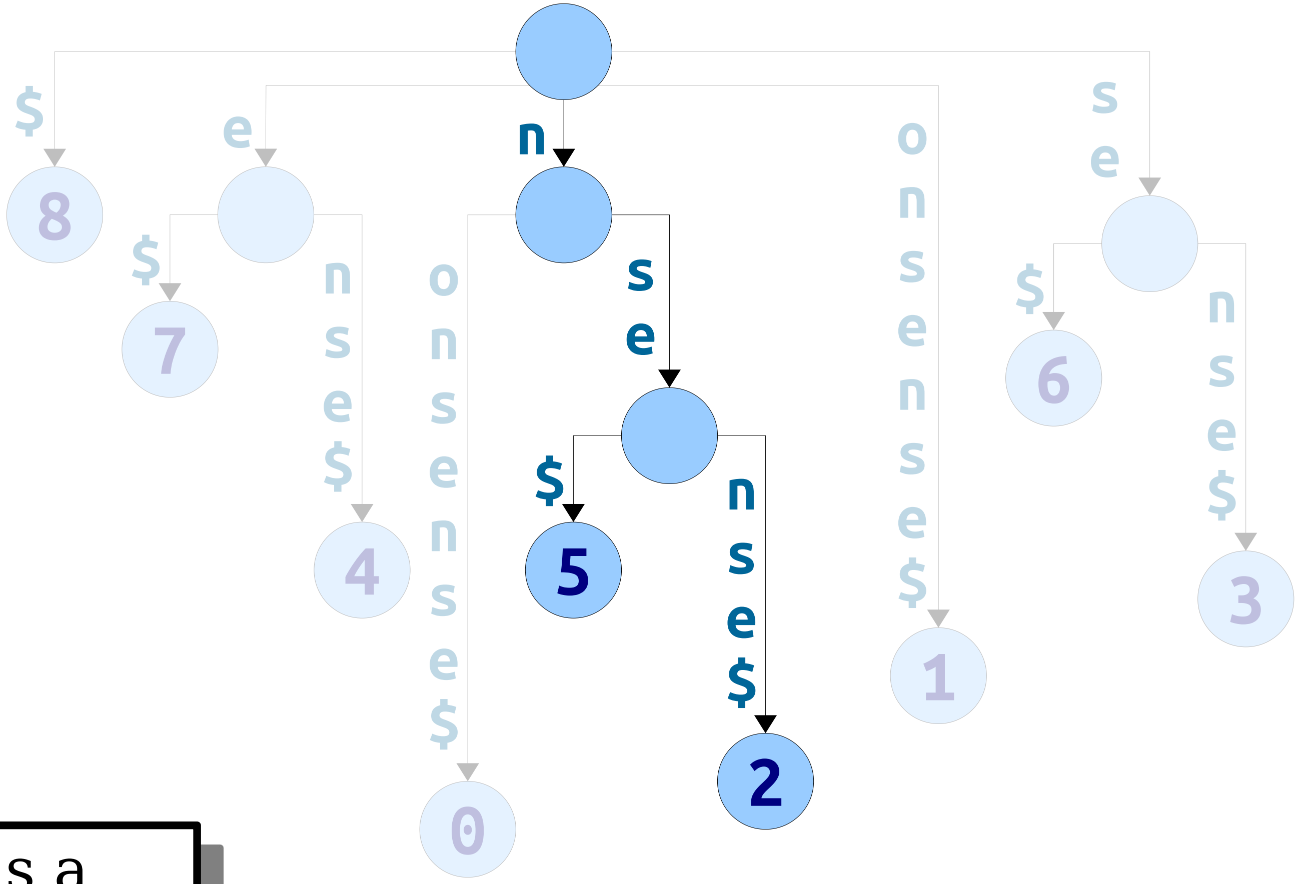
Suffix and LCP Arrays

Recap from Last Time

Suffix Trees

nonsense$
onsense$
nsense$
sense$
ense$
nse$
se$
e$
$



nonsense$
012345678

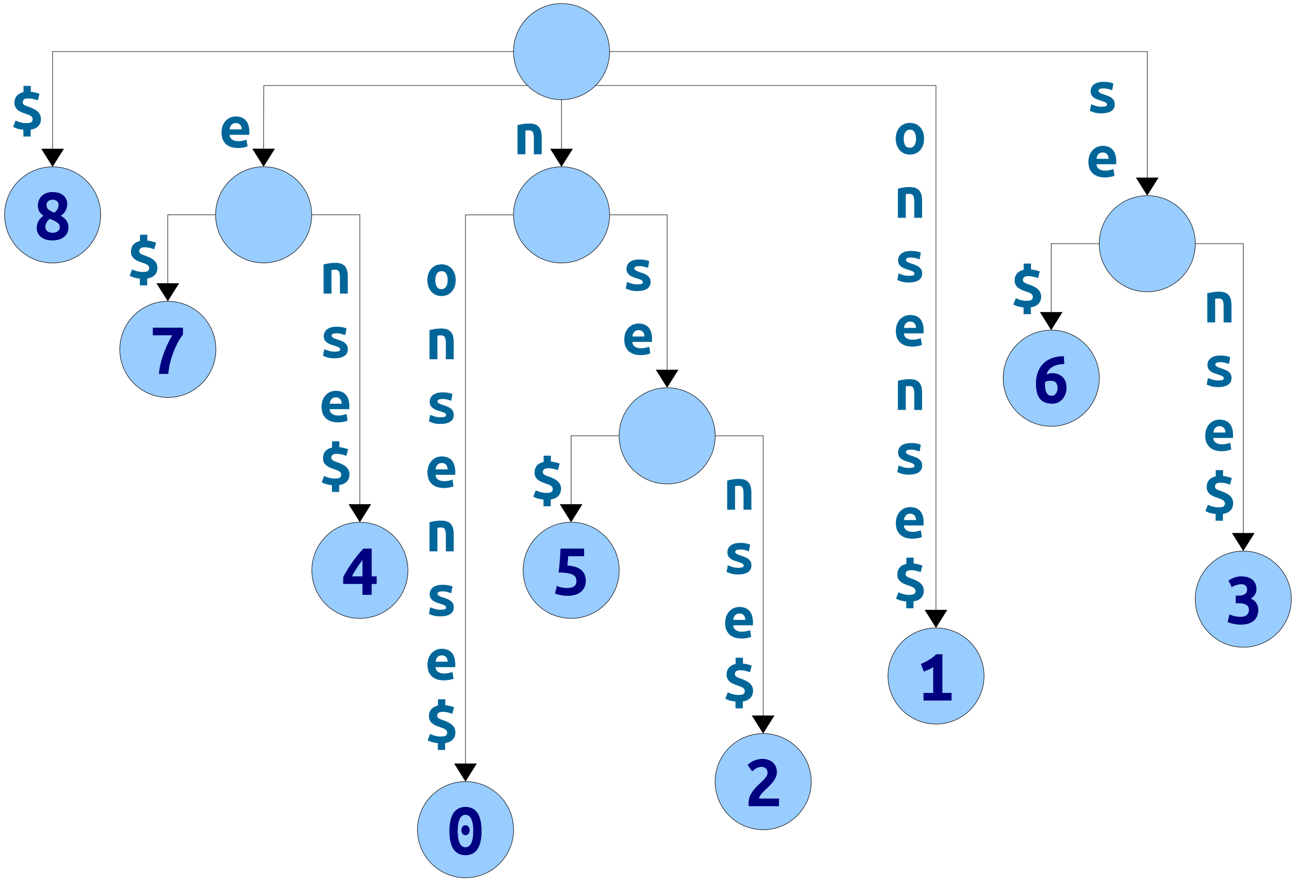
Theorem: *w* is a substring of *x* if and only if *w* is a prefix of a suffix of *x*.

New Stuff!

Representing Suffix Trees

Representing a Suffix Tree

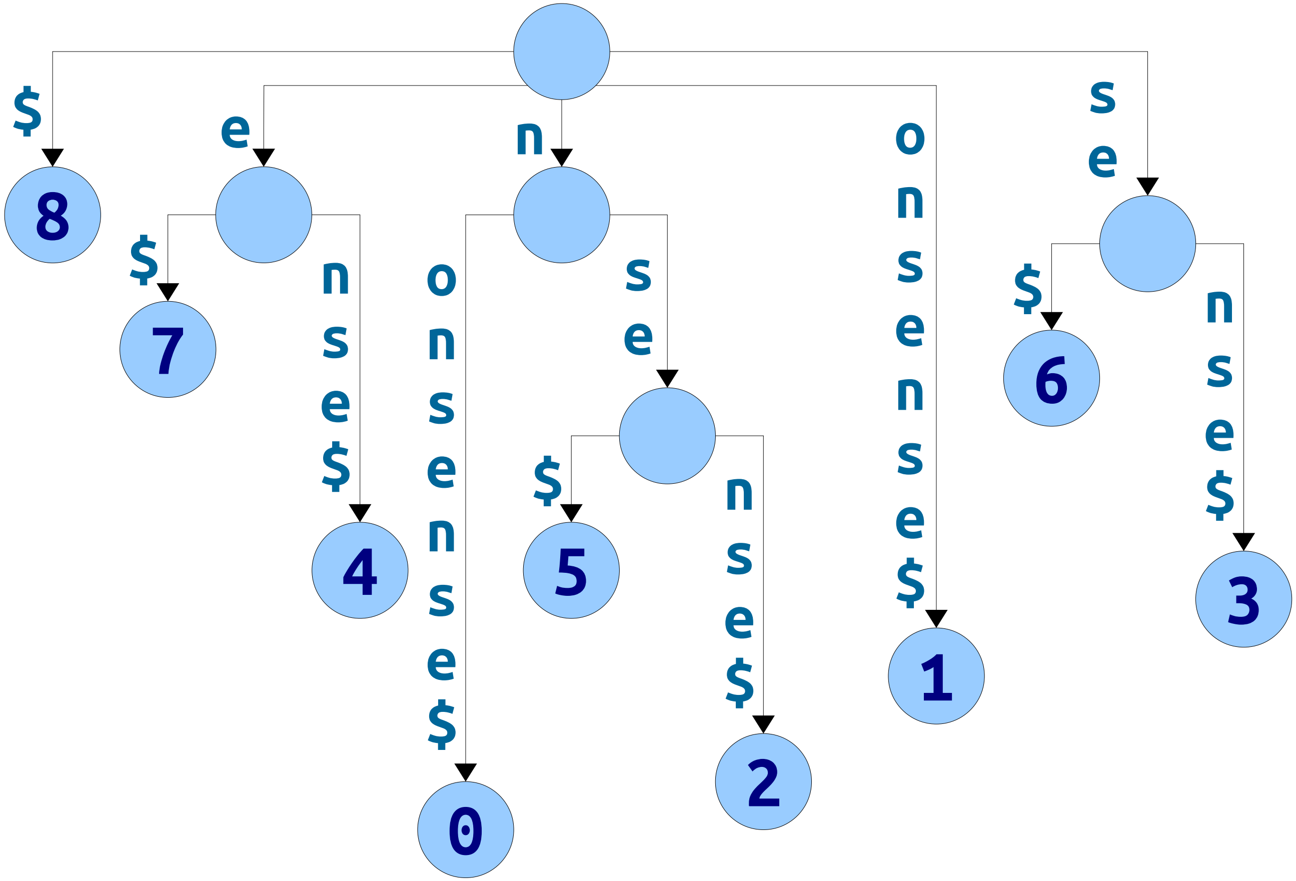
- We know that a suffix tree has O(***m***) nodes, where ***m*** is the number of characters in the input string.
- This means that there are O(***m***) edges.
- ***Question:*** Why can't we immediately claim that the space usage of the suffix tree is O(***m***)?



nonsense$
012345678

Representing a Suffix Tree

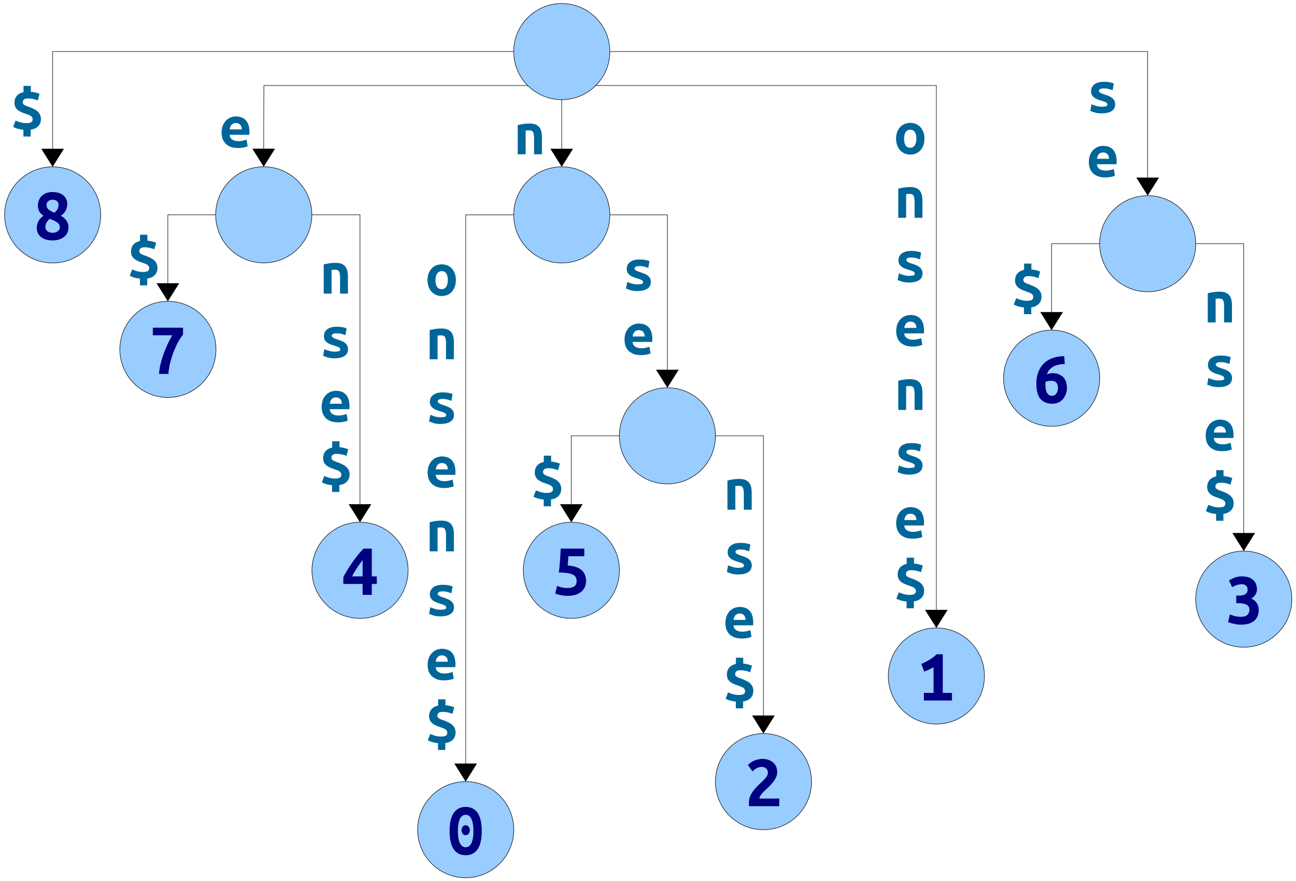
- ***Claim:*** Writing out all suffixes of a string of length m requires $\Theta(m^2)$ characters.
- ***Proof idea:*** Those suffixes have length $1 + 2 + \ldots + (m+1)$, factoring in the special **$** character.
- ***Problem:*** It is indeed possible to build a suffix tree with $\Theta(m^2)$ total letters on the edges.



nonsense$
012345678

Representing a Suffix Tree

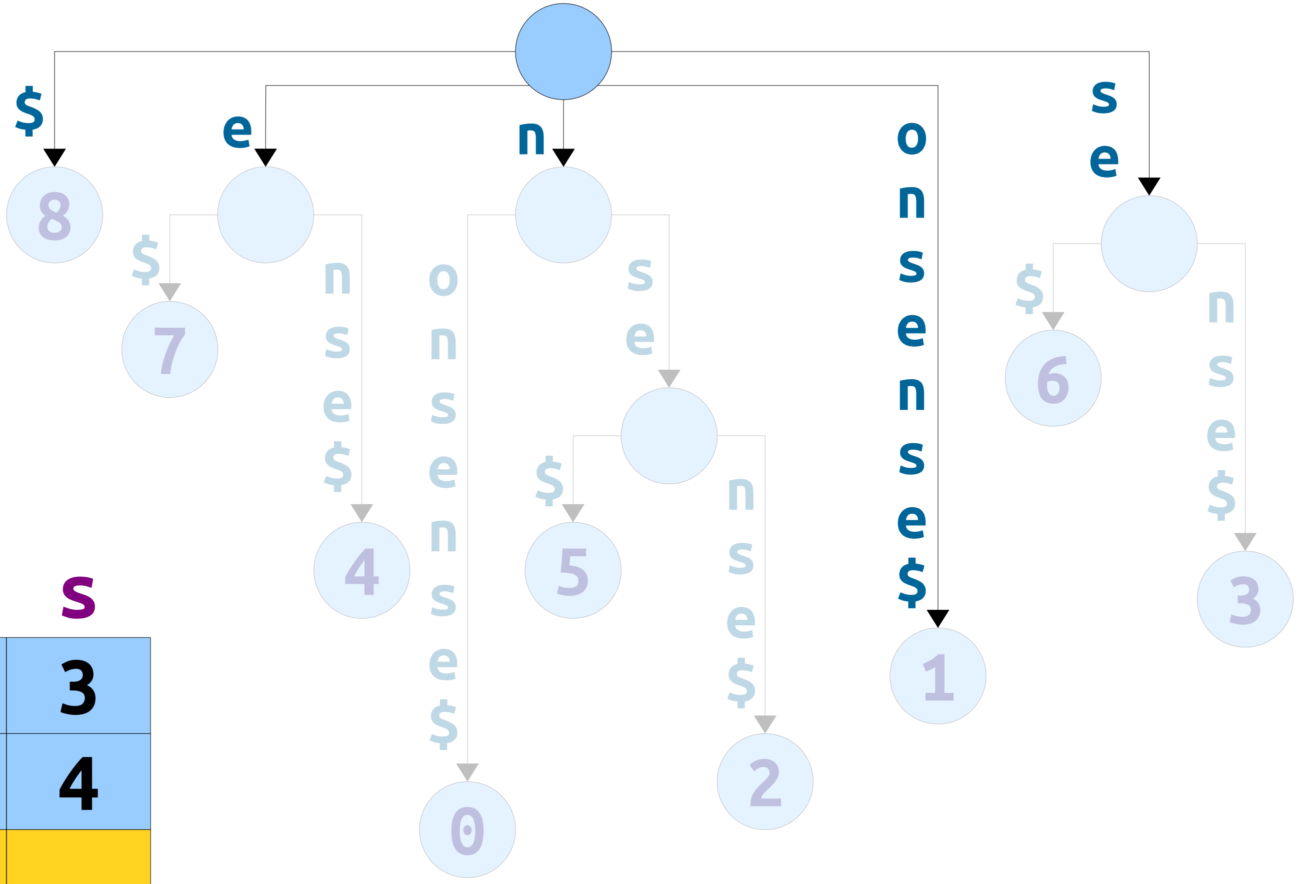
- By being clever with our representation, we can guarantee that a suffix tree uses only Θ(***m***) space, regardless of the input string.
- ***Observation:*** Each edge is labeled with a substring of the original input string.
- ***Idea:*** Don’t actually write out the labels on the edges. Just write down the start and end index!



nonsense$
012345678

Representing a Suffix Tree

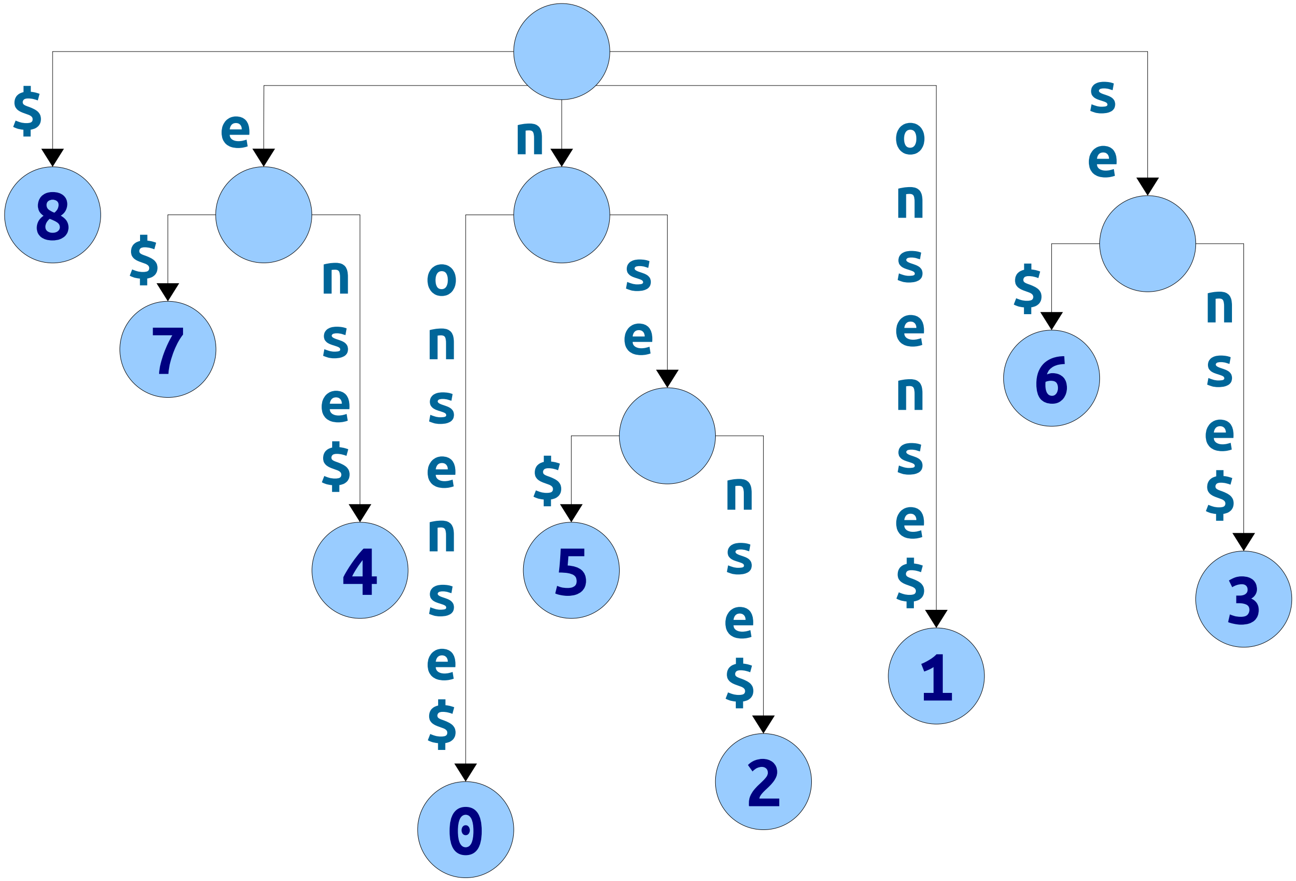
	$	e	n	o	s
start	8	4	0	1	3
end	8	4	0	8	4
child					



nonsense$
012345678

Representing a Suffix Tree

- Space usage required for a suffix tree:
  - O(***m***) space for all the nodes.
  - O(***m***) space for a copy of the original string.
  - O(***m***) space for the edges.
- Total space: O(***m***).



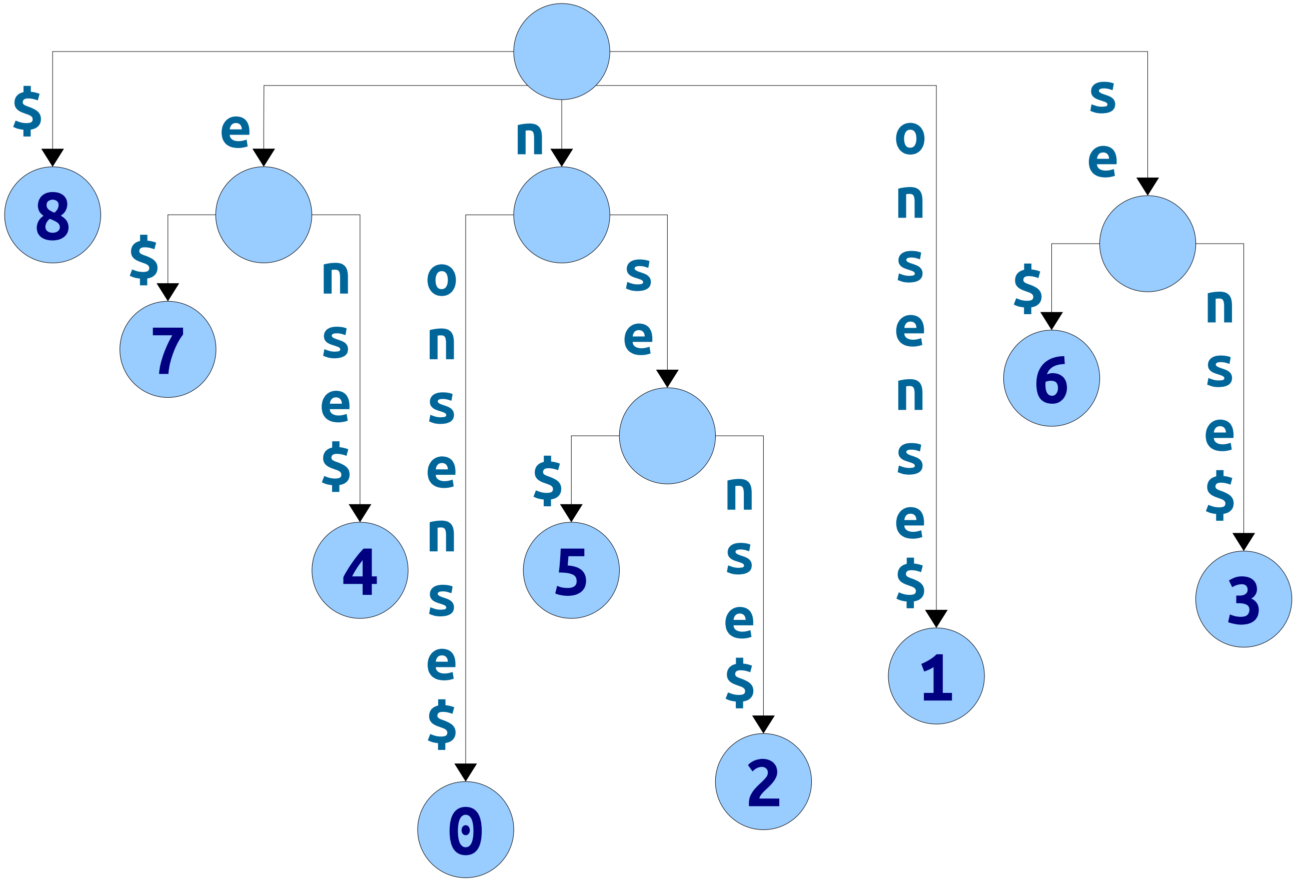
nonsense$
012345678

Suffix Tree Space Usage

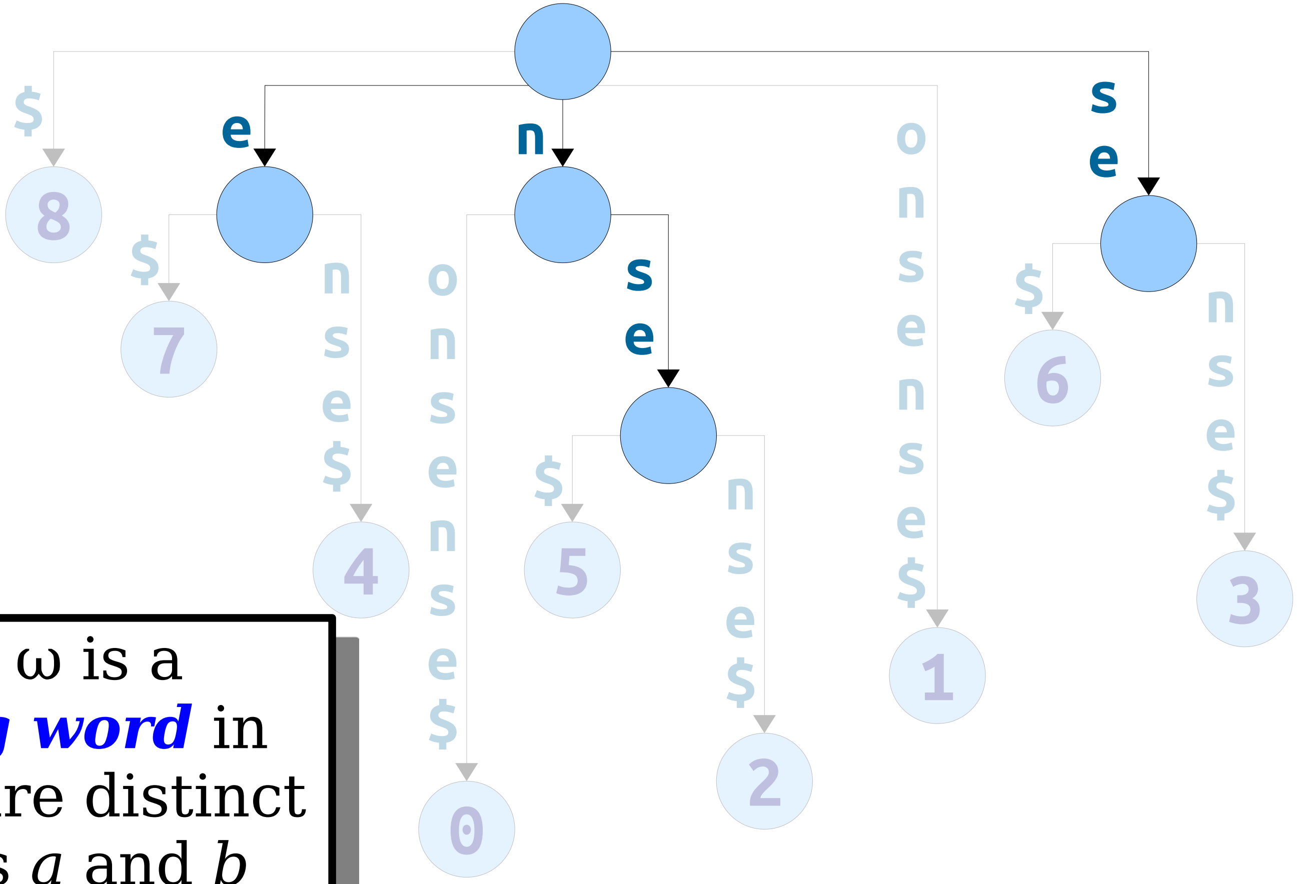
- Suffix tree edges take up a *lot* of space.
  - Two machine words per edge to denote the range of characters visited.
  - One machine word per edge for the pointer itself.
  - Number of edges ranges from m to $2m$ – 1, so this is between $3m$ and $6m$ machine words for the whole string!
- Example: a human genome is about three billion characters long.
  - With clever techniques, that can be packed into about 800MB.
  - On a 32-bit machine, the suffix tree needs about 48GB – too big to fit into memory!
  - On a 64-bit machine, the suffix tree needs about 96GB – way more than a typical machine can hold!

Key Question: Can we get the benefits of a suffix tree without the space penalty?

What is it about suffix trees that make them so useful algorithmically?

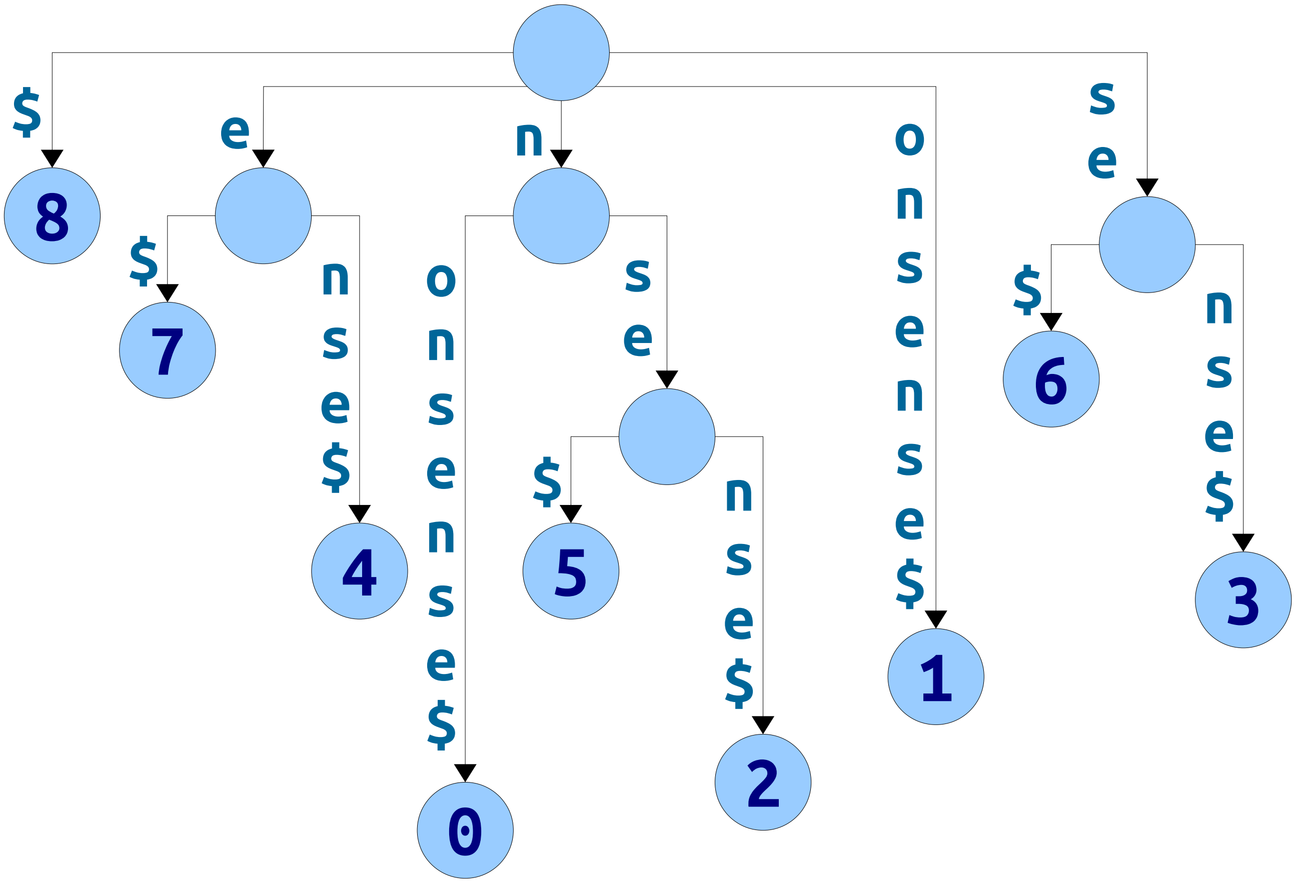


Theorem: There is a node labeled ω in a suffix tree for *T*
if and only if
ω is a suffix of *T*$ or ω is a branching word in *T*$.



A string ω is a ***branching word*** in *T*$ if there are distinct characters *a* and *b* where ω*a* and ω*b* are substrings of *T*$.

Theorem: There is a node labeled ω in a suffix tree for *T* *if and only if* ω is a suffix of *T*$ or ω is a branching word in *T*$.



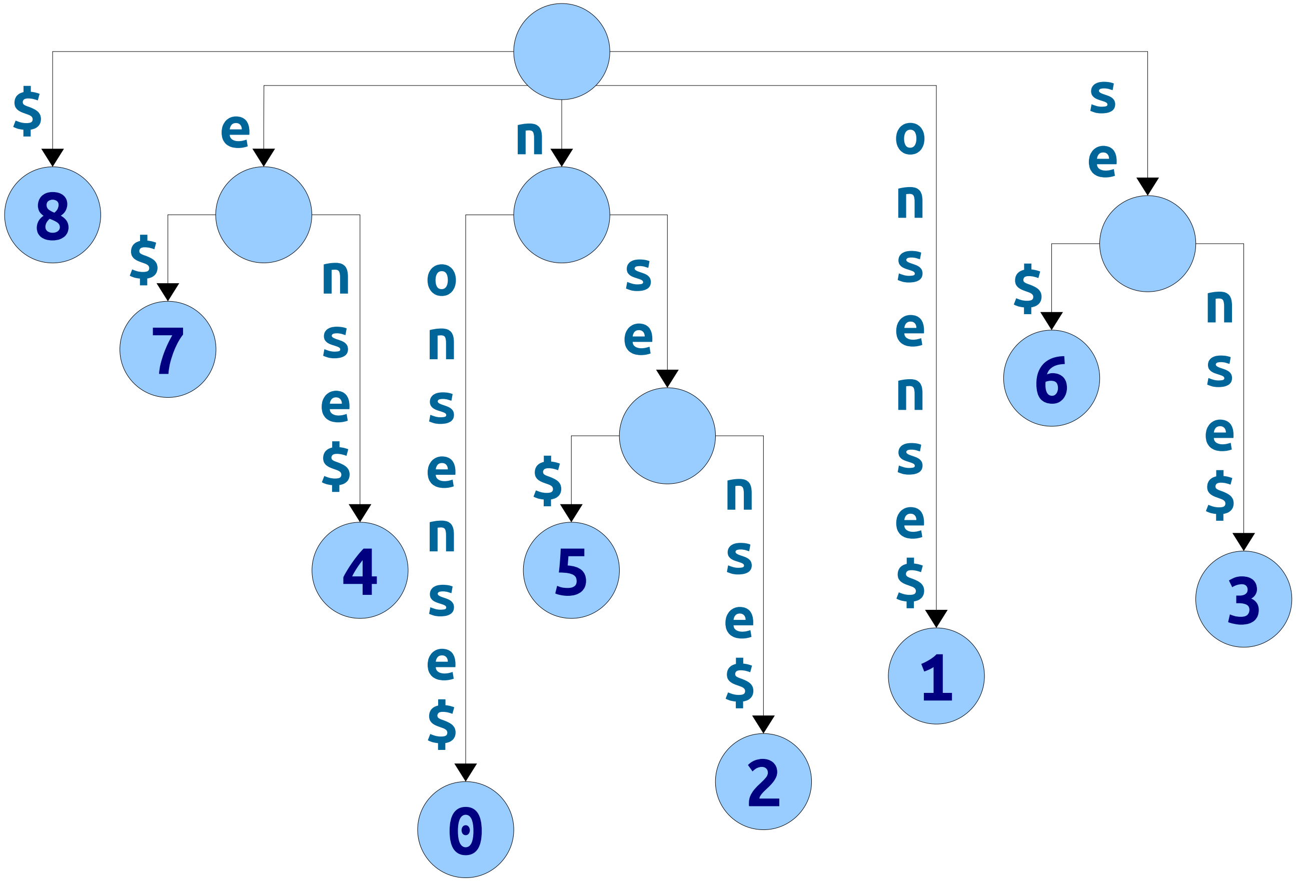
Key Intuition: The efficiency in a suffix tree is largely due to

1. keeping the suffixes in sorted order, and
2. exposing branching words.

Where We’re Going

- Today, we’ll see two data structures that encode much of the same information as suffix trees, but in much less space.
  - The ***suffix array*** stores information about the ordering of the suffixes of a string.
  - The ***LCP array*** stores information about the branching words of a string.
- Together, they’ll provide algorithms that match or are comparable to the time bounds from last time.

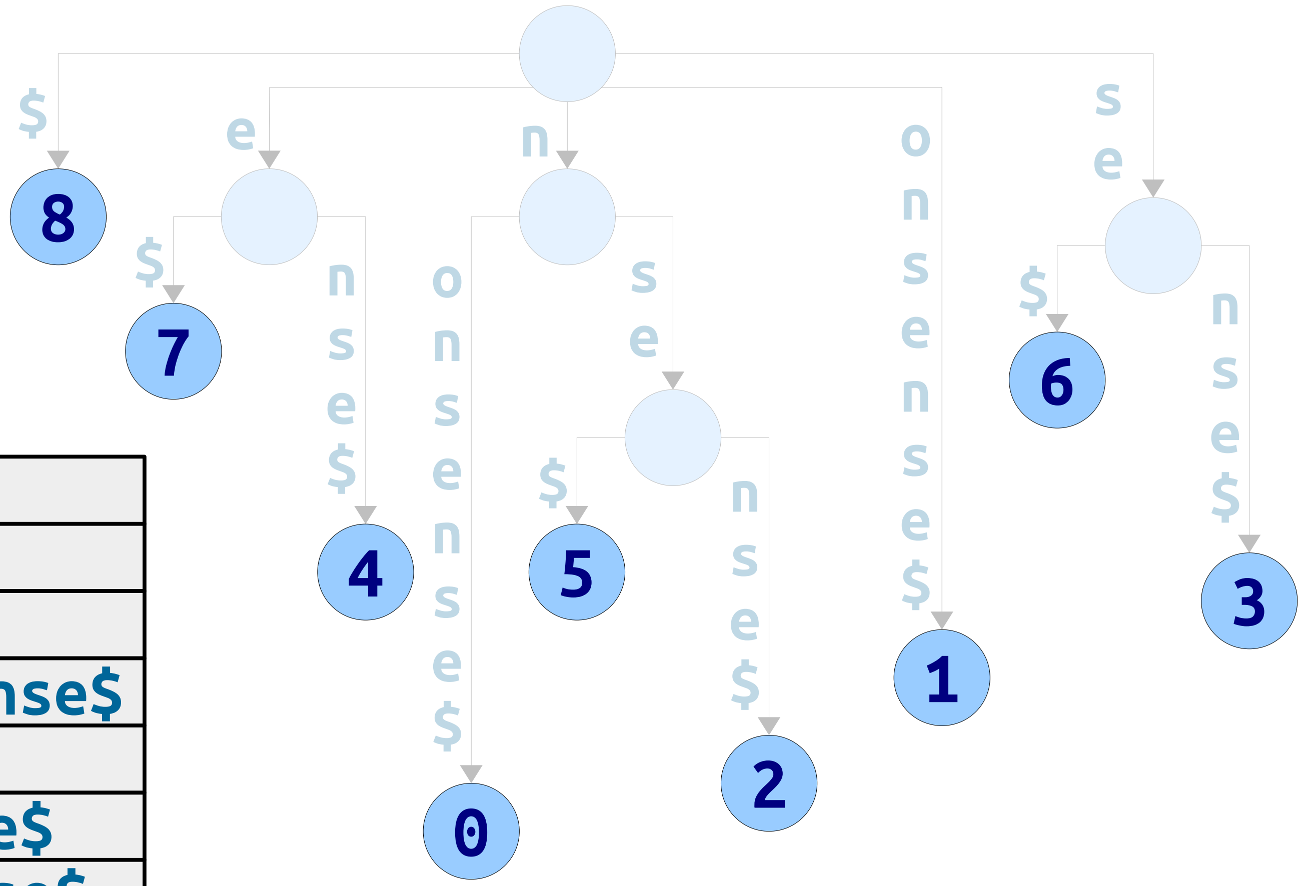
Suffix Arrays



Theorem: There is a node labeled ω in a suffix tree for *T*

if and only if

ω is a suffix of *T*\$ or ω is a branching word in *T*\$.



$
e$
ense$
nonsense$
nse$
nsense$
onsense$
se$
sense$

nonsense$
012345678

Theorem: There is a node labeled ω in a suffix tree for *T*
if and only if
ω is a suffix of *T*$ or ω is a branching word in *T*$.

Suffix Arrays

- A ***suffix array*** for a string *T* is a sorted array of the suffixes of the string *T*$.
- Suffix arrays distill out just the first component of suffix trees: they store suffixes in sorted order.
- ***Non-obvious fact:*** Suffix arrays can be built in time O(***m***). We can cover this later in the quarter if you're interested.

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Suffix Arrays

- The way we've drawn suffix arrays is terribly space-inefficient.
    - It always uses space $\Theta(\boldsymbol{m}^2)$, since that's how many total characters occur in all suffixes.
- Can we do better?

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Suffix Arrays

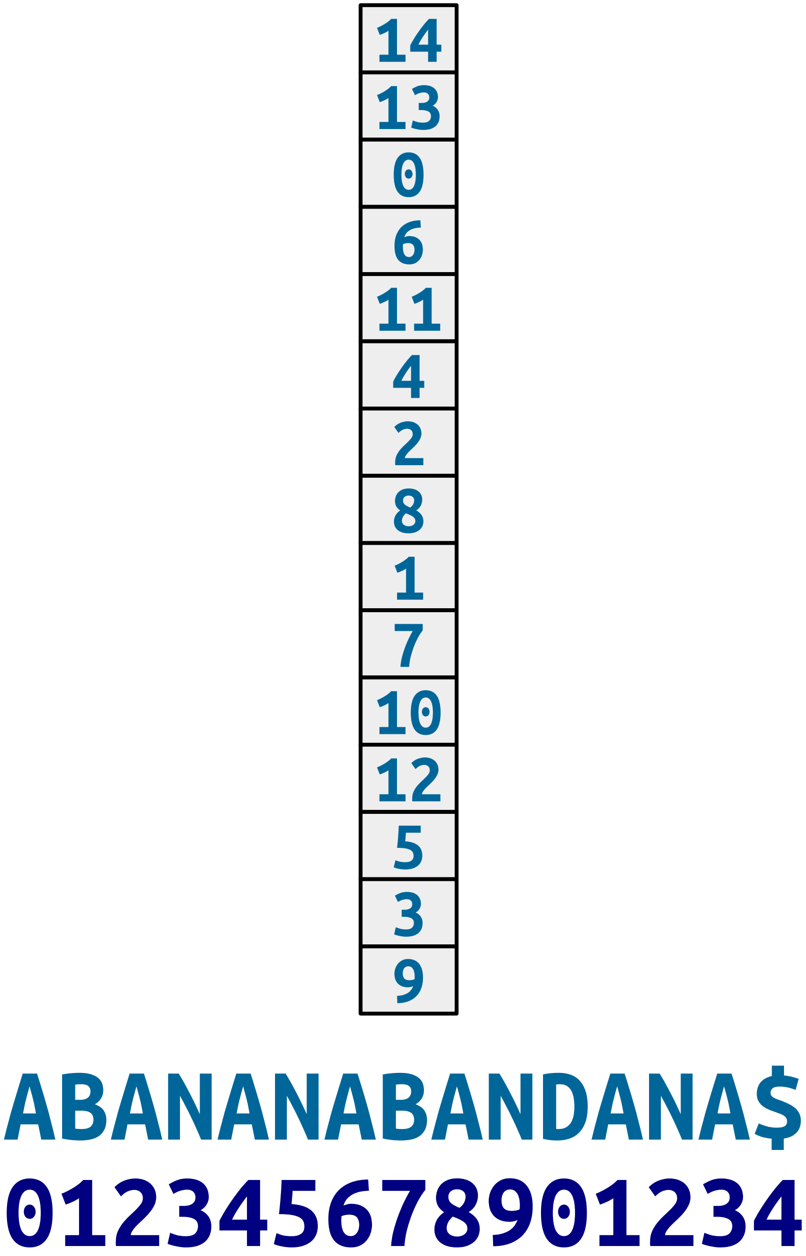
- We reduced the space usage of suffix trees by representing substrings, implicitly, as ranges within the original string.
- ***Idea:*** Don’t store the suffixes themselves. Just store the starting positions of the suffixes.
- Space: Θ(***m***), and with only one machine word used per character of input.

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

ABANANABANDANA$
012345678901234

Suffix Arrays

- Although the picture to the right is how we’d represent the suffix array in memory, for this lecture we’ll draw things out the longer way.
- This is just to build intuition; we wouldn’t actually do that in practice.



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

```
ABANANABANDANA$
012345678901234
```

Using Suffix Arrays

- Last time, we saw how to find all instances of a pattern ***P*** in a text ***T*** using suffix *trees*.
- How could we do that with suffix *arrays*?

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Using Suffix Arrays

- ***Reminder:*** Our text string ***T*** has length ***m***. Our pattern string ***P*** has length ***n***.
- ***Claim:*** With a suffix array, we can determine whether ***P*** appears in ***T*** in time O(***n*** log ***m***).

How?

Answer at

https://pollev.com/cs166spr23

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Using Suffix Arrays

- ***Reminder:*** Our text string ***T*** has length ***m***. Our pattern string ***P*** has length ***n***.
- ***Claim:*** With a suffix array, we can determine whether ***P*** appears in ***T*** in time O(***n*** log ***m***).
  - Binary search has O(log ***m***) rounds.
  - Each probe takes time O(***n***).
- This bound can be made tight. *(How?)*
- Figure that ***m*** is often much bigger than ***n***, so this is a huge win over a raw scan.

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

ANAN

Using Suffix Arrays

- ***Claim:*** With a suffix array, we can find all matches of a pattern ***P*** in ***T*** in time $O(\boldsymbol{n} \log \boldsymbol{m} + \boldsymbol{z})$, where ***z*** is the number of matches.
- ***Idea:*** Binary search can be used to find a range of values equal to some key. Adapt that idea to find all suffixes beginning with the same prefix.

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

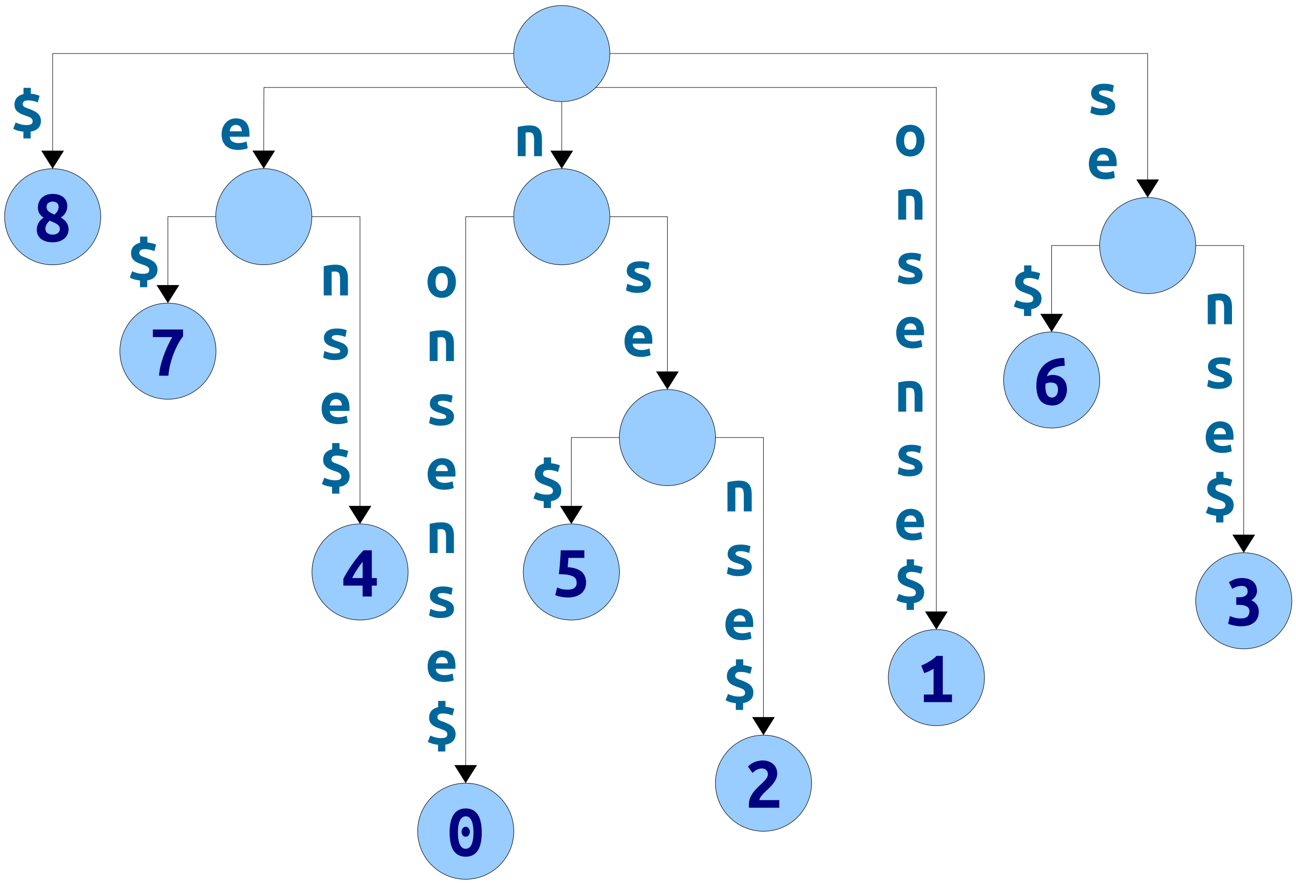
NA

The Story So Far

- Suffix arrays store all the suffixes of a string in sorted order.
- They provide an

  $$\langle O(\boldsymbol{m}), O(\boldsymbol{n} \log \boldsymbol{m} + \boldsymbol{z}) \rangle$$

  solution to the substring search problem.
- ***Intuition:*** Suffix trees are valuable in large part because they just keep the suffixes sorted.
- What else are suffix trees doing?

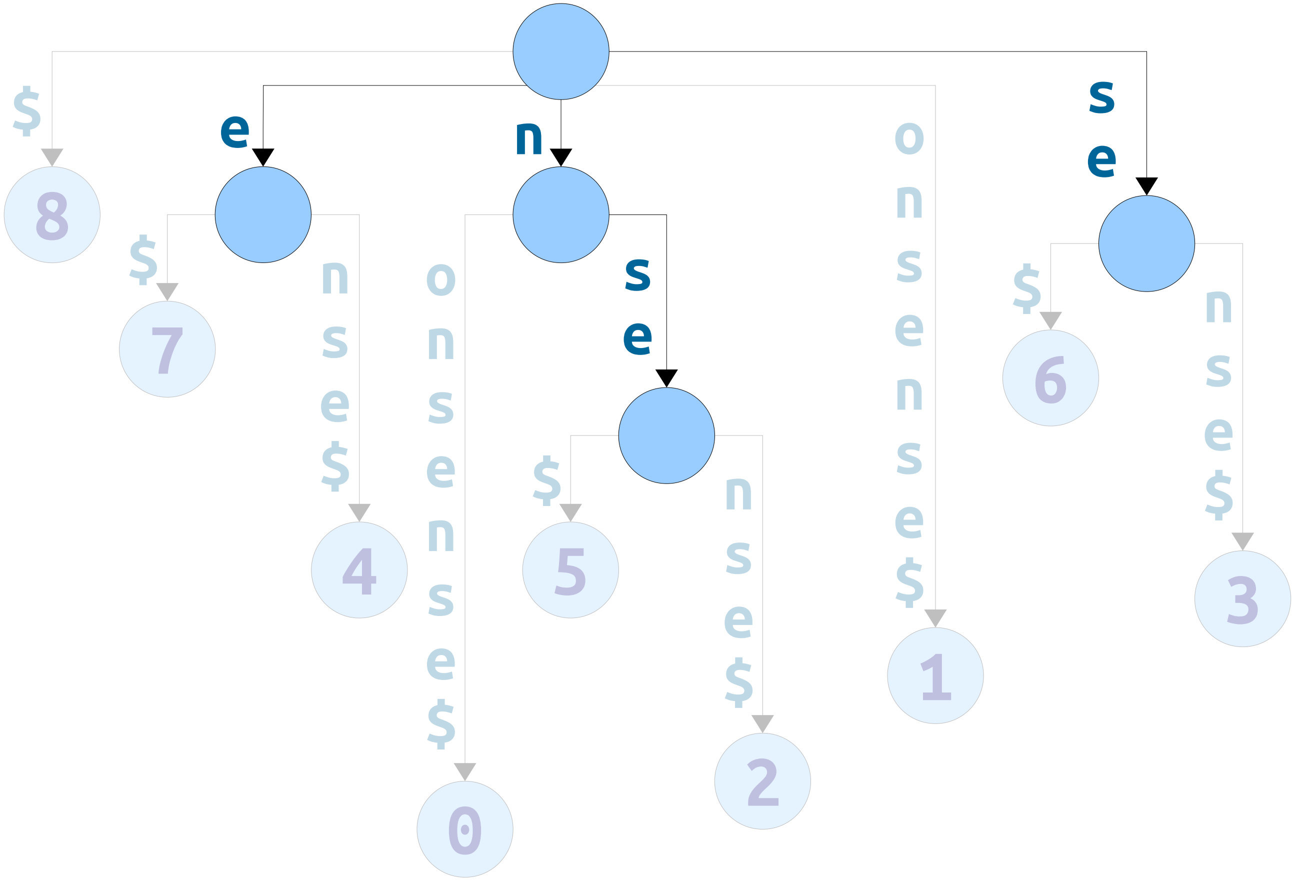


nonsense$
012345678

Theorem: There is a node labeled ω in a suffix tree for *T*

if and only if

ω is a suffix of *T*$ or ω is a branching word in *T*$.



nonsense$
012345678

Theorem: There is a node labeled ω in a suffix tree for *T*
if and only if
ω is a suffix of *T*$ or ω is a branching word in *T*$.

Branching Words

- ***Recall:*** If T is a string, then ω is a ***branching word*** in $T\$$ if there are characters $a \neq b$ such that ωa and ωb are substrings of $T\$$.

ABANANABANDANA$

Branching Words

- ***Recall:*** If T is a string, then ω is a ***branching word*** in $T\$$ if there are characters $a \neq b$ such that ωa and ωb are substrings of $T\$$.

Although `ABA` is a repeated substring, it is not a branching word because all appearances are followed by `N`.

ABANANABANDANA$

Branching Words

- ***Recall:*** If T is a string, then ω is a ***branching word*** in $T\$$ if there are characters $a \neq b$ such that ωa and ωb are substrings of $T\$$.

The substring ANANA only appears once, so it's not a branching word.

AB**ANANA**BANDANA$

Branching Words

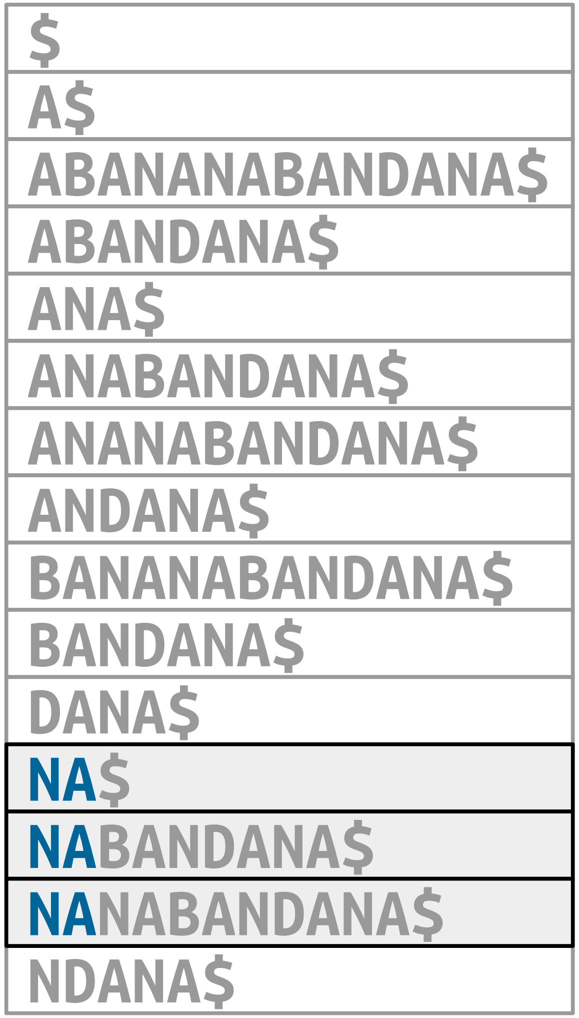
- Notice that, by sorting suffixes, we've made it easier to spot branching words.
- Specifically, all suffixes starting with a branching word will be adjacent in the suffix array.
- The branching word will be the ***longest common prefix*** (or ***LCP***) of those adjacent suffixes.

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Branching Words

- Notice that, by sorting suffixes, we’ve made it easier to spot branching words.
- Specifically, all suffixes starting with a branching word will be adjacent in the suffix array.
- The branching word will be the ***longest common prefix*** (or ***LCP***) of those adjacent suffixes.



$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

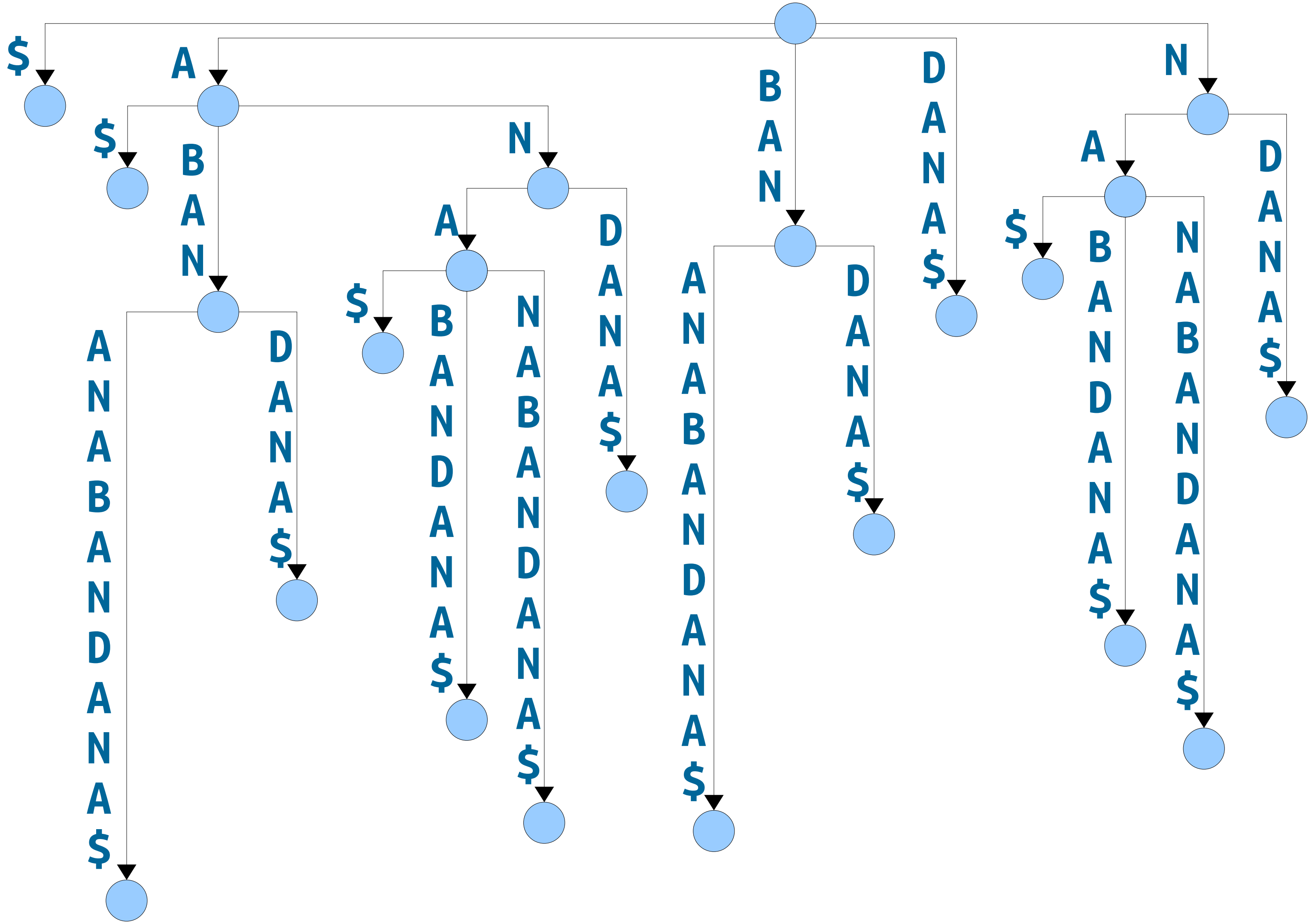
Branching Words

- ***Theorem:*** A string ω is a branching word in string *T*$ if and only if it's the longest common prefix of two adjacent suffixes in *T*'s suffix array.
- ***Proof idea:*** If ω is the longest common prefix of two adjacent suffixes, let *a* and *b* be the characters immediately following ω in those two suffixes. Then ω*a* and ω*b* are substrings of *T*$.

  If ω is branching, choose the lexicographically smallest *a* and *b* making the definition work. Then the last suffix starting with ω*a* and the first suffix starting with ω*b* are adjacent in the suffix array. ■

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$



$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

ω is an internal node in the suffix tree for T

if and only if

ω is a branching word in T$

if and only if

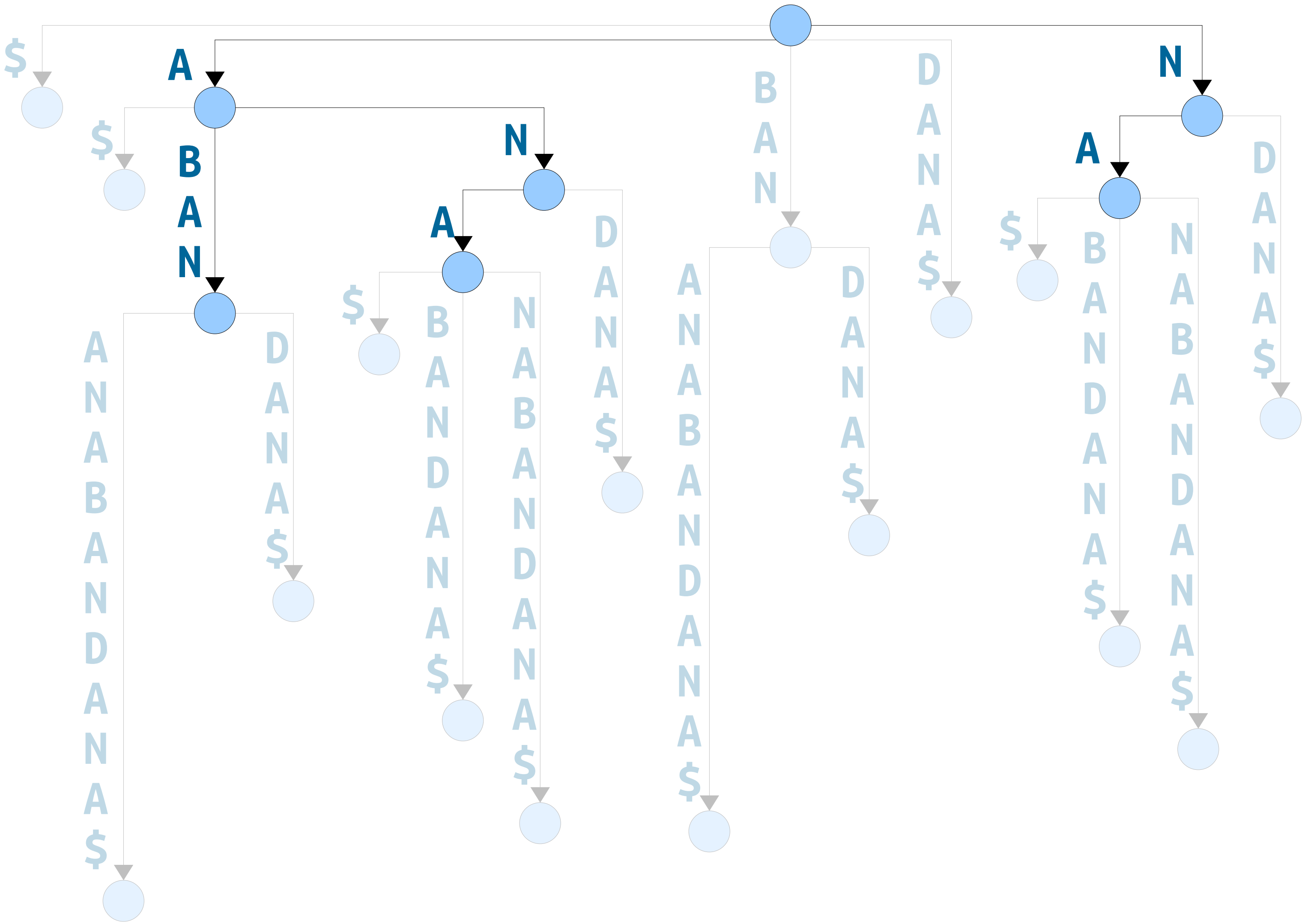
ω is the LCP of two adjacent suffixes in the suffix array for T

Key Intuition: Adjacent suffixes with long shared prefixes correspond to subtrees of the suffix tree.

Harnessing this Connection

Longest Repeated Substring

- Last time, we saw how to solve the longest repeated substring problem by using suffix trees.
- ***Algorithm:*** Find the internal node in the suffix tree with the longest label.
- ***Question:*** Can we do this with just a suffix array?



ABANANABANDANA$

Longest Repeated Substring

- We can list all branching words from a suffix array in time O($\boldsymbol{m}^2$).
  - O($\boldsymbol{m}$) pairs; each pair takes time O($\boldsymbol{m}$) to process.
- This worst-case bound can be realized.
- Contrast this with O($\boldsymbol{m}$) for a suffix tree.
- Can we do better?

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Longest Repeated Substring

- ***Observation:*** We don't actually need to know what all the branching words are to find the longest repeated substring.
- We just need to know how long they are.
- That way, we can figure out which is longest.
- Is there some nice way to do this?

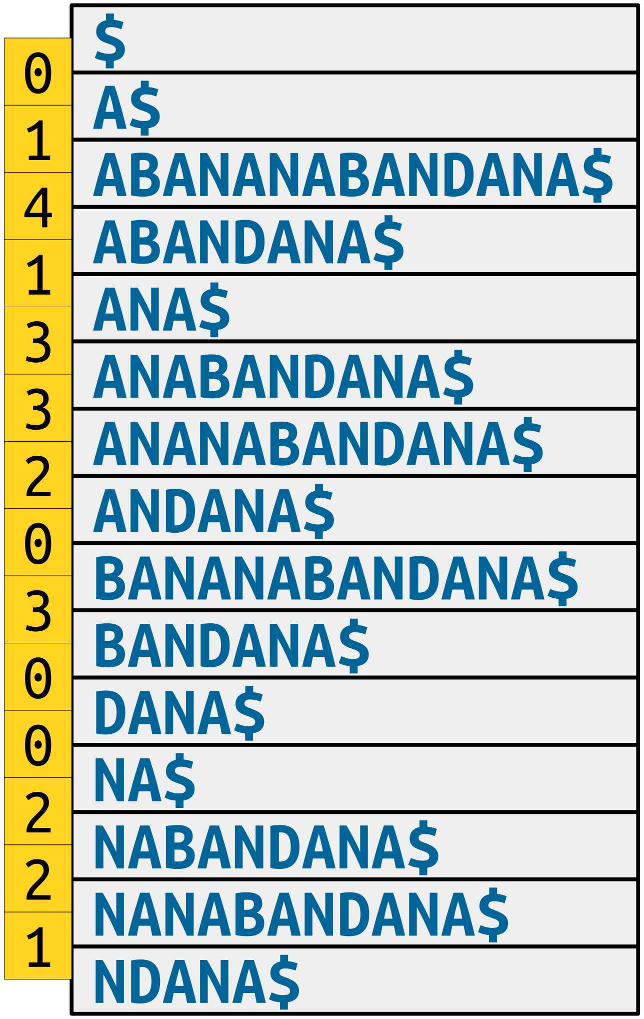
$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

LCP Arrays

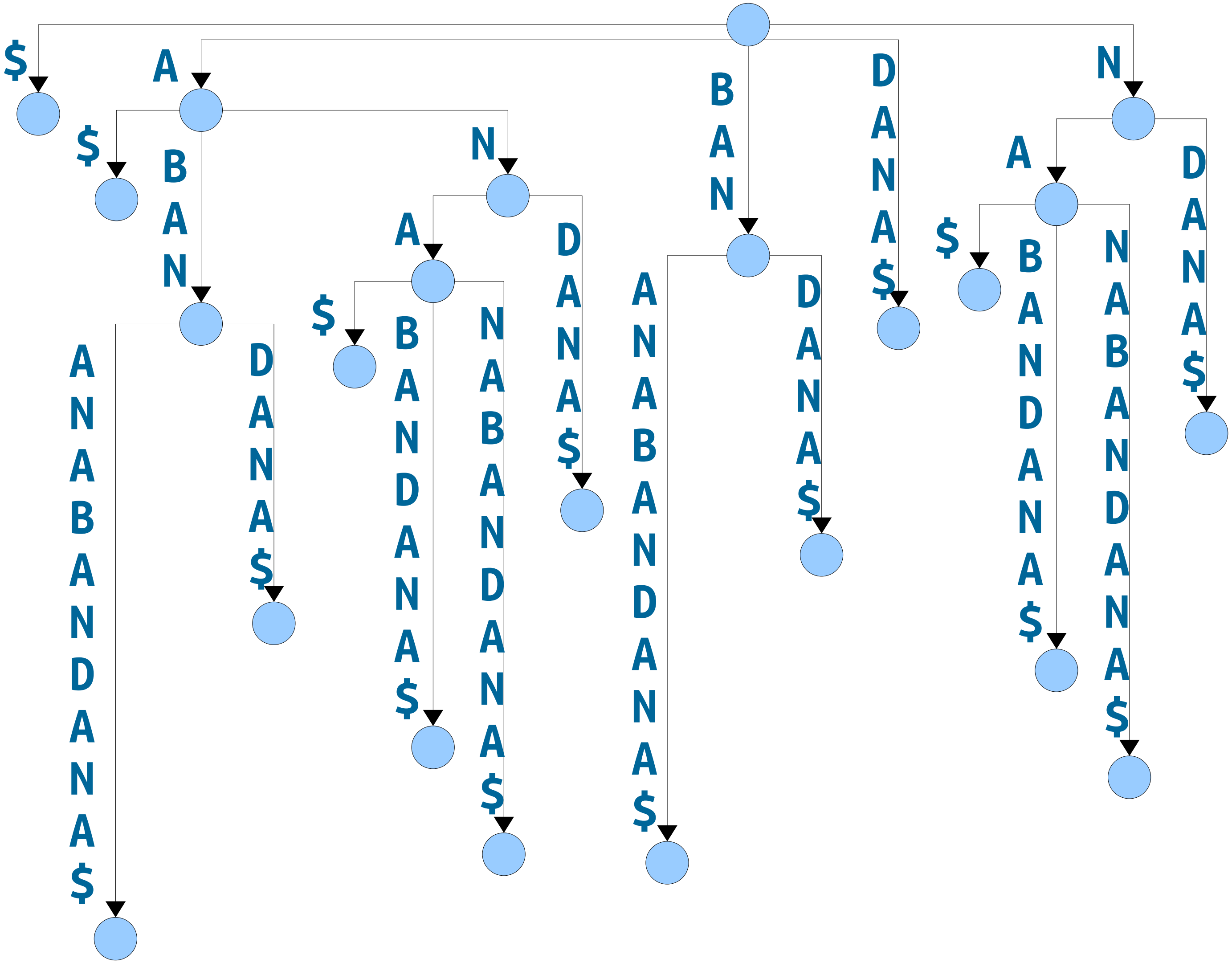
LCP Arrays

- The ***LCP array***, often denoted ***H***, is an array where $H[i]$ is the length of the LCP of the ith and $(i+1)$st suffixes in the suffix array.
- (The letter H comes from “height.”)



	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$



	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$

Key intuition: The suffix array gives the leaves of the suffix tree.
The LCP array gives the internal nodes of the suffix tree.

Using LCP Arrays

- If you already have a suffix array and LCP array, you can solve longest repeated substring in time O(***m***):
  - Find the largest element in the LCP array.
  - Return the string it corresponds to.
- ***Question:*** How fast can we construct an LCP array?

	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$

Building LCP Arrays

Building LCP Arrays

- It never hurts to start with the naive algorithm and see what happens!
- ***Algorithm:*** For each consecutive pair of strings in the suffix array, compute the length of their longest common prefix.
- We can upper-bound the runtime at O(***m***^{2}).
- ***Question:*** Can we realize this upper bound?

	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$

Building LCP Arrays

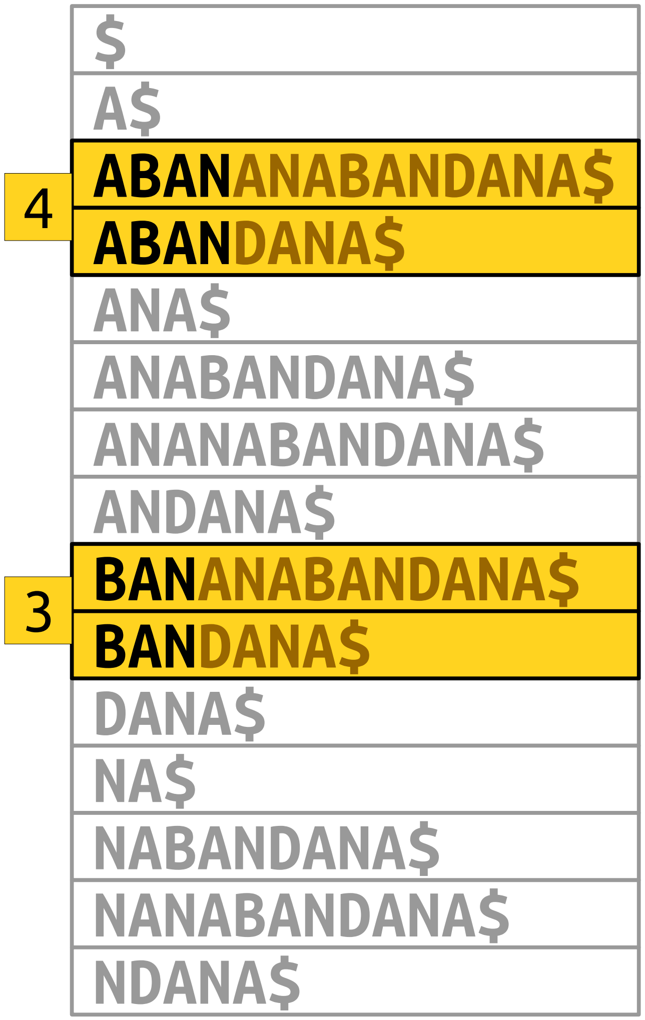
- Why is our naive algorithm slow?
- ***Intuition:*** We aren’t able to carry work from one suffix over to the next.

	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$

Building LCP Arrays

- ***Key intuition:*** Suffixes overlap one another! It should be possible to share LCP information across suffixes.
- For example, suppose we compute the LCP entry shown here.
- Look at the suffixes formed by dropping the first letter of these two suffixes.
- What do we know about their LCP?



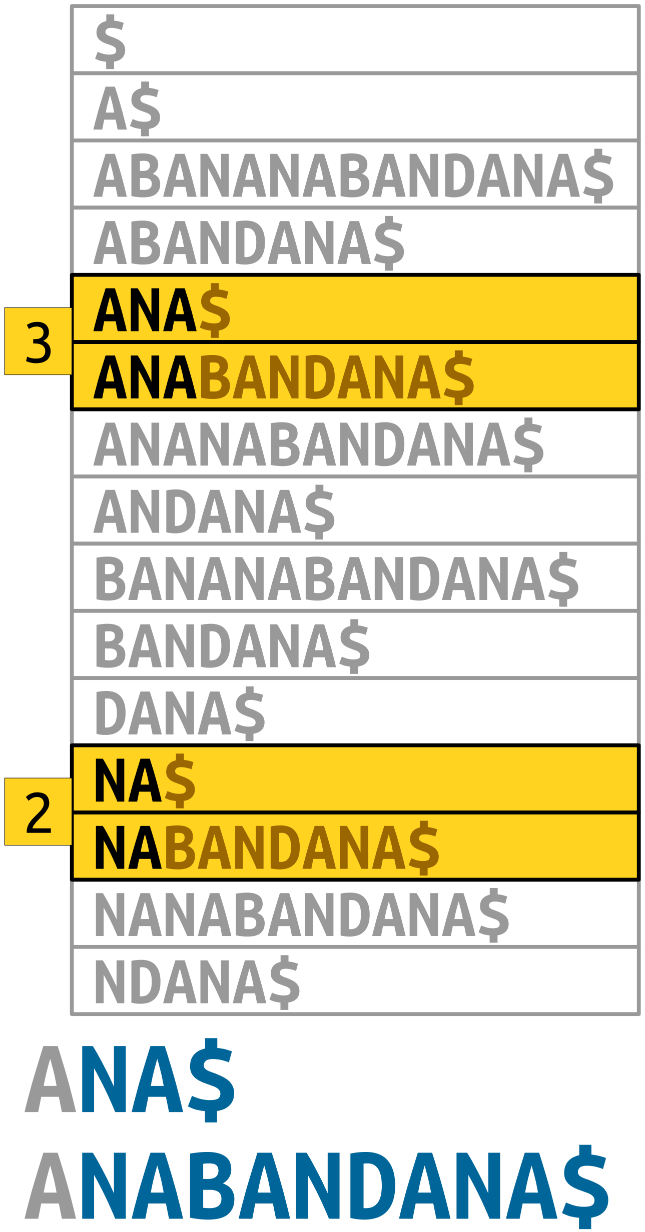
$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

ABANDANA$

Building LCP Arrays

- Let’s do another example. Suppose we know the LCP of these suffixes.
- As before, drop the first letter from each suffix.
- What can we say about the LCP of the resulting suffixes?

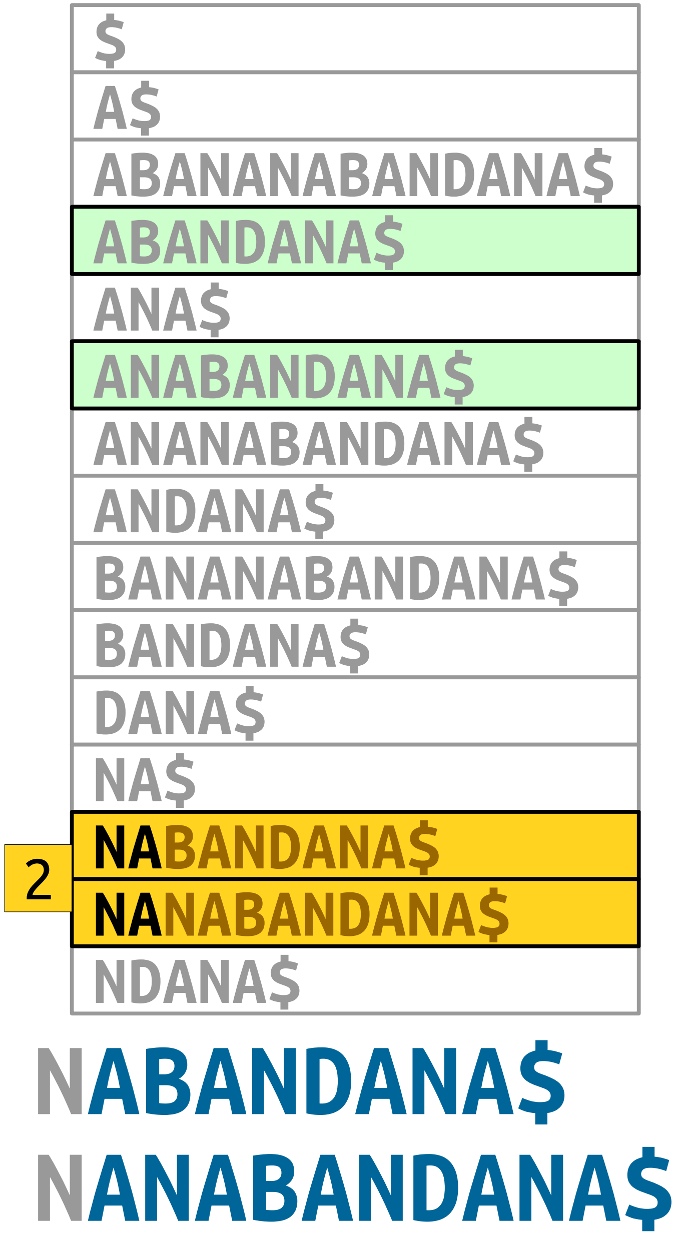


$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ANA$
ANABANDANA$

Building LCP Arrays

- Sometimes, in dropping the first letter, two adjacent suffixes get spread out.



$

A$

ABANANABANDANA$

ABANDANA$

ANA$

ANABANDANA$

ANANABANDANA$

ANDANA$

BANANABANDANA$

BANDANA$

DANA$

NA$

NABANDANA$

2

NANABANDANA$

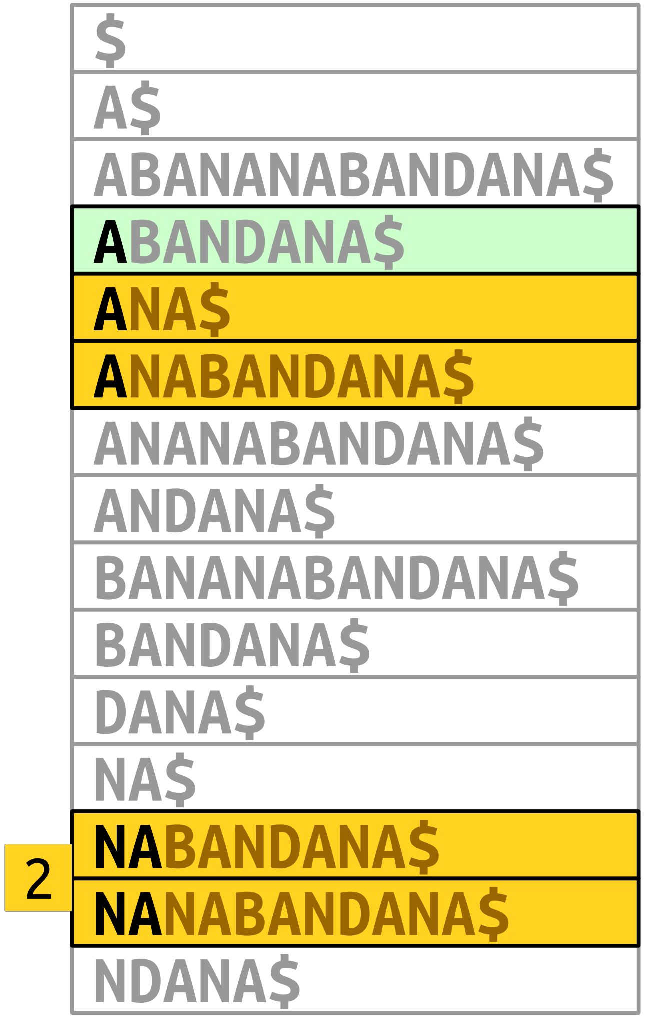
NDANA$

NABANDANA$

NANABANDANA$

Building LCP Arrays

- Sometimes, in dropping the first letter, two adjacent suffixes get spread out.
- ***Claim:*** Look at the second suffix in the pair. Its LCP with the suffix before it is at least the previous LCP minus one.
- Think about the suffix tree. The two shorter suffixes are in the same subtree, so everything between them is also in that subtree.

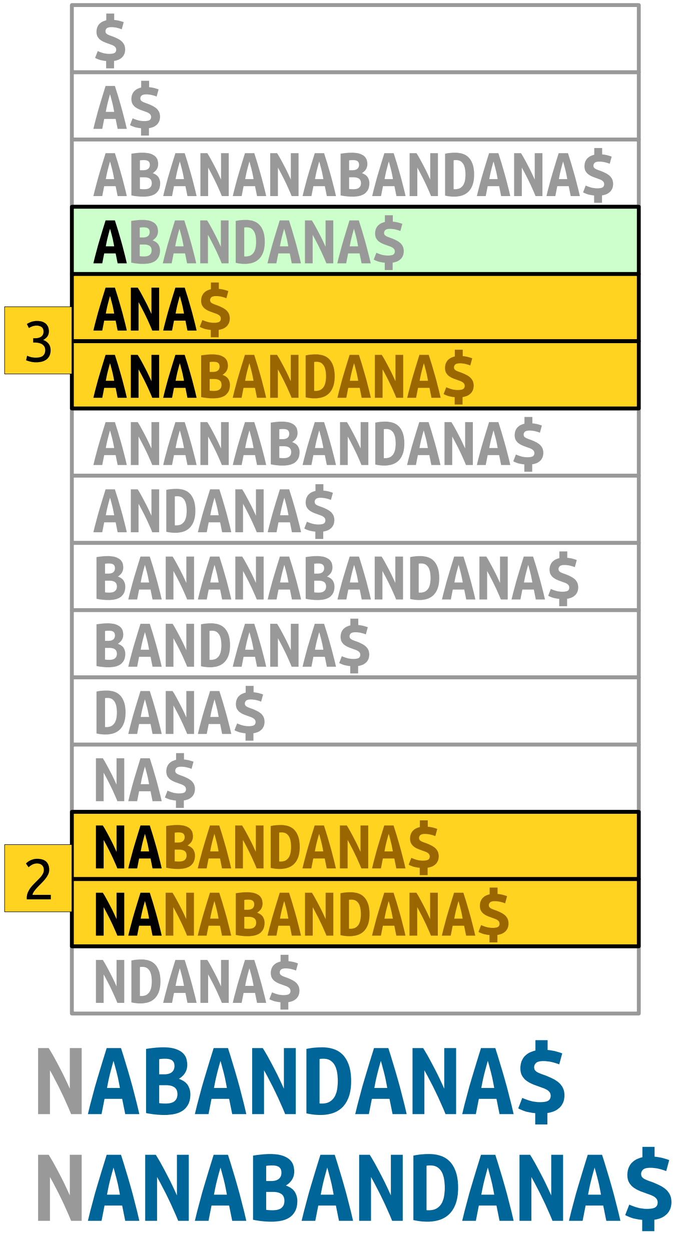


$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
2 NABANDANA$
NANABANDANA$
NDANA$

NABANDANA$
NANABANDANA$

Building LCP Arrays

- We know that these two new suffixes must have an LCP of at least 1, because the two old suffixes have an LCP of 2.
- However, the LCP may be longer than 1, since we've never seen one of these two suffixes.
- We still need to some some scanning, but we won't necessarily have to rescan the entire suffix.



$
A$
ABANANABANDANA$
ABANDANA$
3
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
2
NABANDANA$
NANABANDANA$
NDANA$

NABANDANA$
NANABANDANA$

Kasai’s Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

$
A$
ABANANABANDANA$
ABANDANA$
ANA$
ANABANDANA$
ANANABANDANA$
ANDANA$
BANANABANDANA$
BANDANA$
DANA$
NA$
NABANDANA$
NANABANDANA$
NDANA$

ABANANABANDANA$

Kasai’s Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

	$
0	A$
1	ABANANABANDANA$
4	ABANDANA$
1	ANA$
3	ANABANDANA$
3	ANANABANDANA$
2	ANDANA$
0	BANANABANDANA$
3	BANDANA$
0	DANA$
0	NA$
2	NABANDANA$
2	NANABANDANA$
1	NDANA$

ABANANABANDANA$

Kasai's Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

With O(m) preprocessing time, can be done in time O(1).

Question to Ponder:
How would you do this?

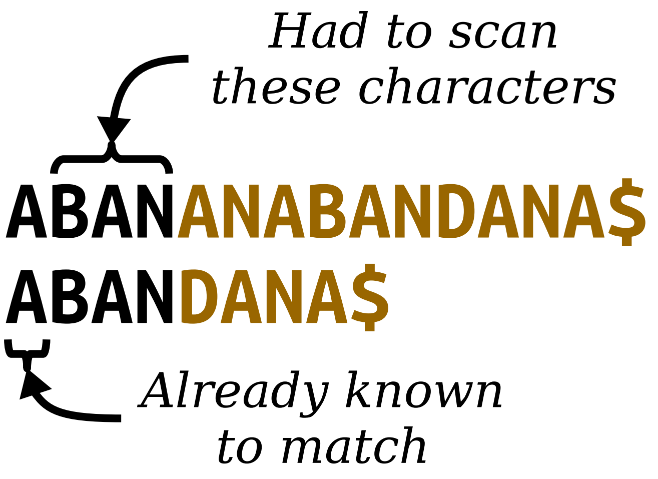
Kasai's Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

With O(*m*) preprocessing time, can be done in time O(1).

Question to Ponder:
How would you do this?

The runtime of this step is proportional to how much the LCP increases on that step.



Kasai's Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

With O(m) preprocessing time, can be done in time O(1).

Question to Ponder:
How would you do this?

The runtime of this step is proportional to how much the LCP increases on that step.

The LCP value decreases by at most one per suffix. *(We saw this earlier.)*

The LCP value maxes out at m. *(Can't match more than all the characters.)*

Therefore, the LCP value can grow at most $2m$ times. *(Prove this!)*

Claim: Across all iterations, this step takes a total of O(m) time.

Kasai’s Algorithm

- For each suffix of the original string, except the last:
  - Find that suffix in the suffix array.
  - Look at the suffix that comes before it.
  - (★) Find the length of the longest common prefix of those suffixes.
  - Write that down in the *H* array.
- Use the insight from the previous slides to speed up step (★).

Total runtime: O(***m***).

More to Explore

- We could easily spend a whole quarter talking about suffix arrays. Here’s what we didn’t cover:
  - ***Bottom-up tree simulations:*** Using LCP arrays, you can simulate any O($\boldsymbol{m}$)-time suffix tree algorithm that works with a bottom-up DFS in time O($\boldsymbol{m}$).
  - ***Faster substring searching:*** Using LCP arrays, plus RMQ, you can improve the cost of a substring search to O($\boldsymbol{n}$ + $\boldsymbol{z}$ + log $\boldsymbol{m}$).
  - ***Burrows-Wheeler transforms:*** Suffix arrays, plus LCP arrays, can be used to significantly improve the performance of text compressors.
- Check these out – they’re super interesting!

Next Time

- ***Amortized Analysis***
  - Lying in a runtime analysis.
- ***The Potential Method***
  - Physics meets data structure design.