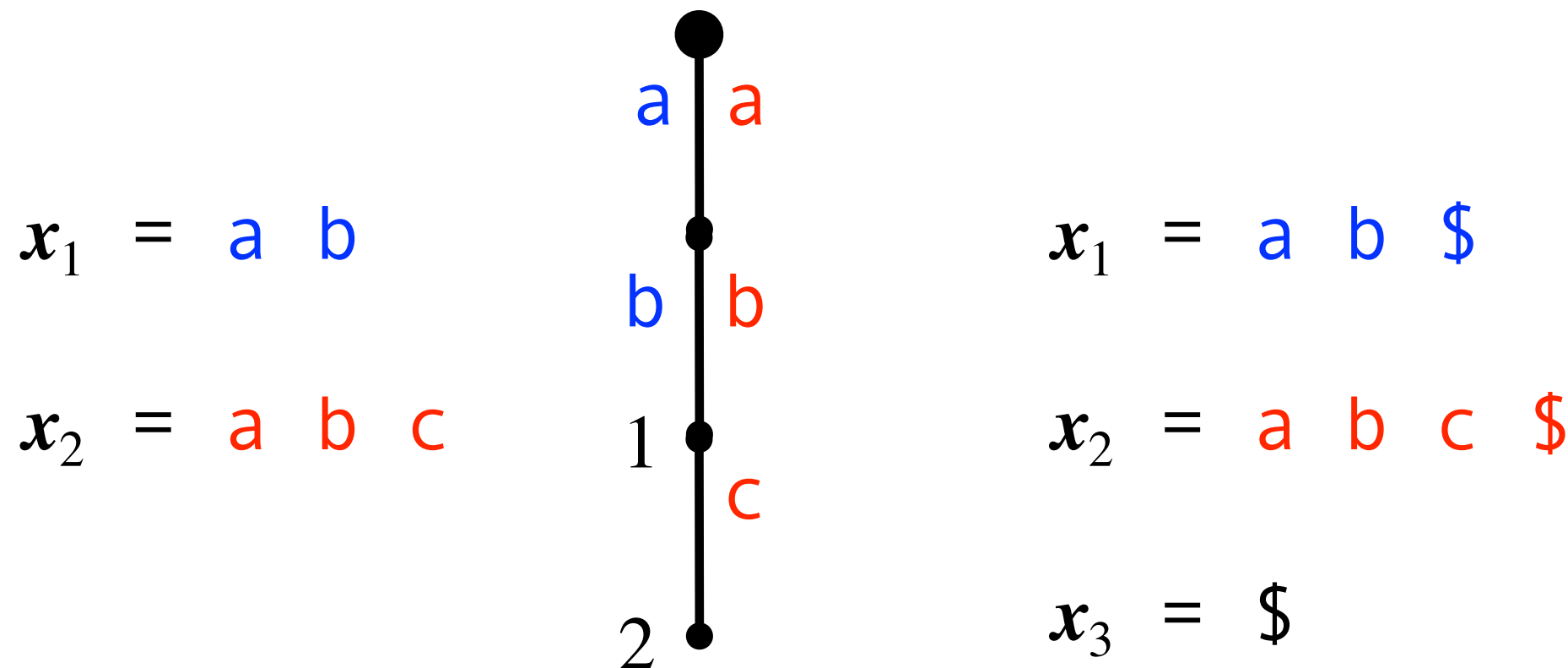
**Observations:** shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?



# Tries

... a *trie* is a data structure for storing and *retrieval* of strings ....

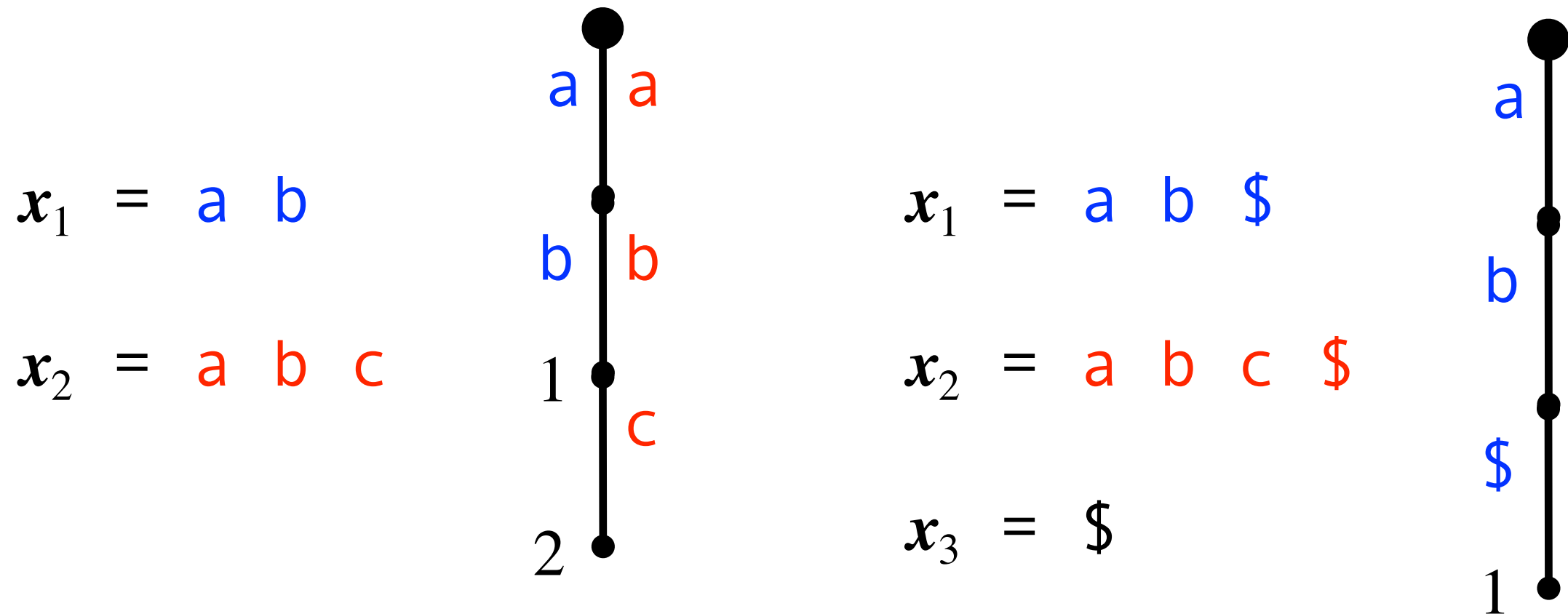


**Observations:** shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?

# Tries

... a *trie* is a data structure for storing and *retrieval* of strings ....

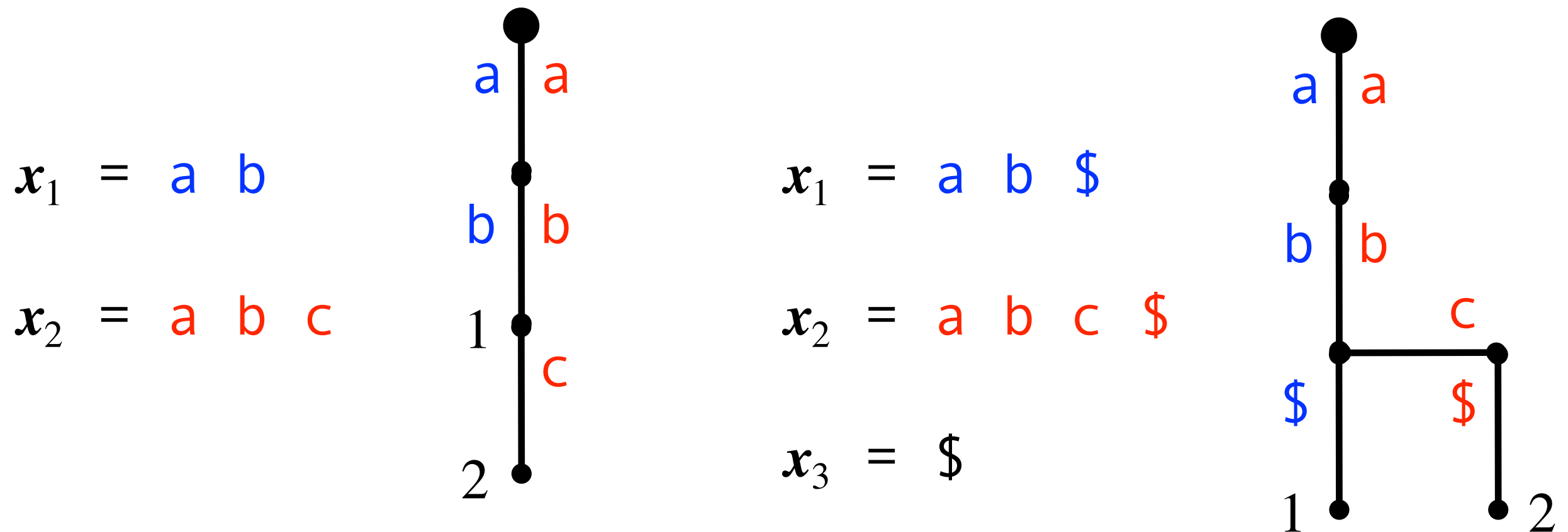


**Observations:** shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?

# Tries

... a *trie* is a data structure for storing and *retrieval* of strings ....

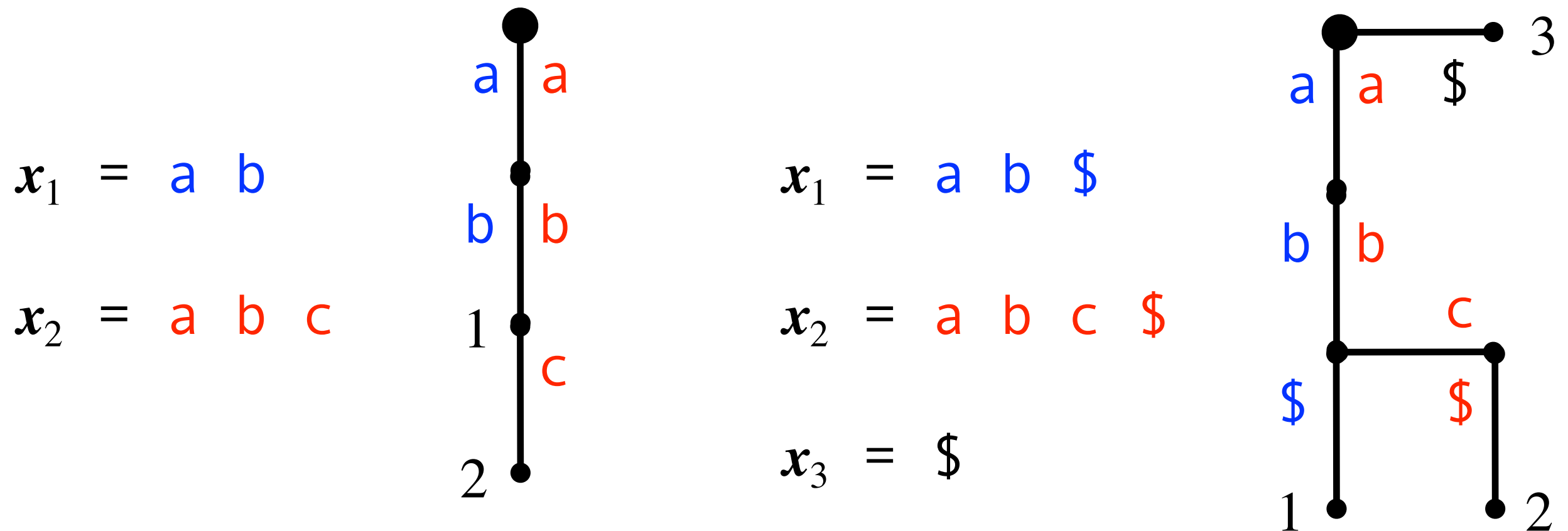


**Observations:** shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?

# Tries

... a *trie* is a data structure for storing and *retrieval* of strings ....



**Observations:** shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?

```
> gsa show trie x y z | idot
```

```
alias idot 'dot -Tpng | kitty icat -align=left'  
kitty icat from kitty terminat
```

**(Do you want to see  
an implementation?)**

# Construction time

- What is the running time for constructing tries?

$$\text{let } n = \sum_i n_i \text{ where } n_i = |x_i|$$

# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

## Patterns:

1: aa

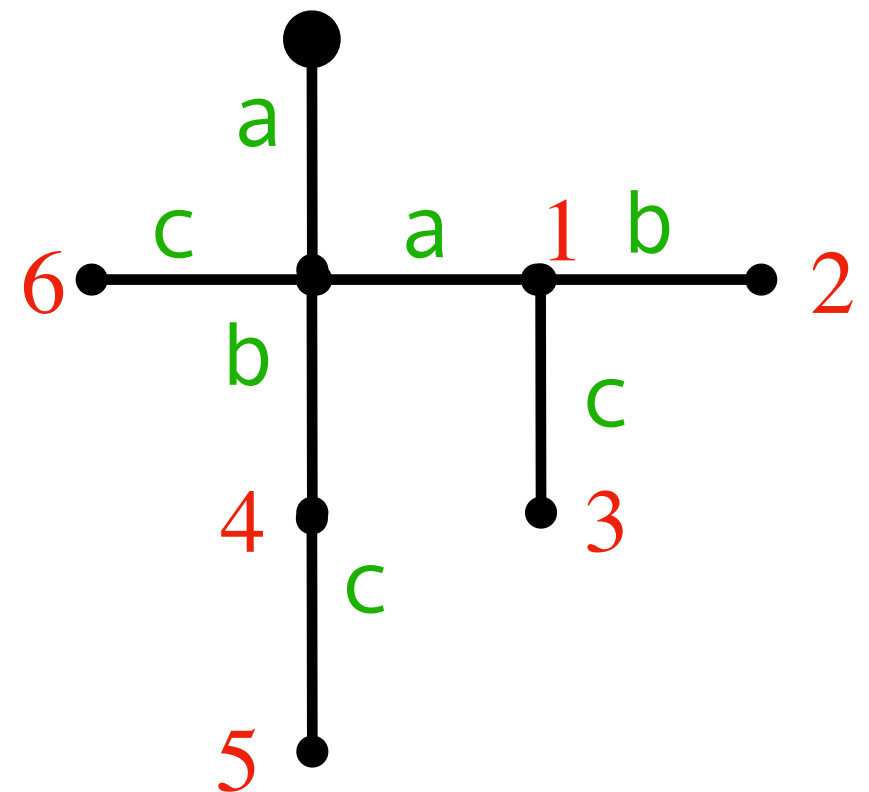
2: aab

### 3: aac

4: ab

5: abc

6: ac

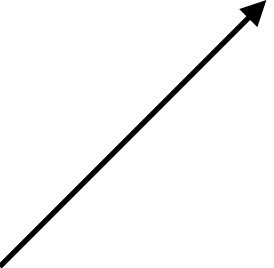


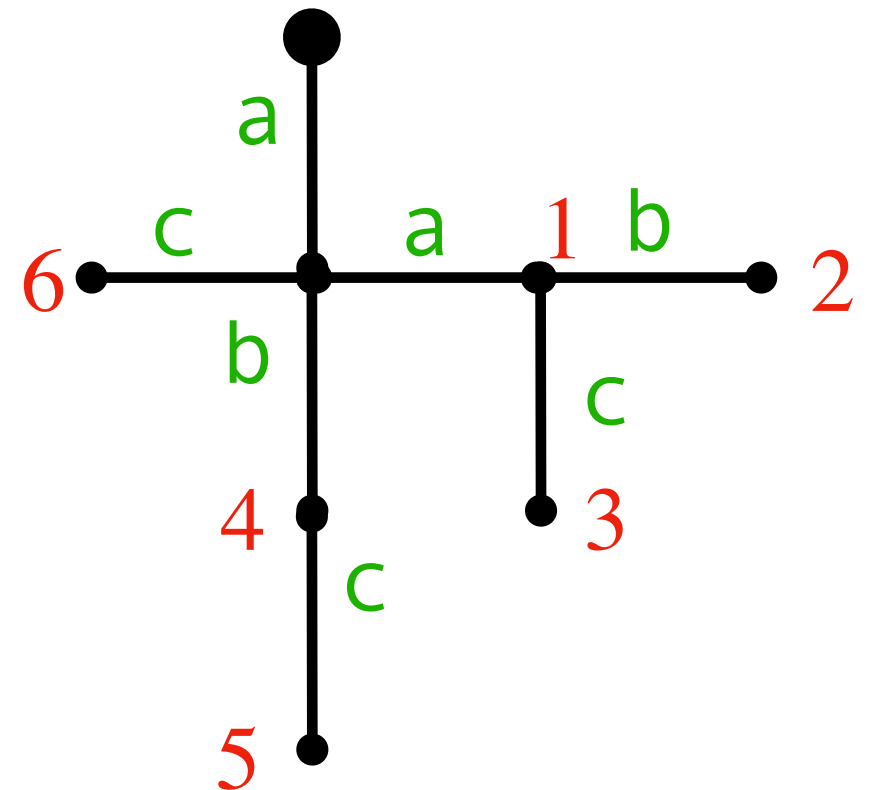


# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abcaabca}$

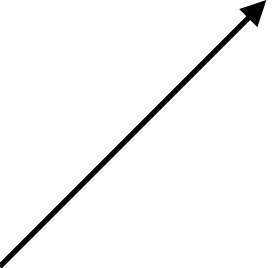
$j$  

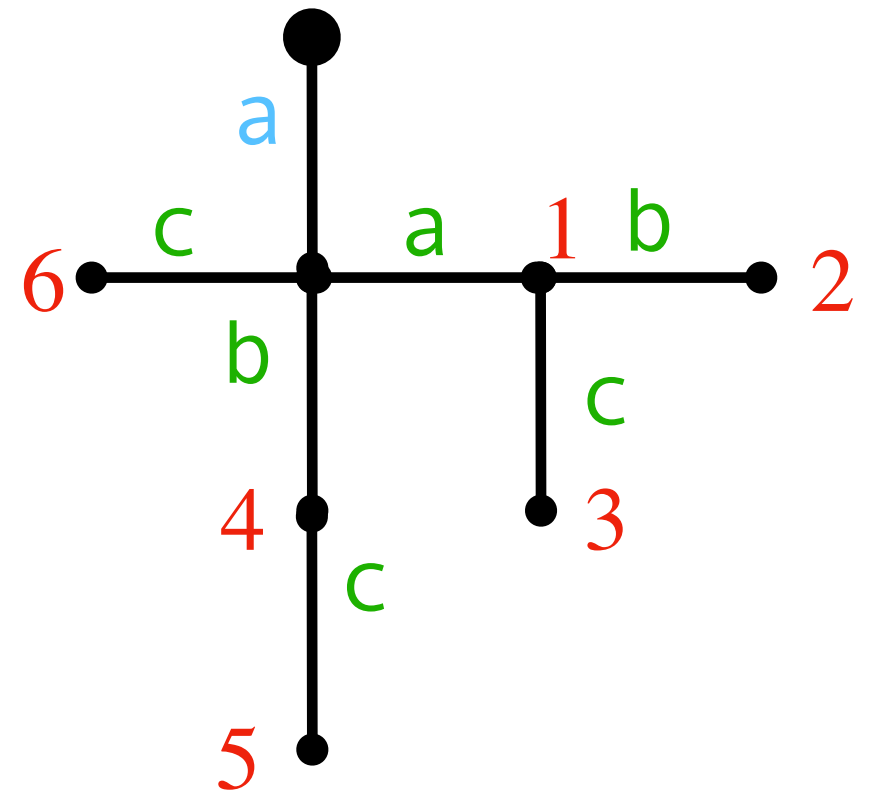


# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abcaabca}$

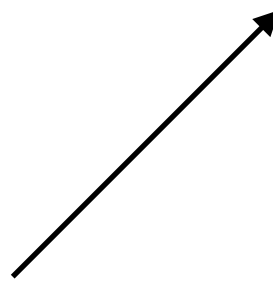
$j$  



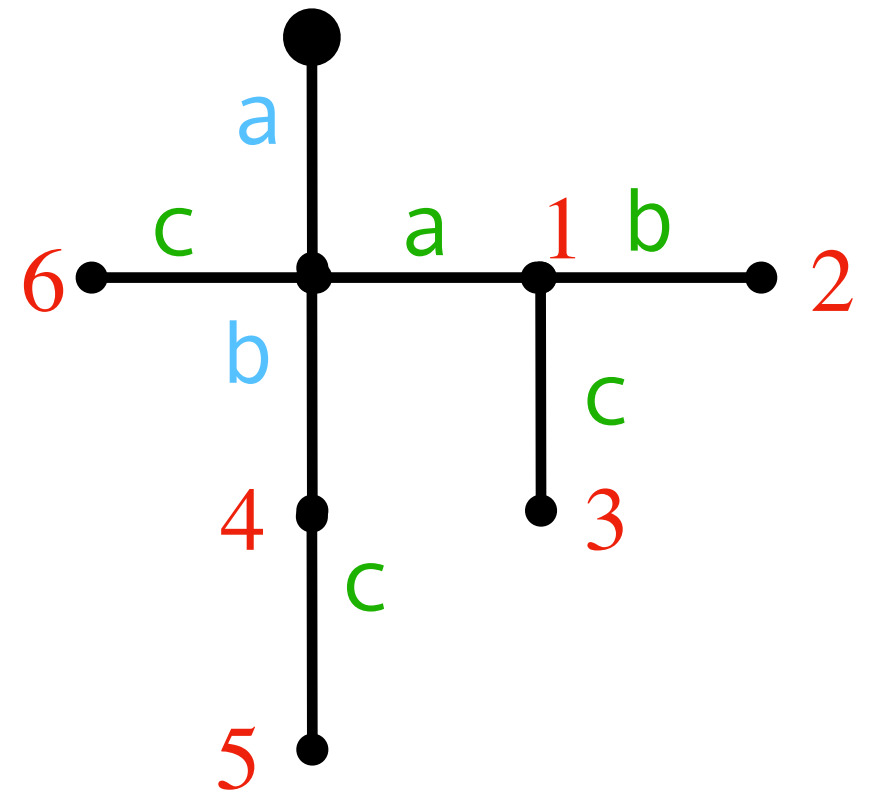
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = abcaabca$

$j$  

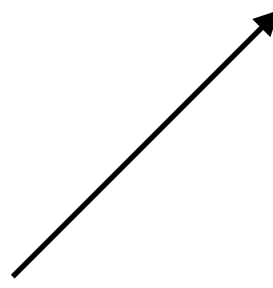
Report 4 at index 0



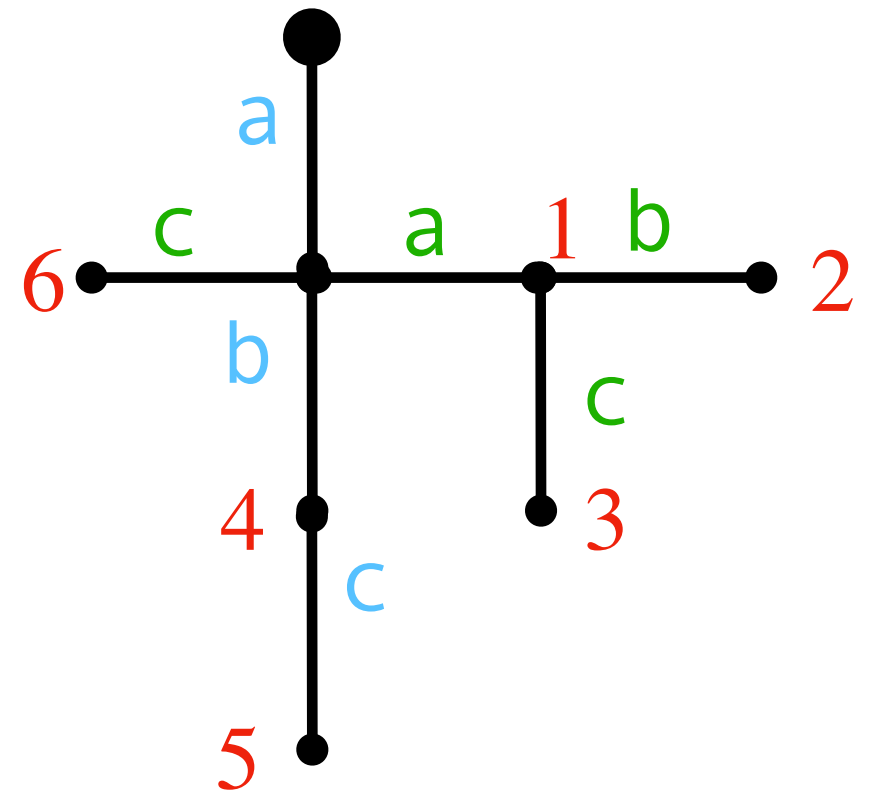
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abcaabca}$

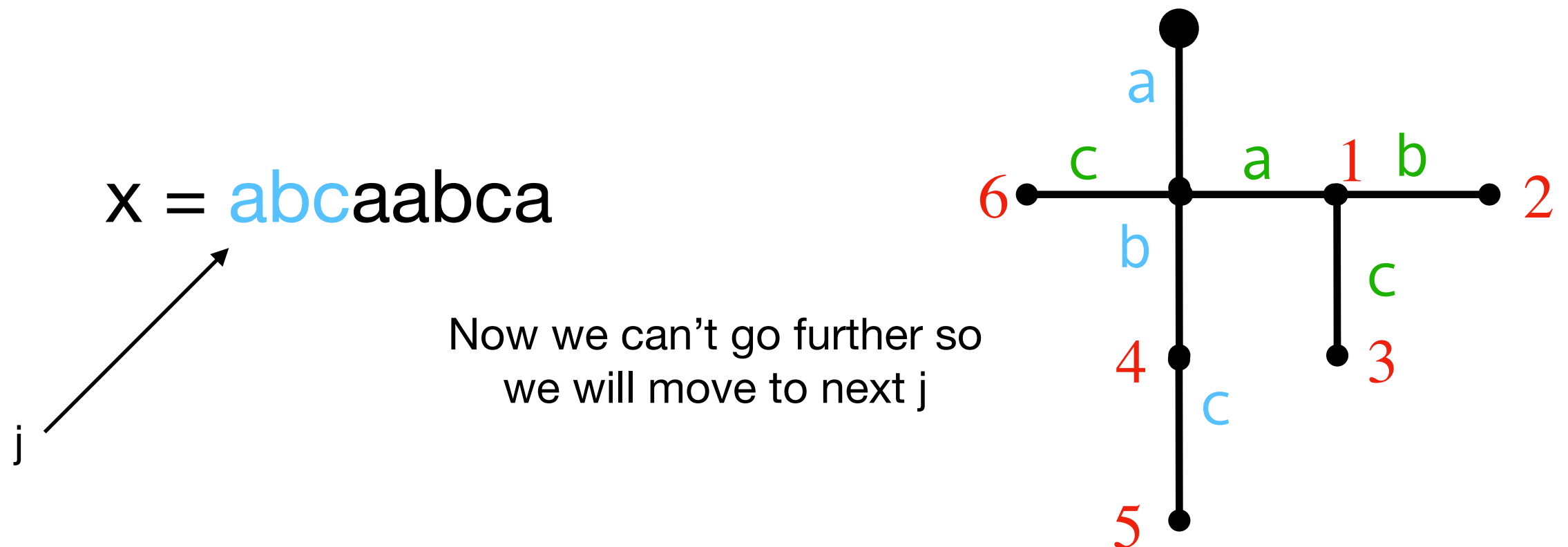
$j$  

Report 5 at index 0



# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches



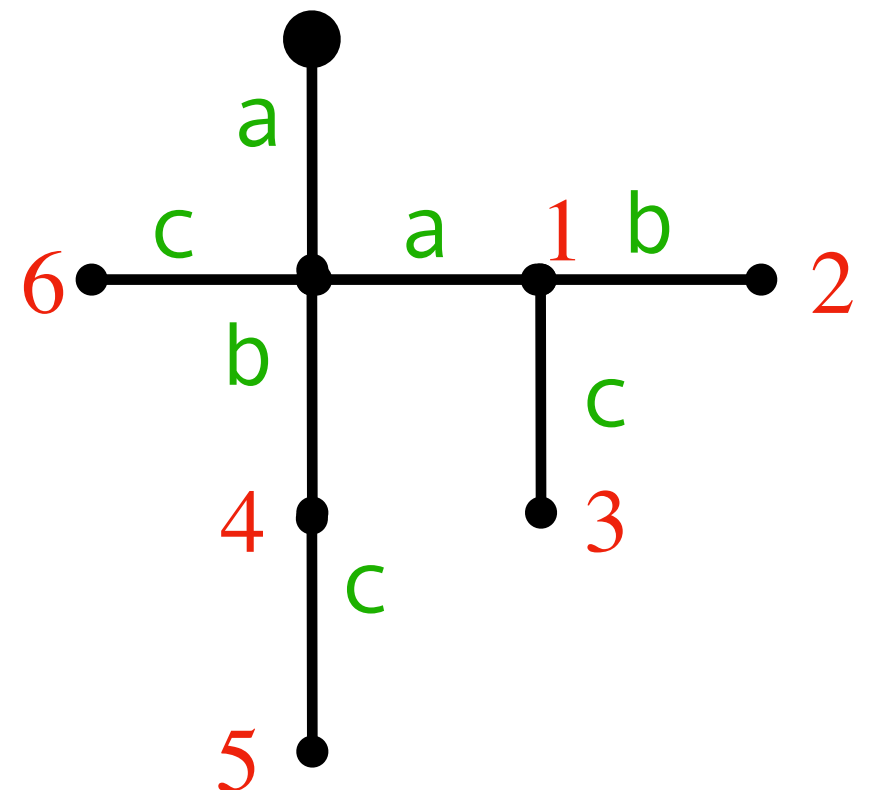
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

**x = abcaabca**

j

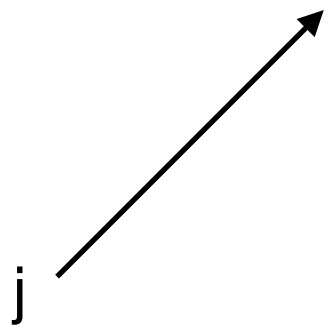
Here we cannot even get started, so again we will move to next j



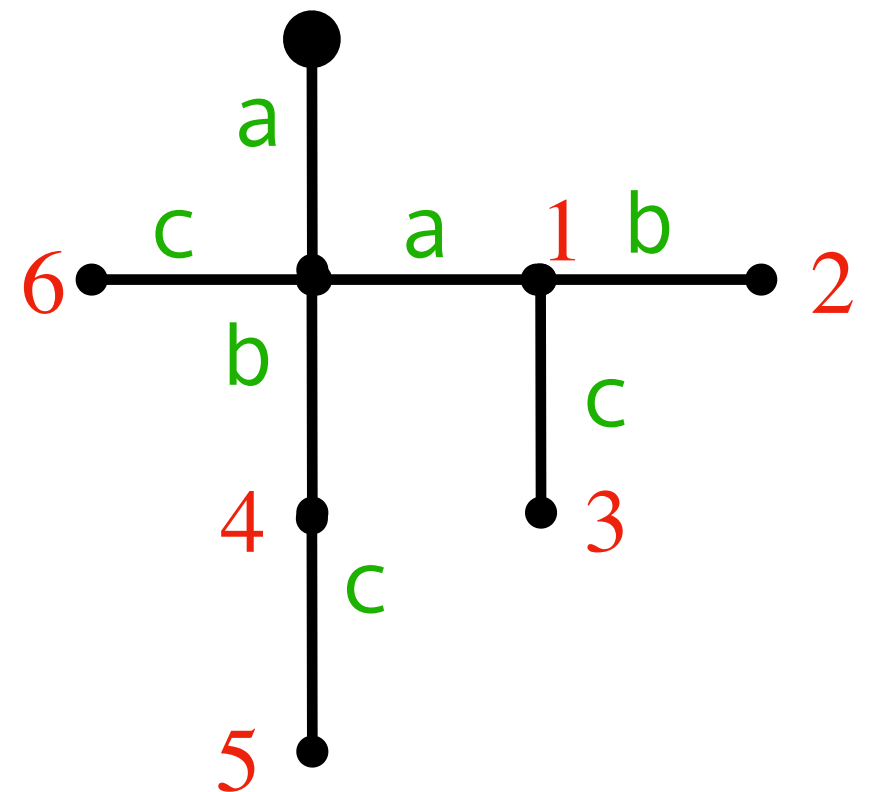
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abcaabca}$



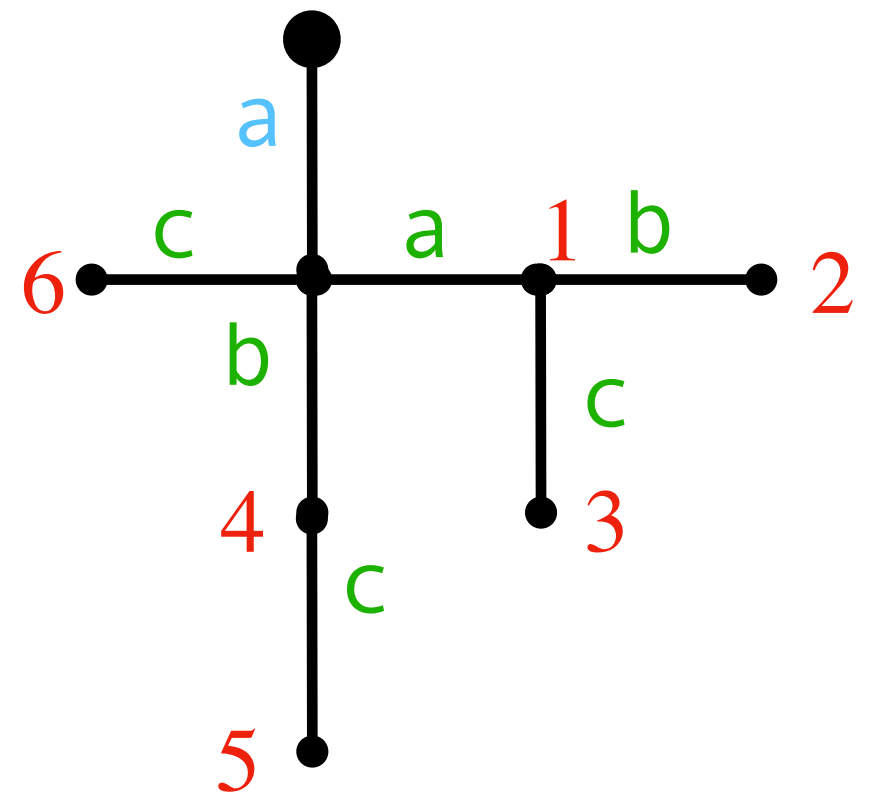
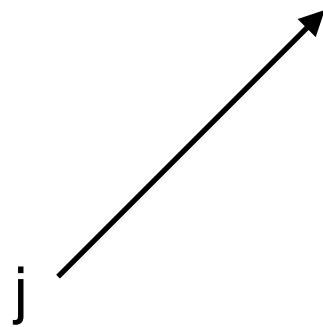
Here we cannot even get started, so again we will move to next  $j$



# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abc}a\text{abca}$

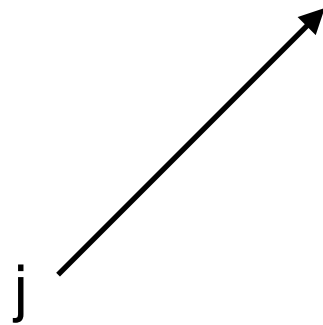




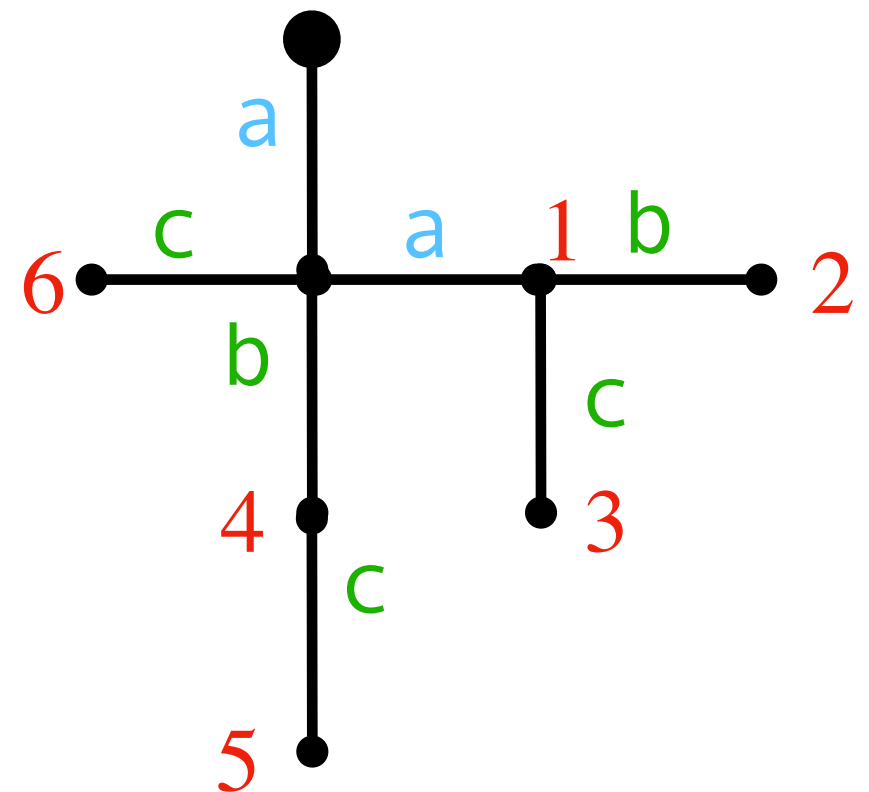
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abc} \textcolor{blue}{aa} \text{bca}$



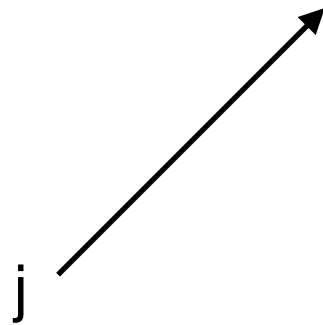
Report 1 at index 3



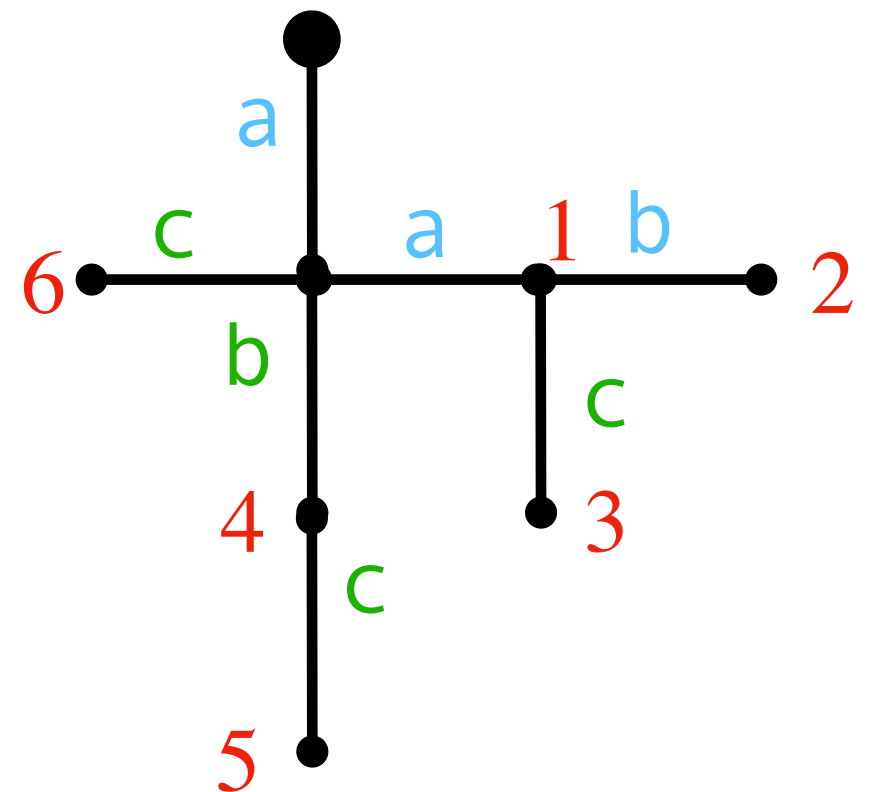
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abc} \textcolor{blue}{a} \textcolor{blue}{a} \textcolor{blue}{b} \textcolor{blue}{c} \textcolor{blue}{a}$



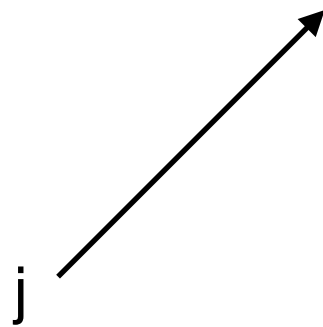
Report 2 at index 3



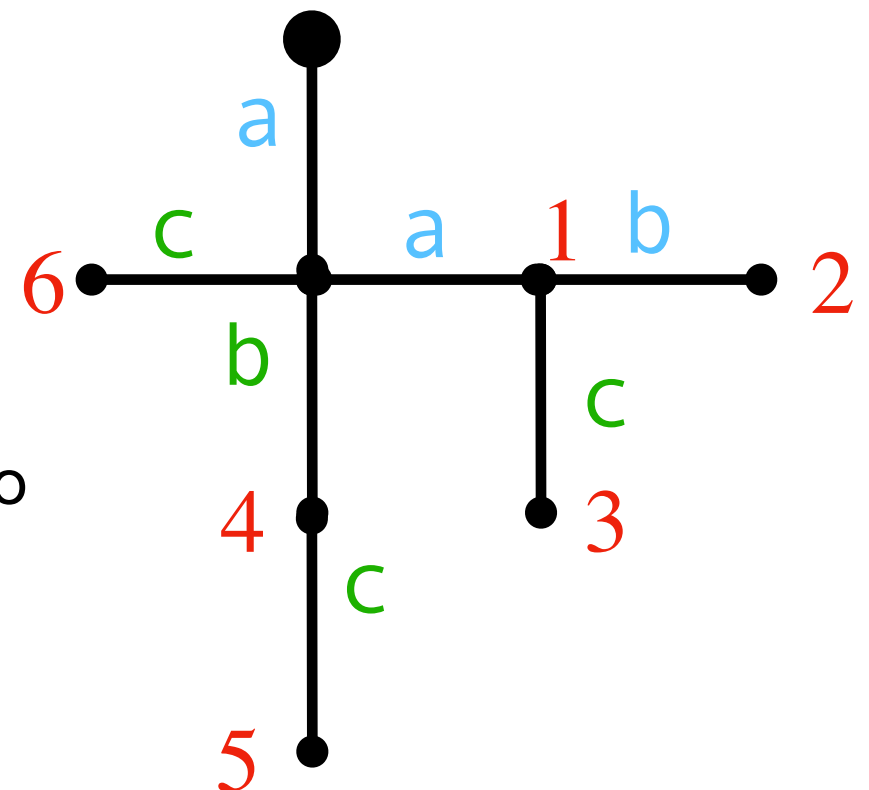
# How do we use a trie for pattern matching?

- For each  $j$  we can search down the trie for  $x[j..n]$  and any patterns we find until we cannot go any further will be matches

$x = \text{abc} \textcolor{blue}{a} \textcolor{blue}{a} \textcolor{blue}{b} \textcolor{blue}{c} \textcolor{blue}{a}$



Now we can't go further so  
we will move to next  $j$



Well, we won't because it is getting boring  
but we could if we wanted to...

# Efficiency?

- We do save some time here compared to search for each pattern separately, but the runtime is not linear.

linear is  $O(n + m)$

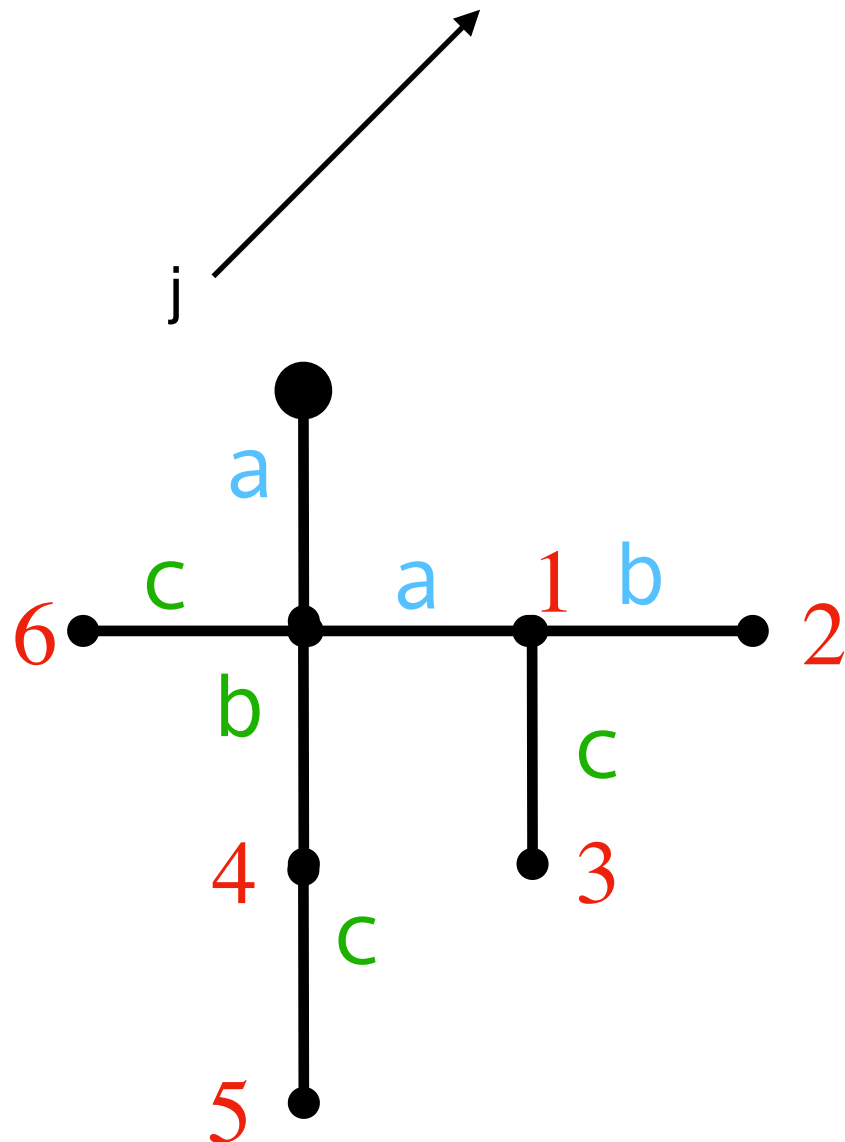
$$m = \sum_{i=0}^{k-1} m_i \quad \text{where } m_i = |p_i|$$

(KMP :  $O(kn + m)$ )

- What is the worst case complexity?
- Give an example

# How do we use a trie for pattern matching?

x = abc**ab**ca



When we got to this point,  
we couldn't go any further.

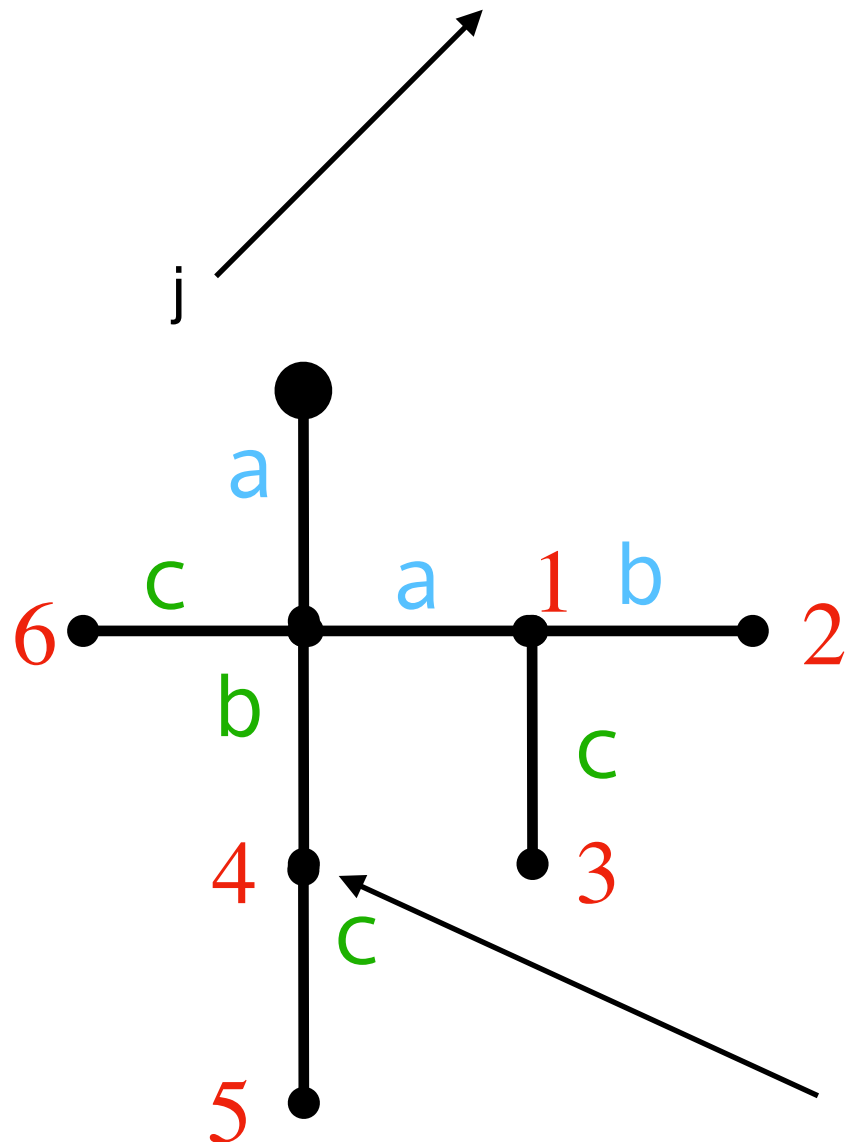
## But why should we go to the root?

We know that 'a' matches

We even know that 'ab' matches  
(because 'ab' is in the trie)

# How do we use a trie for pattern matching?

x = abc**ab**ca



When we got to this point,  
we couldn't go any further.

## But why should we go to the root?

We know that 'a' matches

We even know that 'ab' matches

We should resume searching from here!

# Suffix links

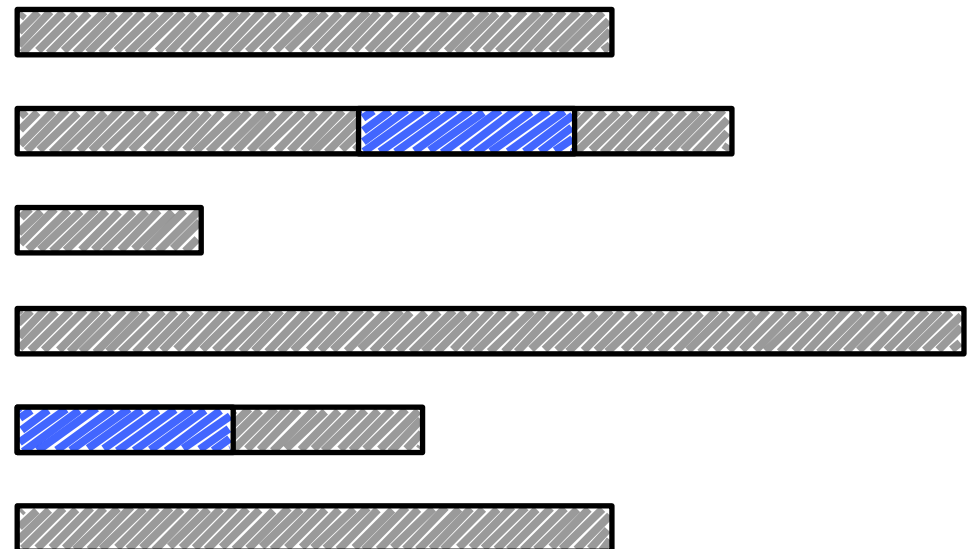
- The solution is to think a bit like Knuth-Morris-Pratt where we had borders — in the trie we will instead have *suffix links*

**Definition** For any prefix  $u$  of a pattern in  $P$ ,  $s(u)$  denotes the longest proper suffix of  $u$  that is a prefix of a pattern in  $P$ .

Borders, in KMP

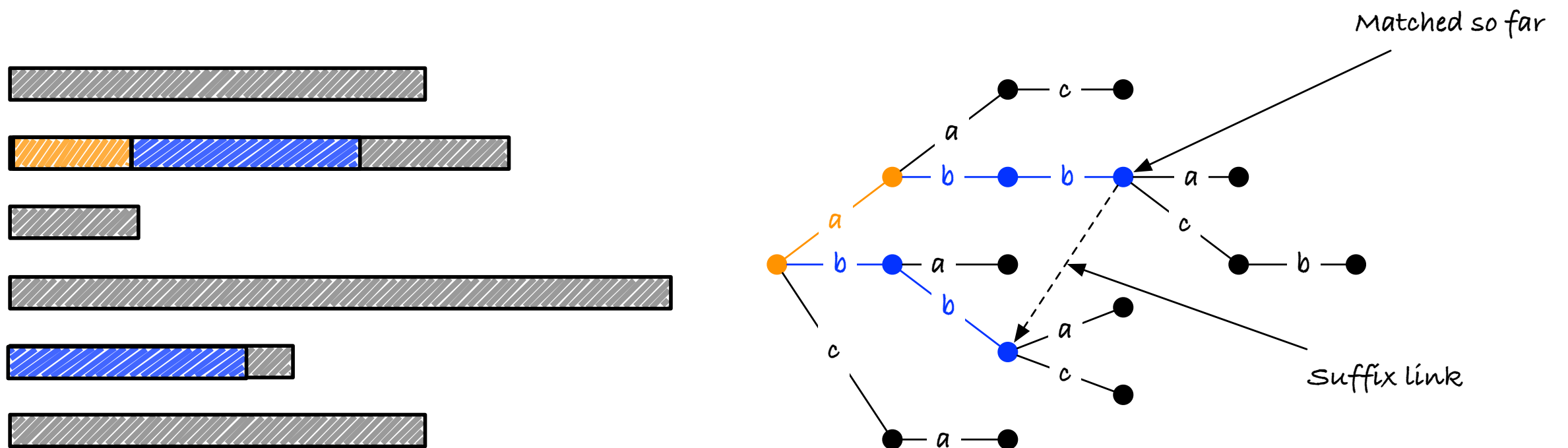


Suffix links in Aho-Corasick



# Suffix links

- We want suffix links to be pointers in the trie as well

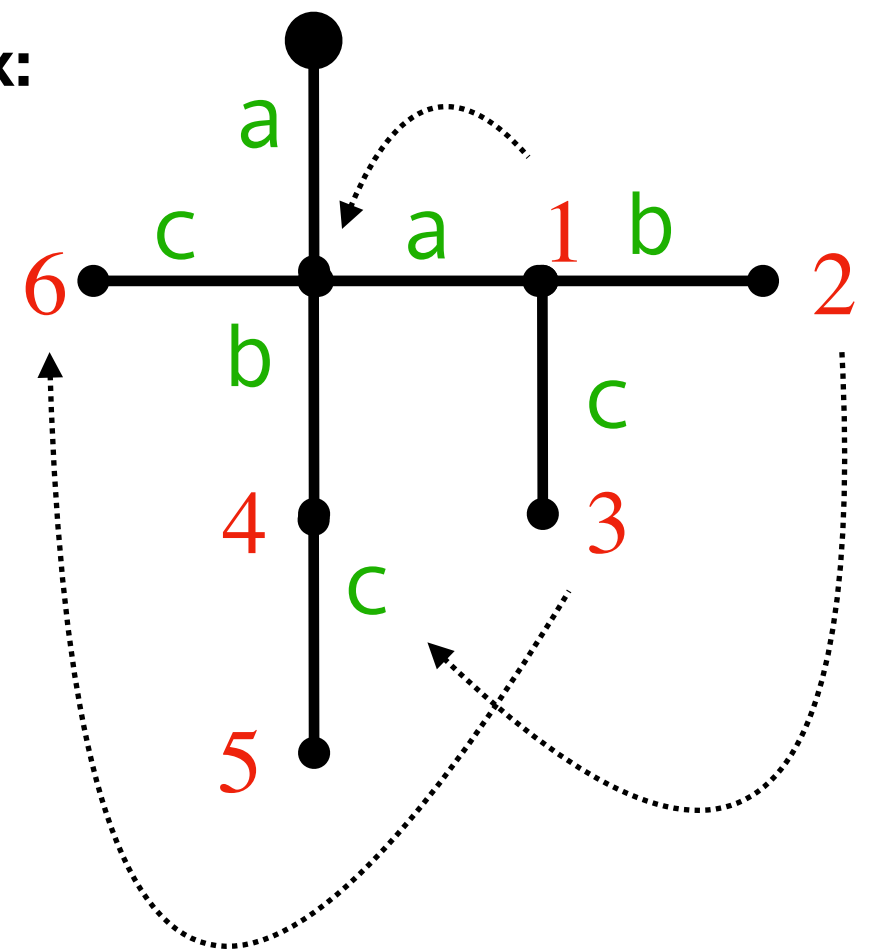




# Suffix links

- We want suffix links to be pointers in the trie as well

Patterns:	Prefixes of patterns:	Longest proper suffix:
1: aa	a	⊥
2: aab	aa	a
3: aac	aab	ab
4: ab	aac	ac
5: abc	ab	⊥
6: ac	abc	⊥
	ac	⊥



(Not showing pointers to the root,  
i.e. the empty suffix links)

# Using suffix links...

- Update the algorithm such that, when we cannot search any longer, we follow the suffix link unless we are at the root in which case we increase  $j$

**KMP after inner loop:**

```
if (i == 0) { j++; }  
else      { i = ba[i - 1]; }
```

**AH after inner loop:**

```
if (is_root(v)) { j++; }  
else          { v = s(v); }
```

# Search...

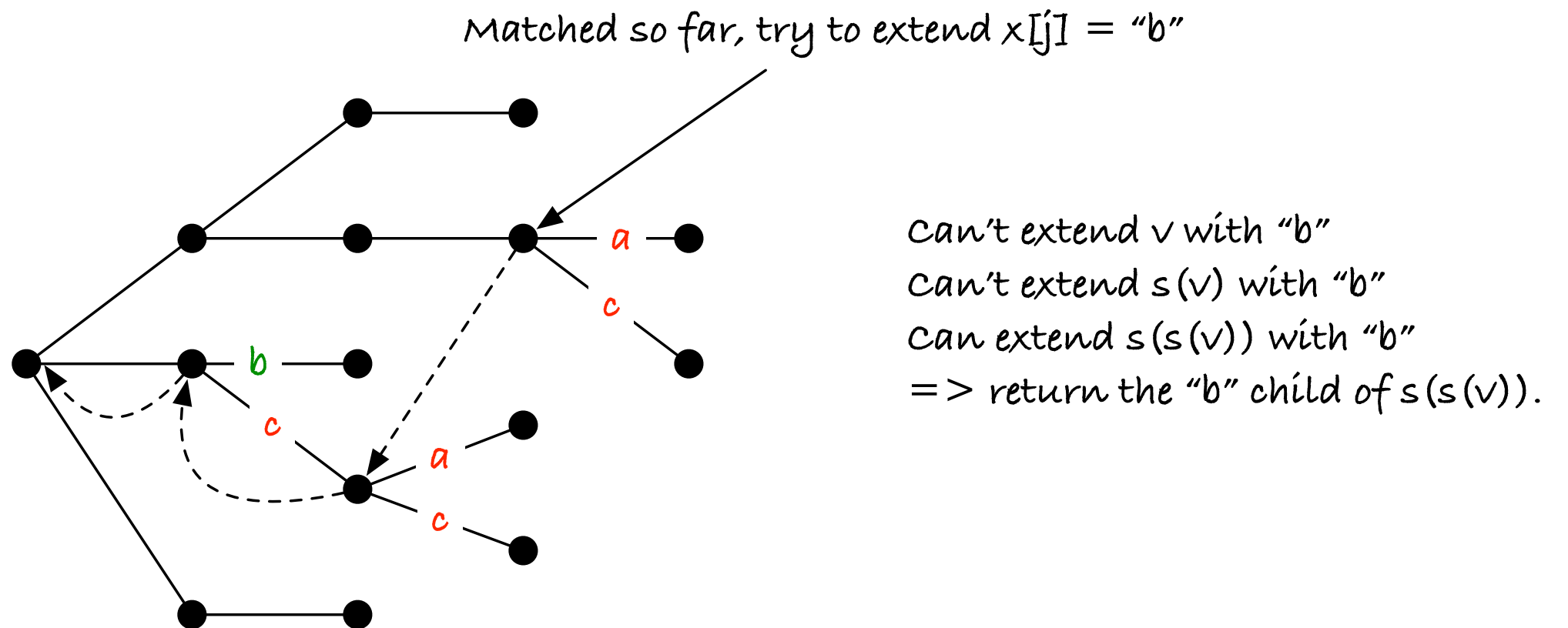
Change `j++` to `find_out(n, a)`

Where `find_out()` finds the next node if we extend the current with the letter `a`...

```
trie = build_trie(p)
n = trie.root
for j, a in enumerate(x):
    n = find_out(n, a)
    if n.label:
        report(n.label, j)
```

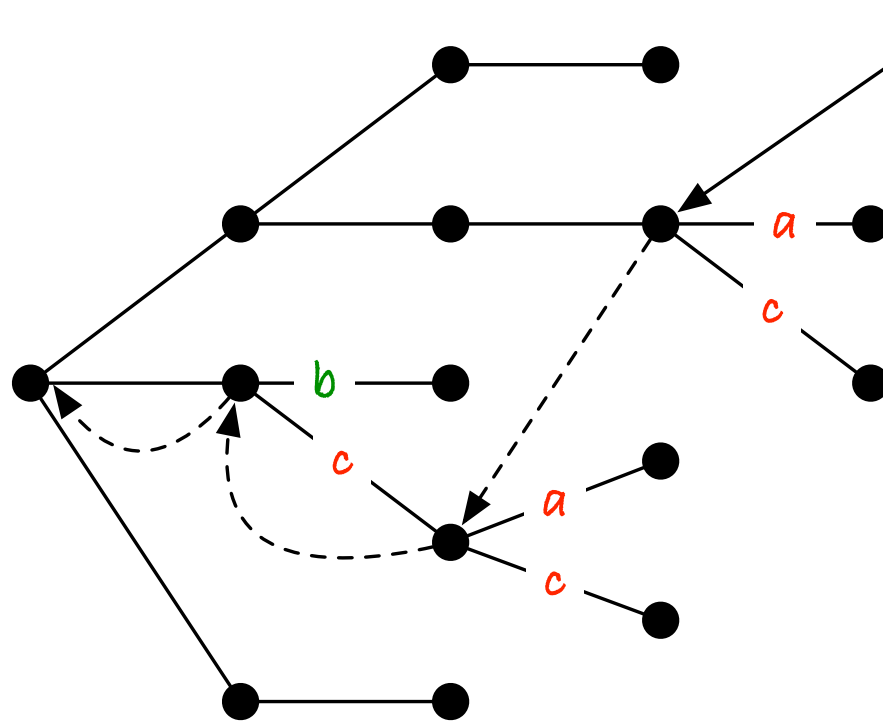
# Get next node

```
def find_out(v: TrieNode, a: str) → TrieNode:  
    while a not in v.children and not v.is_root:  
        v = v.suffix_link  
    return v.children[a] if a in v else v
```



# Get next node

Matched so far, try to extend  $x[j] = "b"$



Can't extend  $v$  with "b"  
Can't extend  $s(v)$  with "b"  
Can extend  $s(s(v))$  with "b"  
 $\Rightarrow$  return the "b" child of  $s(s(v))$ .

Conceptually, this is the border array algorithm:



# Is all well, then?

- Almost, but if one pattern is a substring of another, we will not discover it when we follow the suffix link...

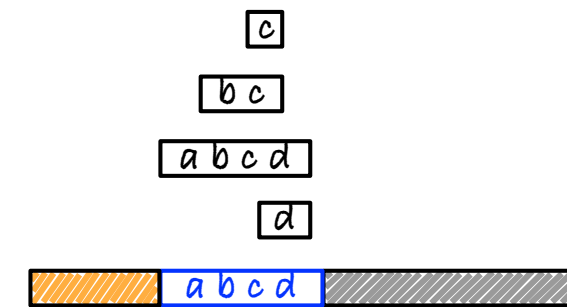
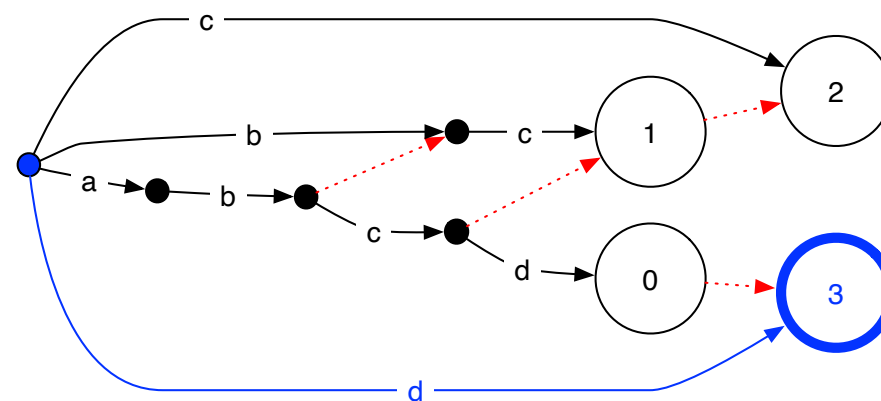
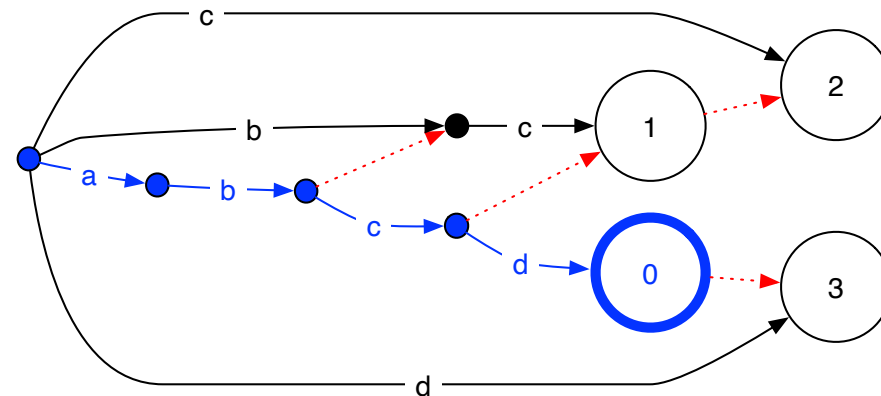
Patterns:

0: abcd

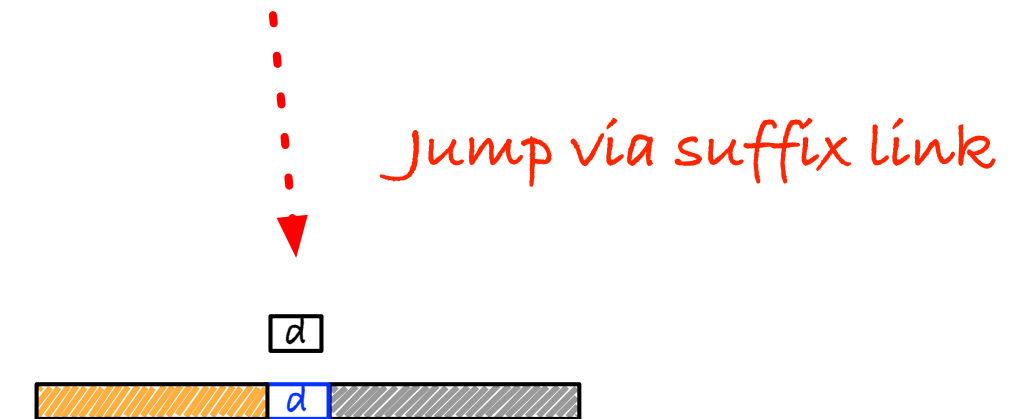
1: bc

2: c

3: d



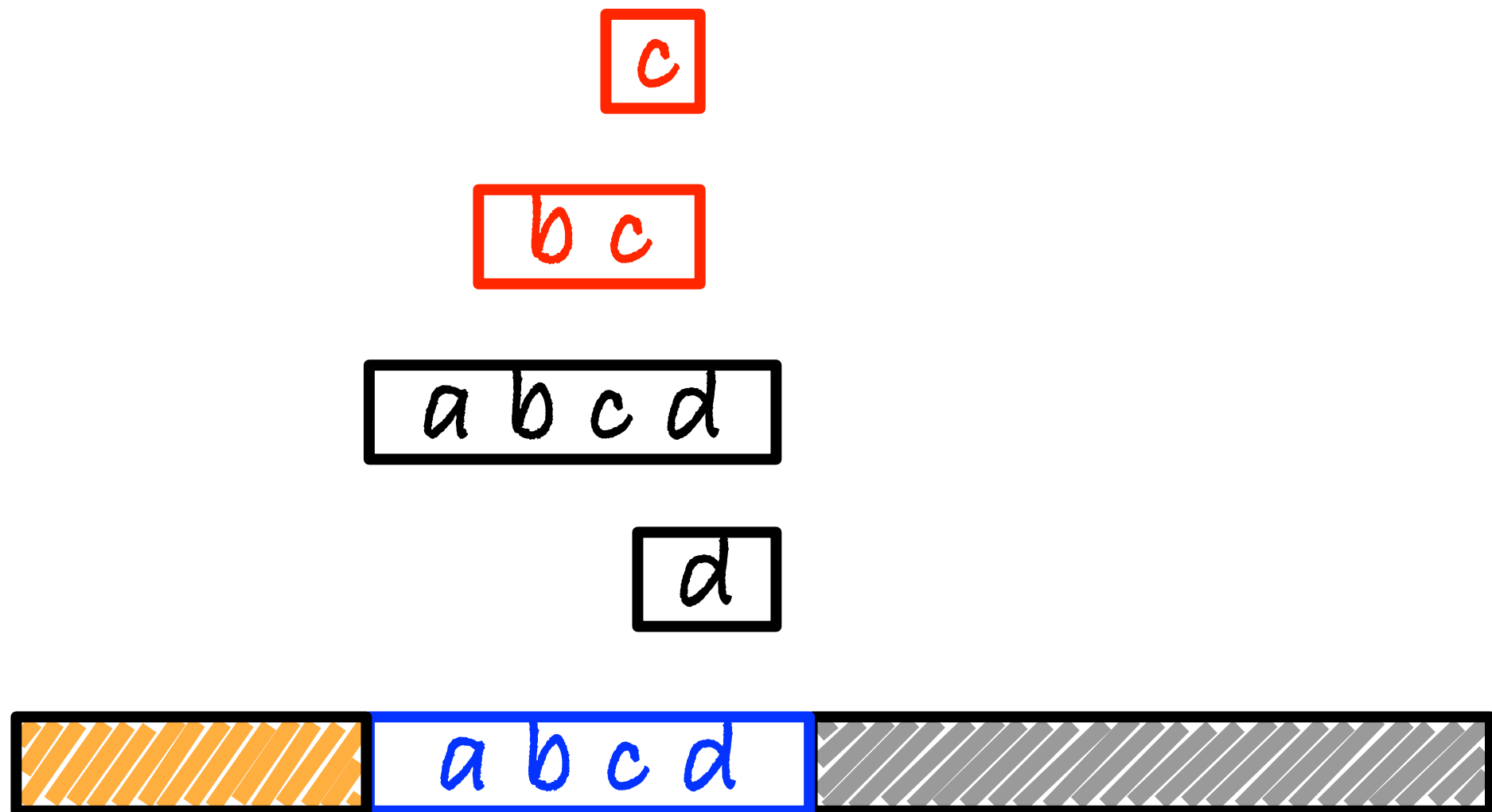
Report match of 0



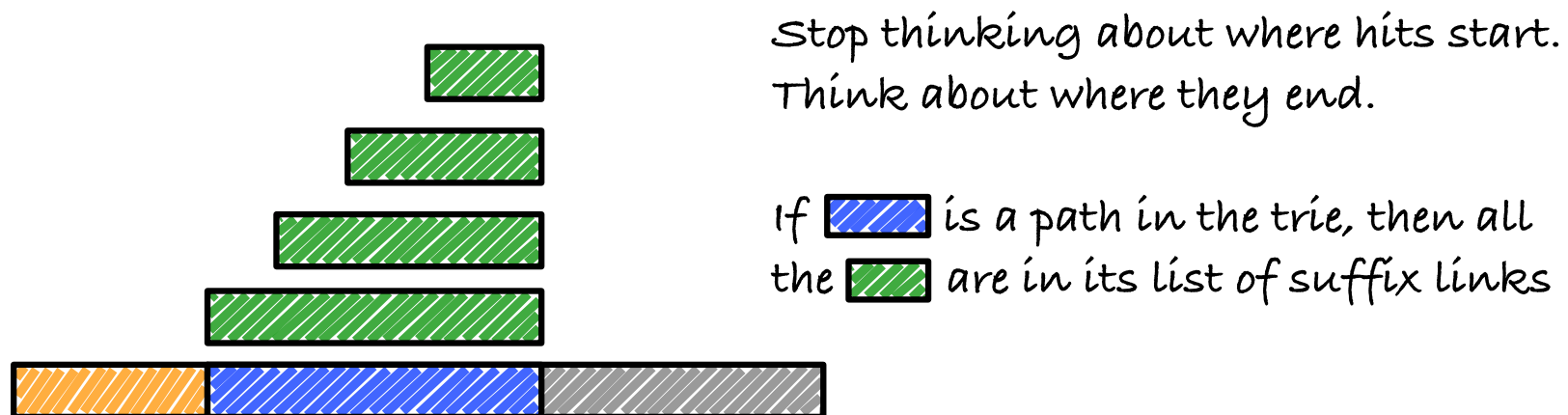
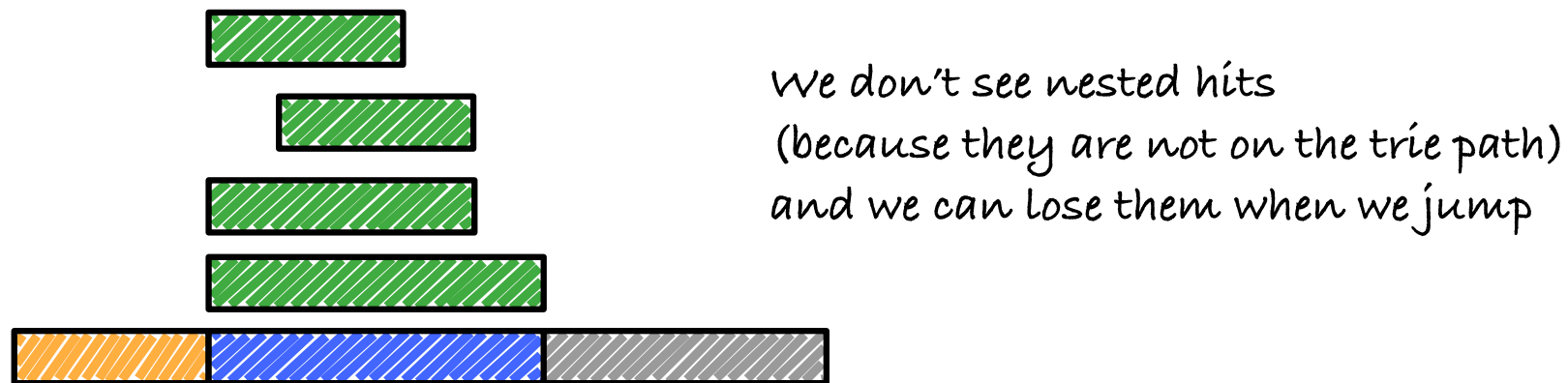
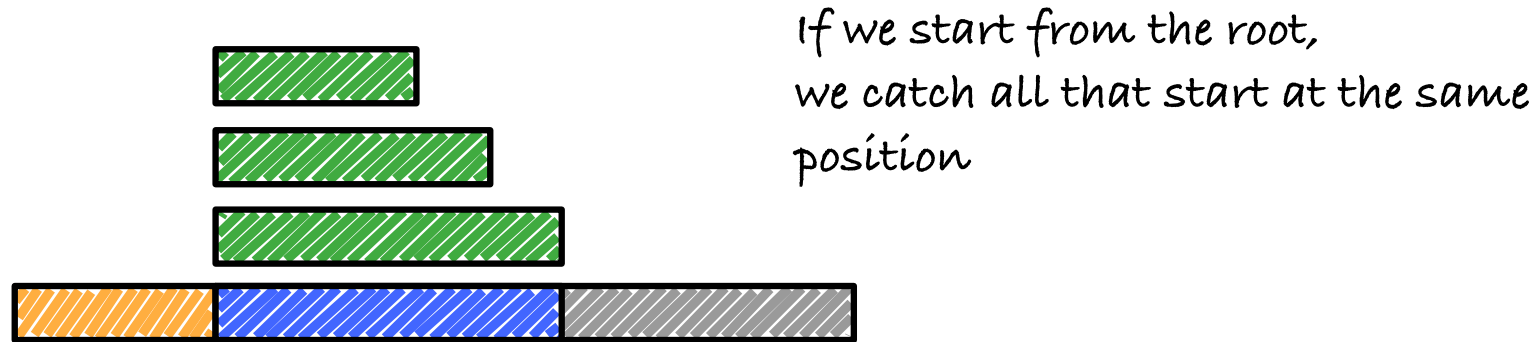
Report match of 3

We skipped the matches of 1 and 2 when we jumped!

# We miss nested strings 🤯

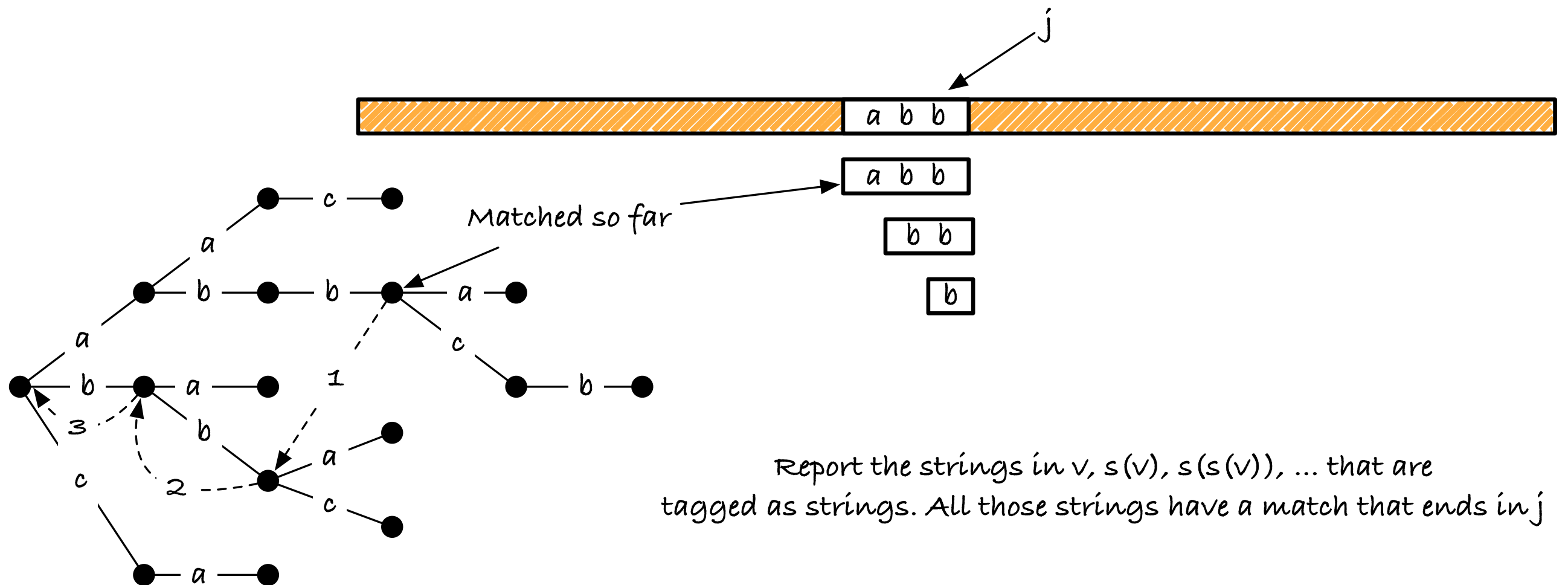


# Rethink





# Report hits from the suffix-links



# Updated algorithm

- For each  $j$ ,  $j = 0, \dots, n-1$ , update the current trie node  $v$ .
- For all nodes in the chain of suffix links, if the node is tagged, report a match that ends in  $j$

# Report occurrences

```
def occurrences(v: TrieNode) → typing.Iterator[int]
    while not v.is_root:
        if v.label is not None:
            yield v.label
        v = v.suffix_link

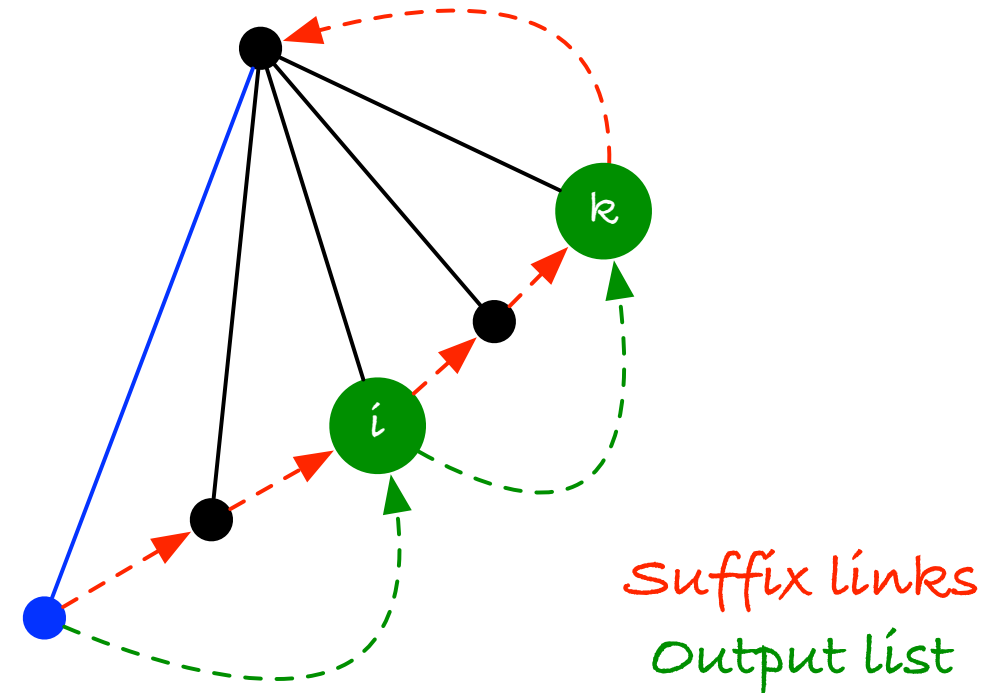
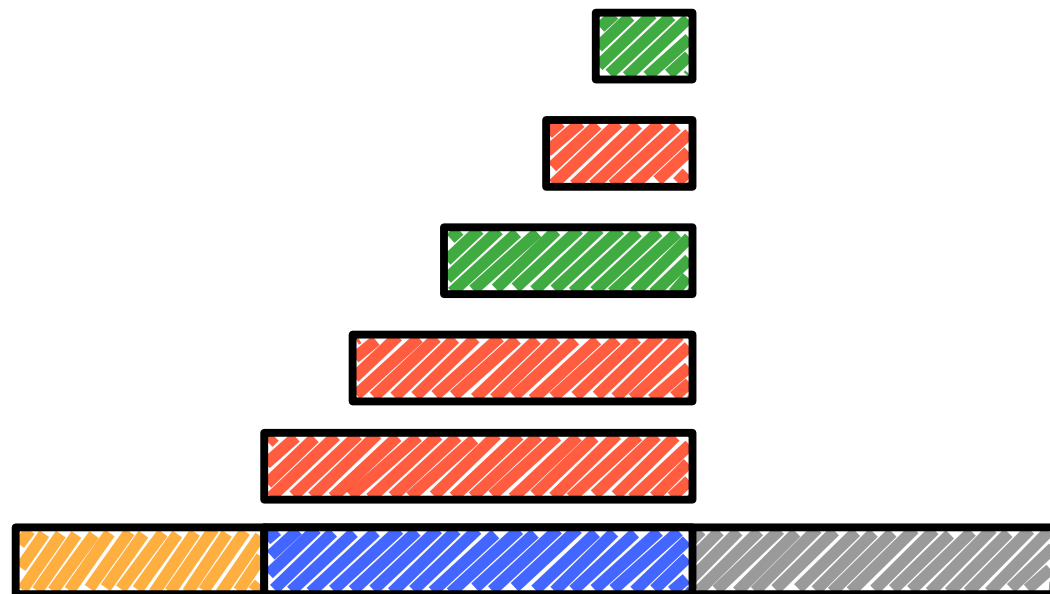
n = trie.root
for j, a in enumerate(x):
    n = find_out(n, a)
    for label in occurrences(n):
        yield (label, j)
```

With this impl. the root cannot hold a string  
(we never emit from it), but you can allow that with  
a different loop condition (just a little uglier)

# Problem

- Running up the suffix link can be time-consuming and give us a quadratic time algorithm.
- Give an example of input where we get  $O(nm)$  running time

# Output lists



```
def occurrences(v: TrieNode) → typing.Iterator[int]
    for w in v.output_list:
        yield w.label
```

# Running time

- For each index  $j$  in  $x$ , we
  - Find the node to extend (if there is one) [  $O(\text{next})$  ]
  - We extend [  $O(1)$  for each  $j$ , so  $O(n)$  ]
  - Output the hits that end in  $j$  [  $O(z)$  total,  $z$  # reported hits]

$$O(\text{next} + n + z)$$

# Running time

$$O(\text{next} + n + z)$$

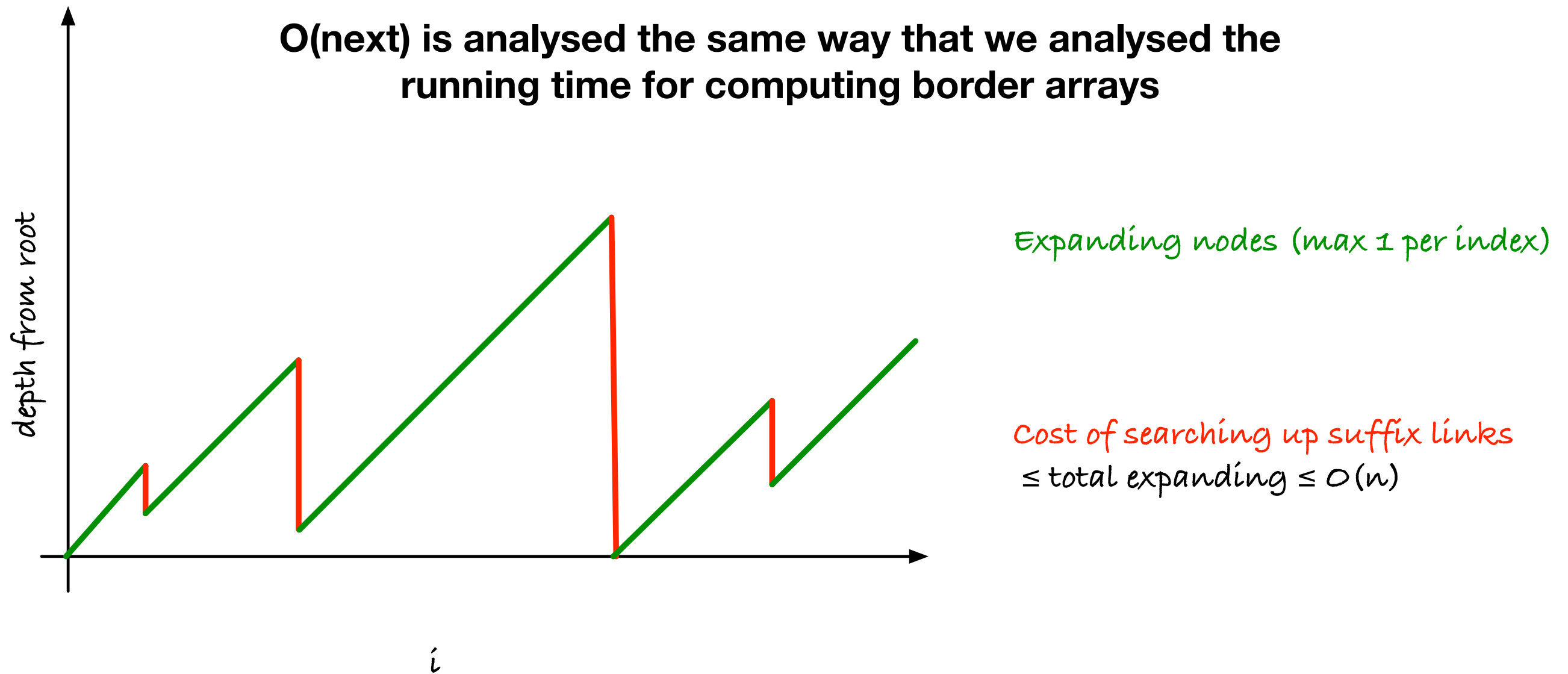
- **Observations:**

- Splitting the running time of an outer loop into the total contribution of various parts is often a good idea.
  - It is not the first thing we learn (multiplying outer and inner time usage), but sometimes the simple solution is too pessimistic.
  - We will do that quite often in this class.
- Using the output as a separate contribution is sometimes useful.
  - We can never do better than  $O(z)$ , but worst case  $z$  is often rare. Here it is  $O(nk)$  where  $k$  is # of patterns.
  - When we don't have worst case  $z$ , we want a fast algorithm, and it is then important to keep the other components small

# Running time

$$O(\text{next} + n + z)$$

$O(\text{next})$  is analysed the same way that we analysed the running time for computing border arrays



$$O(n + n + z) = O(n + z)$$



# Running time

$$O(n + \textcolor{blue}{z})$$

$O(n + \textcolor{blue}{kn})$  if you don't like  $z$

**This time does not include preprocessing, though...**

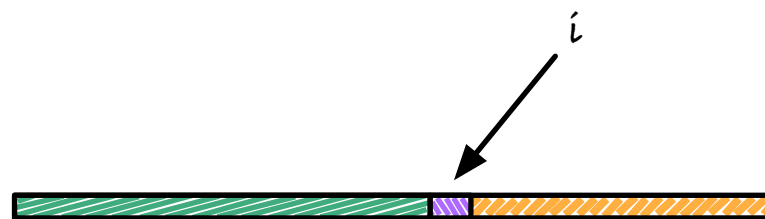
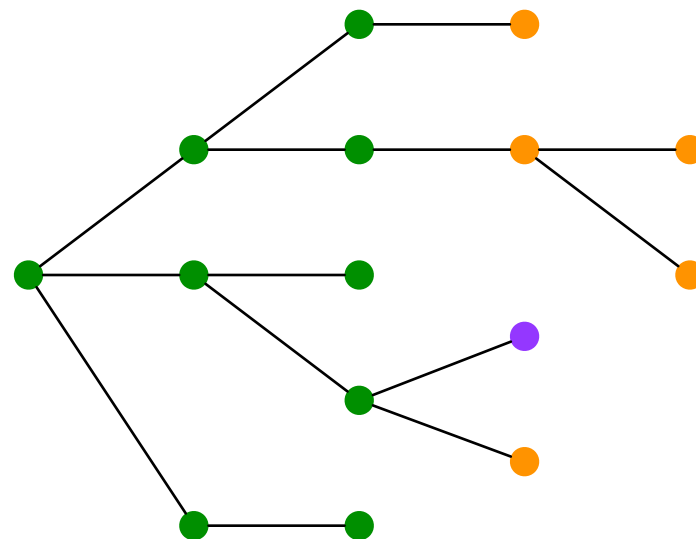
# Computing the suffix links and output lists

- We can build the trie in  $O(m)$  where  $m$  is the sum of lengths of the patterns.
- We will compute the lists in time  $O(m)$  as well.
- The result will be a running time of  $O(m + n + z)$  for the algorithm in total.
- We compute the lists in a breadth first order so the lists will already be computed for all nodes closer to the root than the one we are already processing.

# Breadth-first traversal

Breadth first traversal

All nodes closer to the root are processed



Corresponds to left-to-right processing

All positions to the left of the current are processed

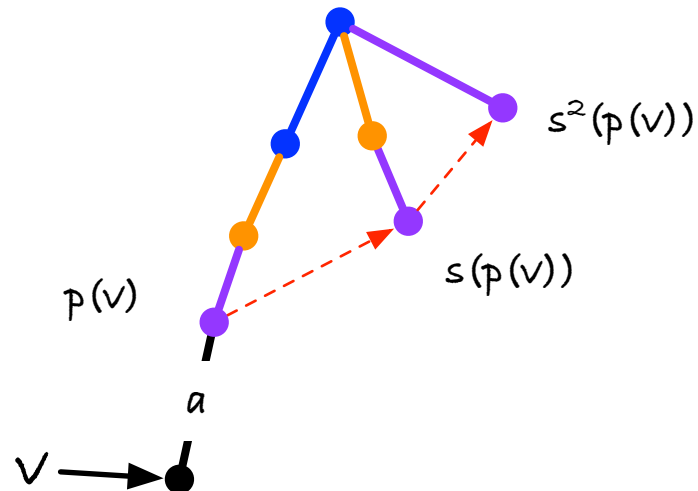
**How do you traverse a tree breadth-first?**

# Computing suffix links

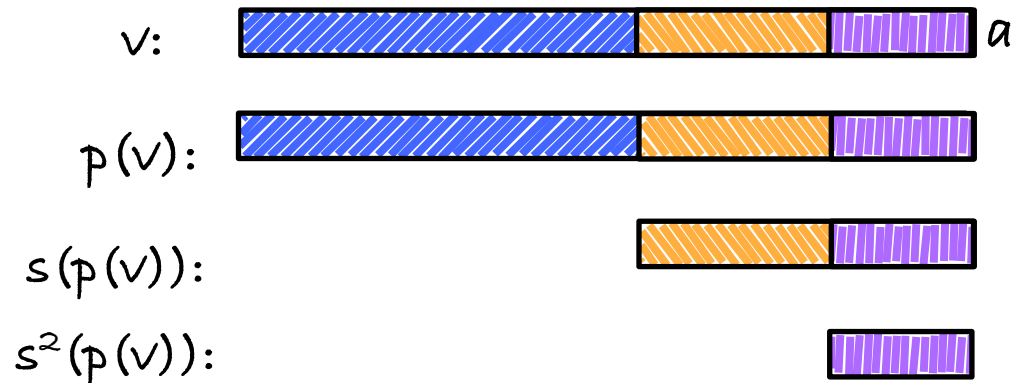
Suffix link of root is root (similar to  $b[0] = 0$  for border arrays).

Non-root node  $v$  correspond to string  $p(v) + "a"$

Trie:



String:



BA analogue:

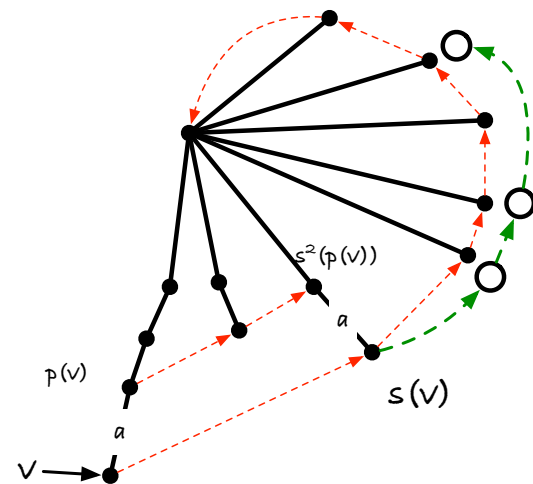
$i$   
 $i-1$   
 $b[i-1]$   
 $b^2[i-1]$

---> links exist because we traverse breadth-first

Computed like border array. Instead of looking at  $b[i-1]$ , look at the parent of node  $v$ ,  $p(v)$  and its suffix link,  $s(p(v))$ . Try to extend it. If it fails, look at  $s(s(p(v)))$  and try to expand it. If that fails, keep going, until you reach the root (which you may or may not be able to expand).

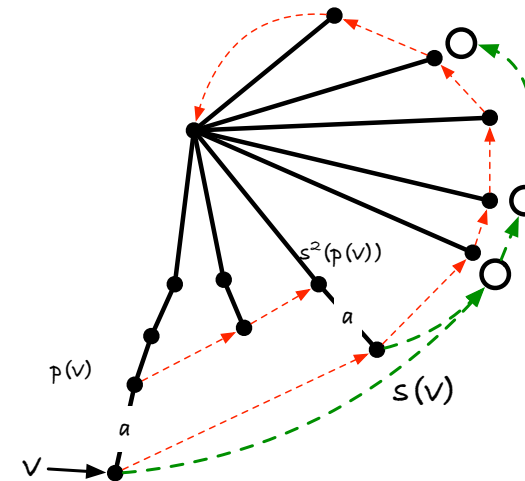
# Computing output lists

Output links are sub-lists of suffix links.  
It's either  $s(v)$ 's output list or you prepend  $v$  if  $v$  is tagged  
as a string.

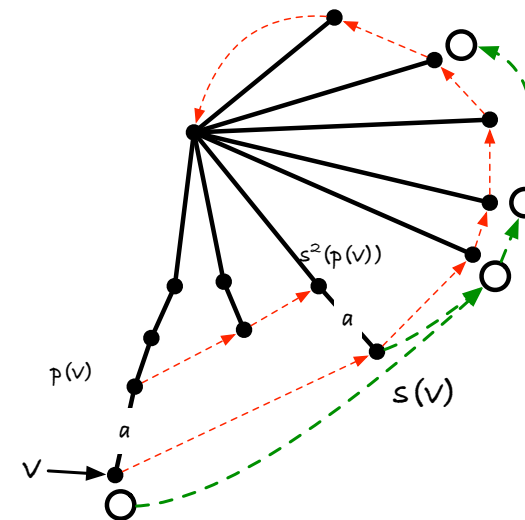


---> links exist because we traverse breadth-first  
---> and we just added  $v \rightarrow s(v)$

```
if v.label is None:  
    v.outlist = v.suffix.outlist
```



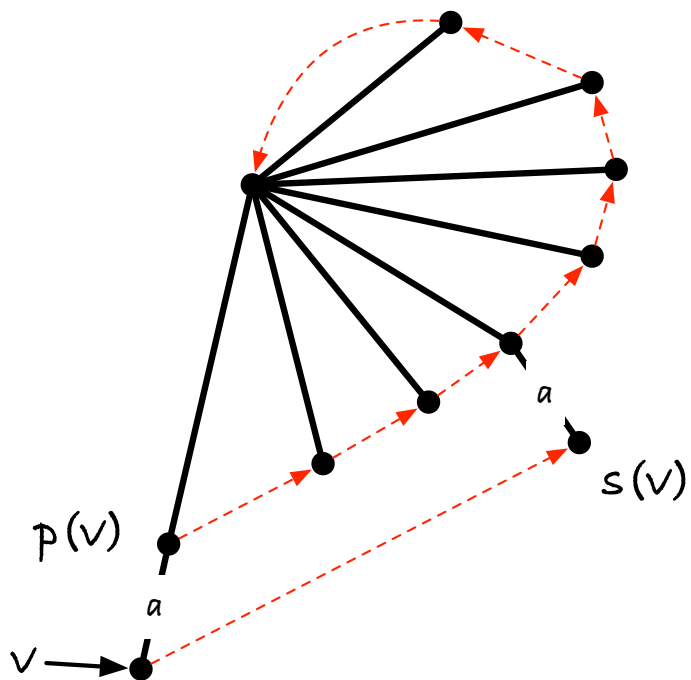
```
if v.label is not None:  
    v.outlist = Link(v, v.suffix.outlist)
```



# Running time

- Once we have the failure links, we can obviously set the output lists in constant time per node (update a linked list)
- Traversing the trie breadth-first takes time  $O(m)$  (size of the trie, which is in  $O(m)$  where  $m$  is the sum of pattern lengths)
- The tricky part is the analysis of computing the suffix links

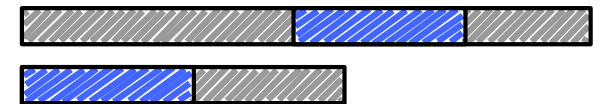
# Running time



$d(w)$  = depth of node  $w$   
(distance to root)

$d(s(w)) = d(w)$  if  $w$  is root, otherwise  
 $d(s(w)) \leq d(w) - 1$

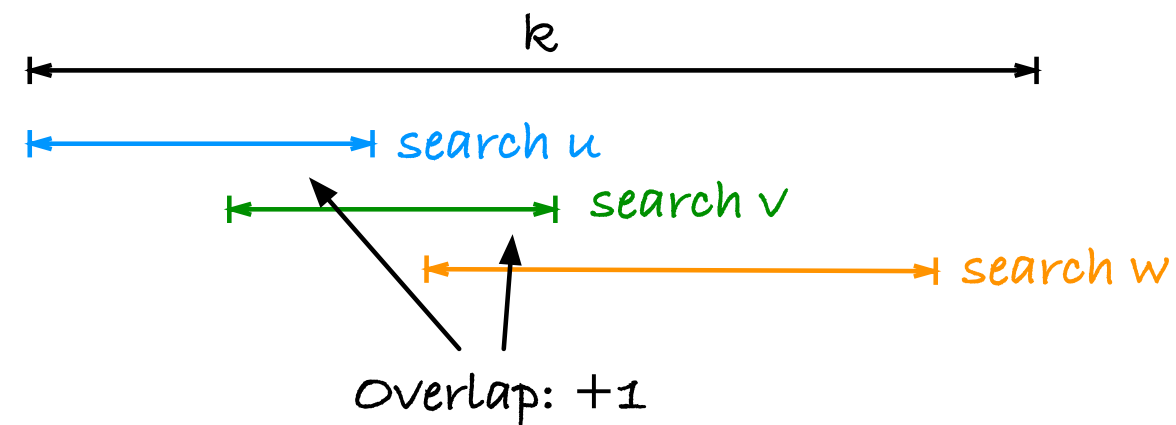
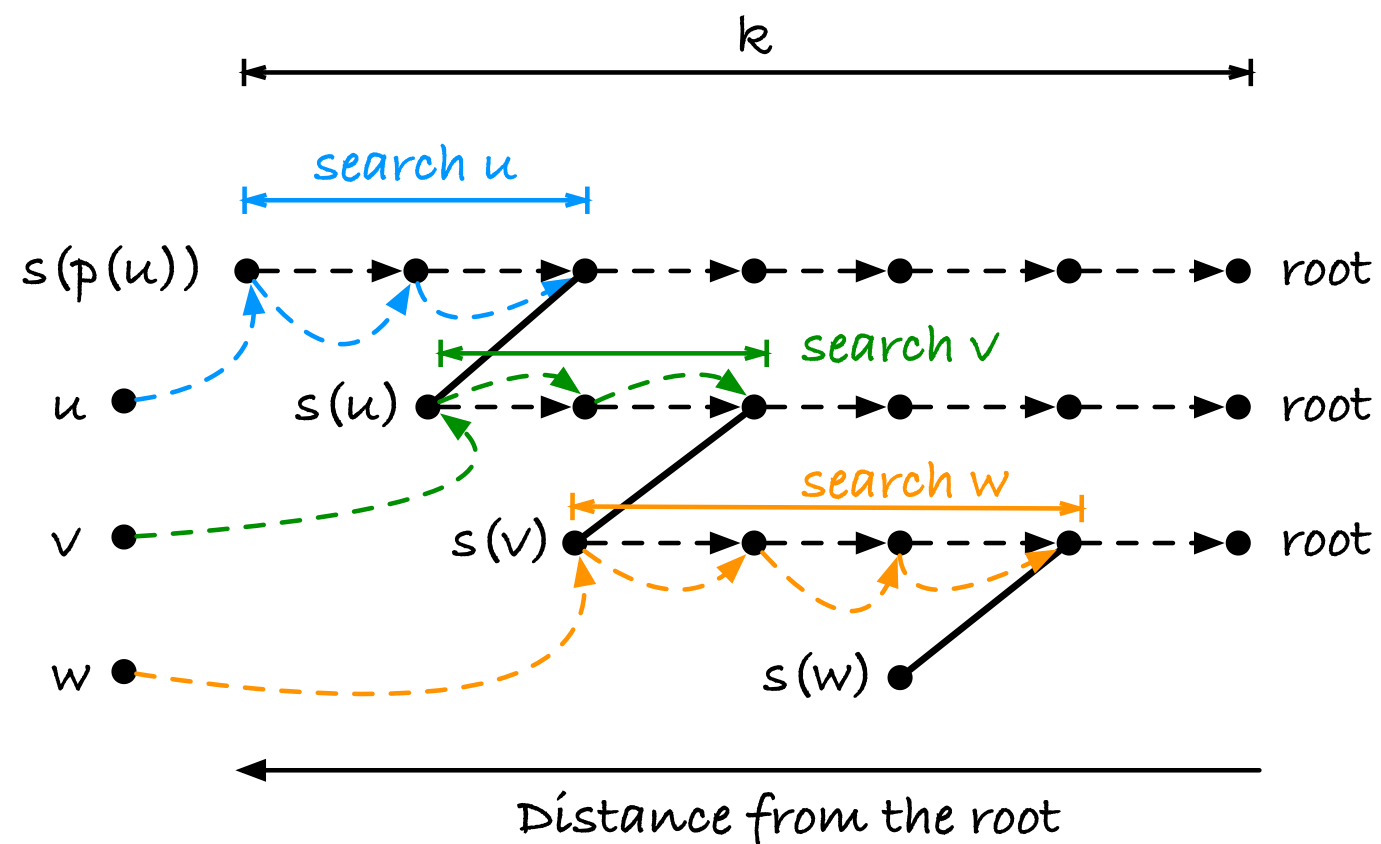
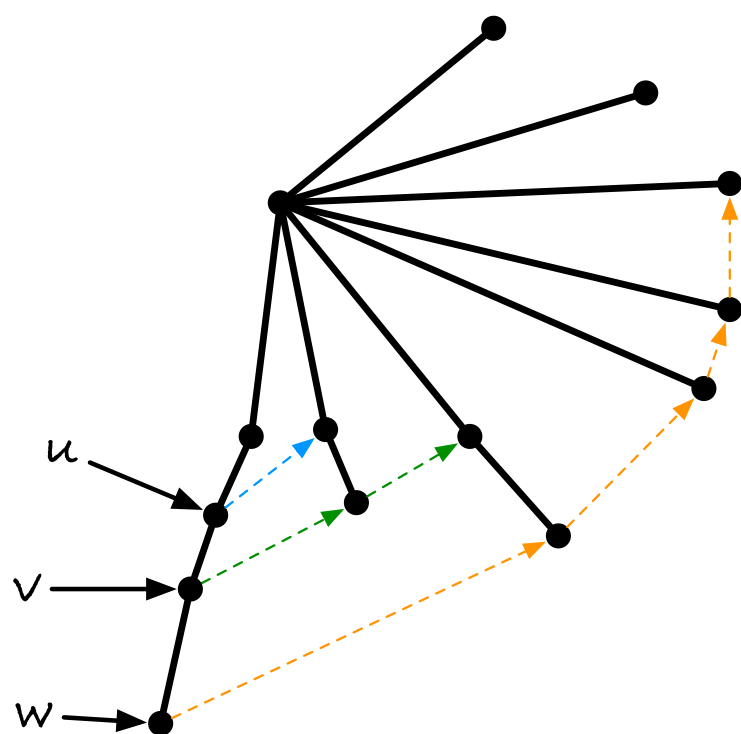
suffix links are proper, i.e.,  
not equal to the full string



Moving along  $s$  links gets us closer to  
the root, so search time bounded by  $d(s(p))$

What is the total search time along a path?

What's the cost of searching for  $u$ ,  $v$ , and  $w$ ?



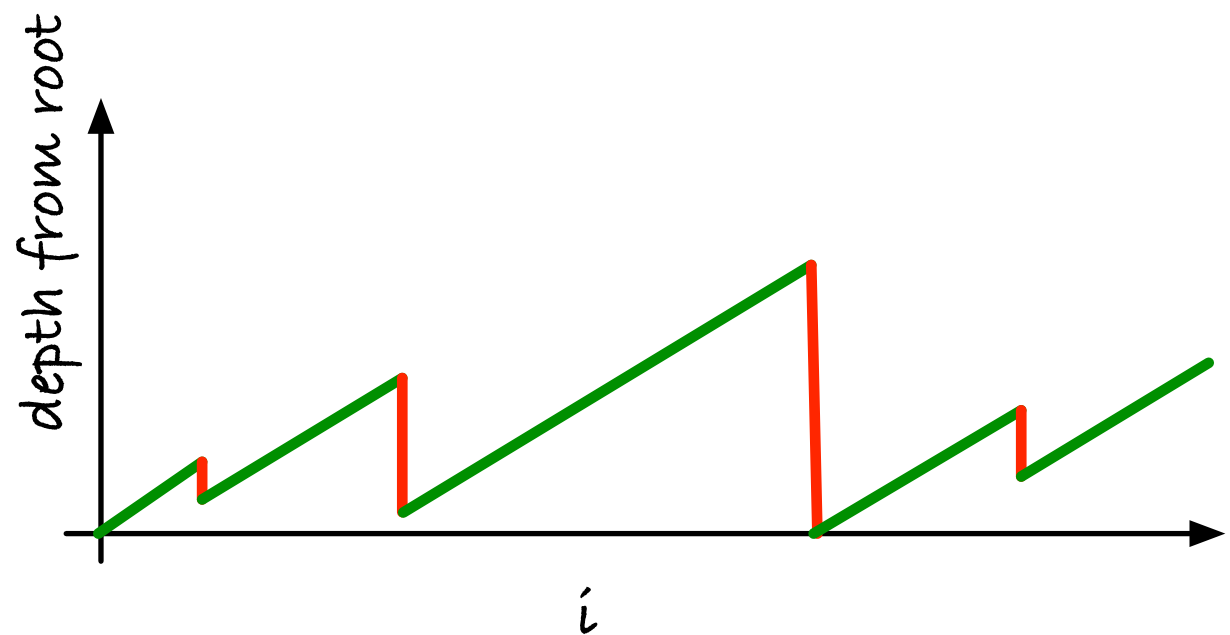
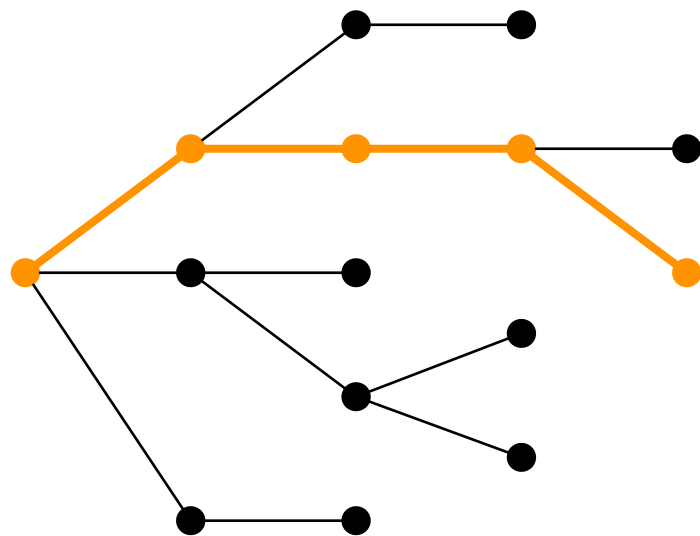
$$\text{search } u + \text{search } v + \text{search } w \leq k + 2$$

(general  $\leq n + \text{depth}$ )



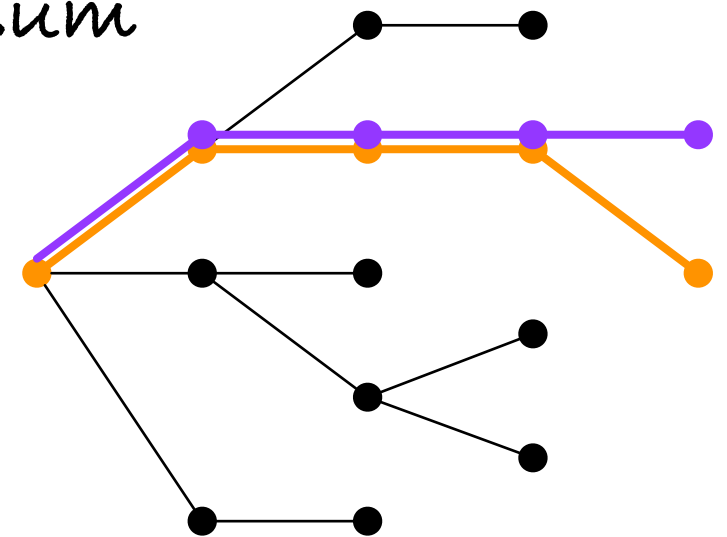
Imagine that we can build the links depth-first (we can't, but imagine it anyway, because we will count the amount of work we did in each node).

On any path from root to node  $w$ , we cannot search more than  $O(d(w))$  (amortised argument)



If we know that the cost of getting down to a leaf is bounded, we can bound the cost of getting to all leaves (i.e. the cost of the entire tree). It is the sum of all paths from the root to a leaf.

All paths from root to leaves are bounded by the strings we made the trie from, so the running time is  $O(m)$

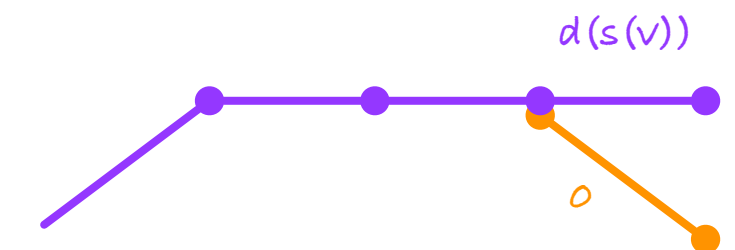
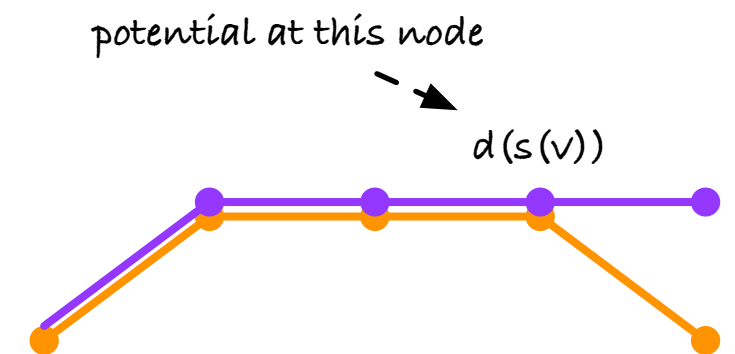
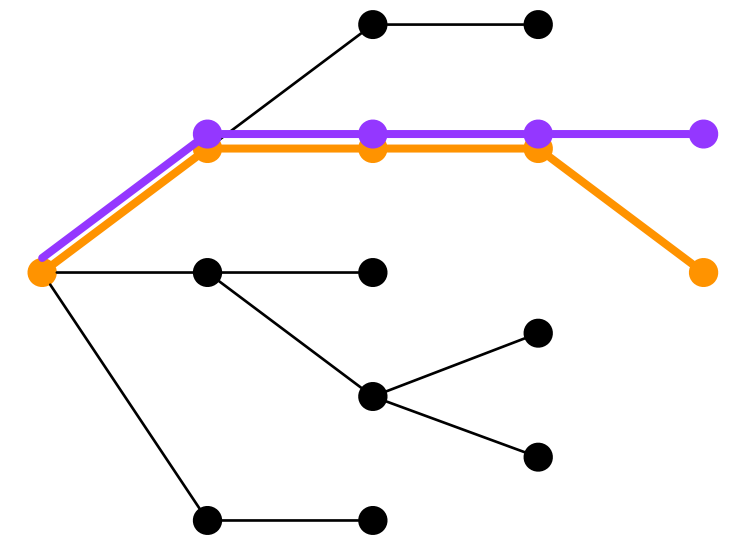


NB! We are counting some edges more than once, even though we never process an edge more than once. We have to, for the argument to work.

Sometimes we can merge paths, but not with amortisation arguments. If you feel tempted to, consider potentials.

You save up potential along a path. If you use it in one branch, it isn't available for another. If you merge paths, you only get to use the savings once, but often you need it for each path (like we do here).

Be careful when you use amortisation on trees and graphs. If you use a potential, you have to include all the paths you use it on. (lazy evaluation can sometimes alleviate this, but not here).





*That's all Folks!*