Suffix Arrays

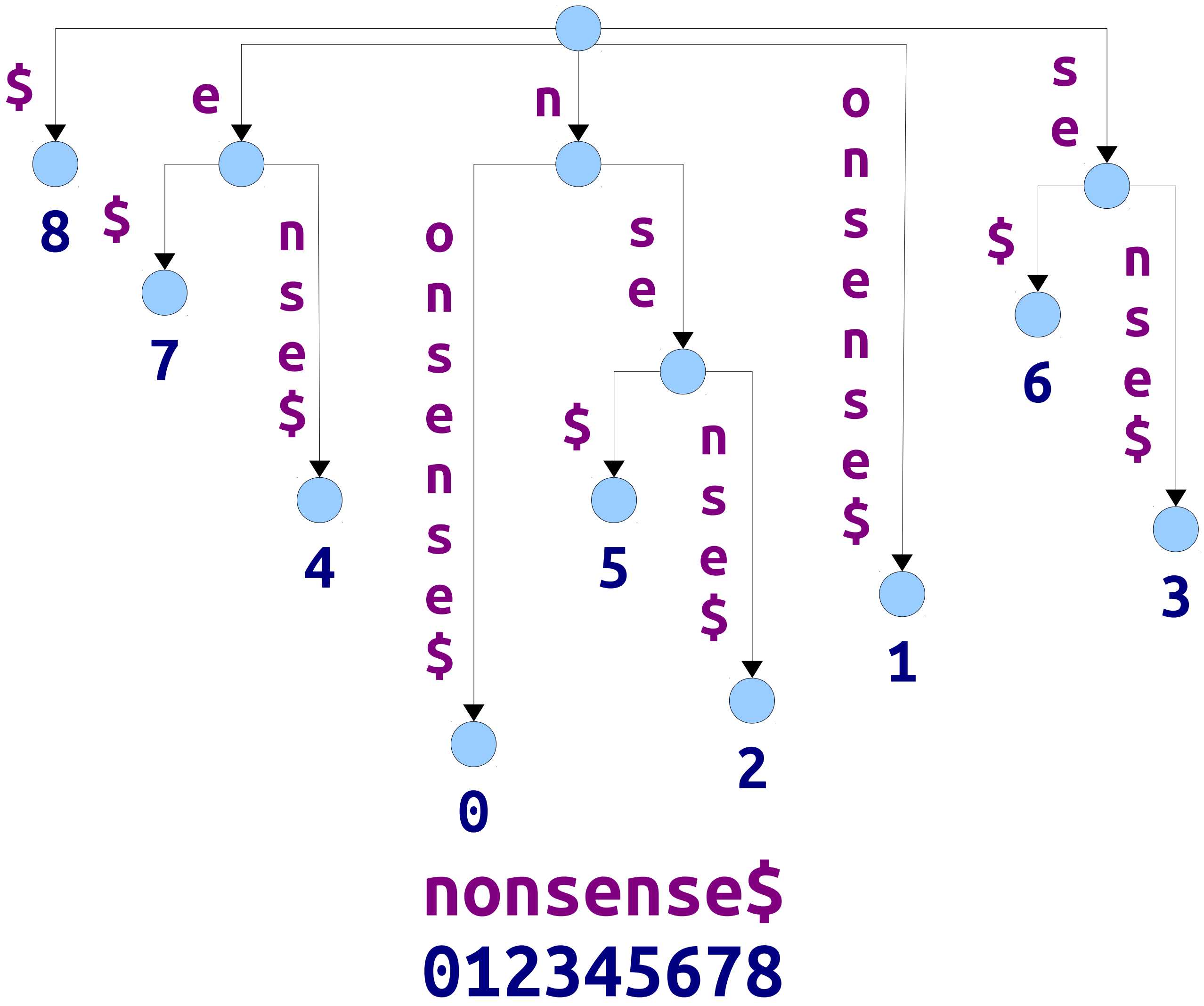
Outline for Today

- **Review from Last Time**
    - Quick review of suffix trees.
- **Suffix Arrays**
    - A space-efficient data structure for substring searching.
- **LCP Arrays**
    - A surprisingly helpful auxiliary structure.
- **Constructing Suffix Trees**
    - Converting from suffix arrays to suffix trees.
- **Constructing Suffix Arrays**
    - An extremely clever algorithm for building suffix arrays.

Review from Last Time

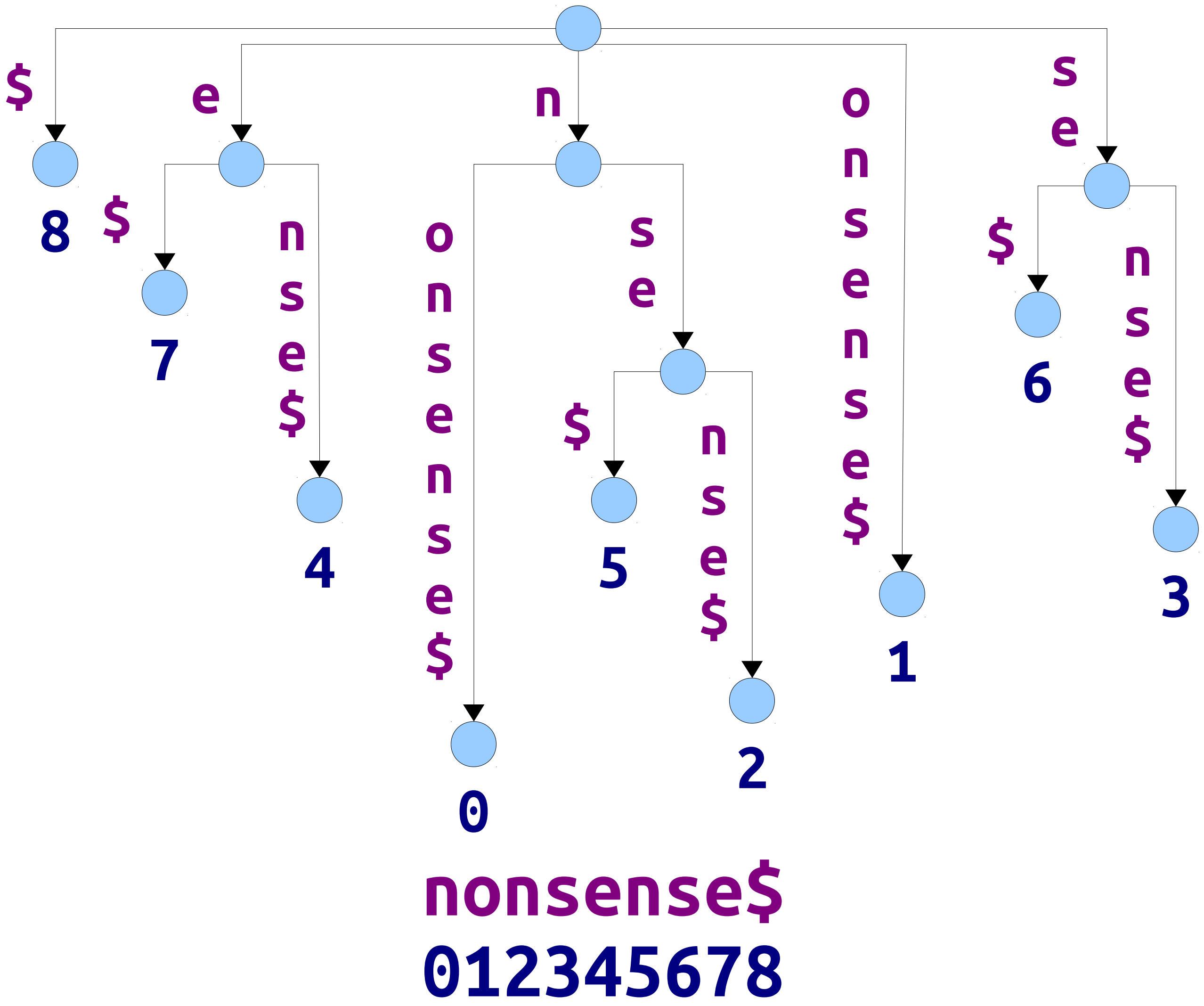
Suffix Trees

- A ***suffix tree*** for a string *T* is an Patricia trie of *T*\$ where each leaf is labeled with the index where the corresponding suffix starts in *T*\$.



Suffix Trees

- If $|T| = m$, the suffix tree has exactly $m + 1$ leaf nodes.
- For any $T \neq \varepsilon$, all internal nodes in the suffix tree have at least two children.
- Number of nodes in a suffix tree is $\Theta(m)$.



Space Usage

- Suffix trees are memory hogs.
- Suppose Σ = {`A`, `C`, `G`, `T`, `$`}.
- Each internal node needs 15 machine words: for each character, we need three words for the start/end index of the label and for a child pointer.
- This is still O(m), but it's a huge hidden constant.

Suffix Arrays

Suffix Arrays

- A ***suffix array*** for a string *T* is an array of the suffixes of *T*$, stored in sorted order.
- By convention, $ precedes all other characters.

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

Representing Suffix Arrays

- Suffix arrays are typically represented implicitly by just storing the indices of the suffixes in sorted order rather than the suffixes themselves.
- Space required: $\Theta(m)$.
- More precisely, space for $T\$$, plus one extra word for each character.

8
7
4
0
5
2
1
6
3

nonsense$

Searching a Suffix Array

- ***Recall:*** P is a substring of T iff it's a prefix of a suffix of T.
- All matches of P in T have a common prefix, so they'll be stored consecutively.
- Can find all matches of P in T by doing a binary search over the suffix array.

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

Analyzing the Runtime

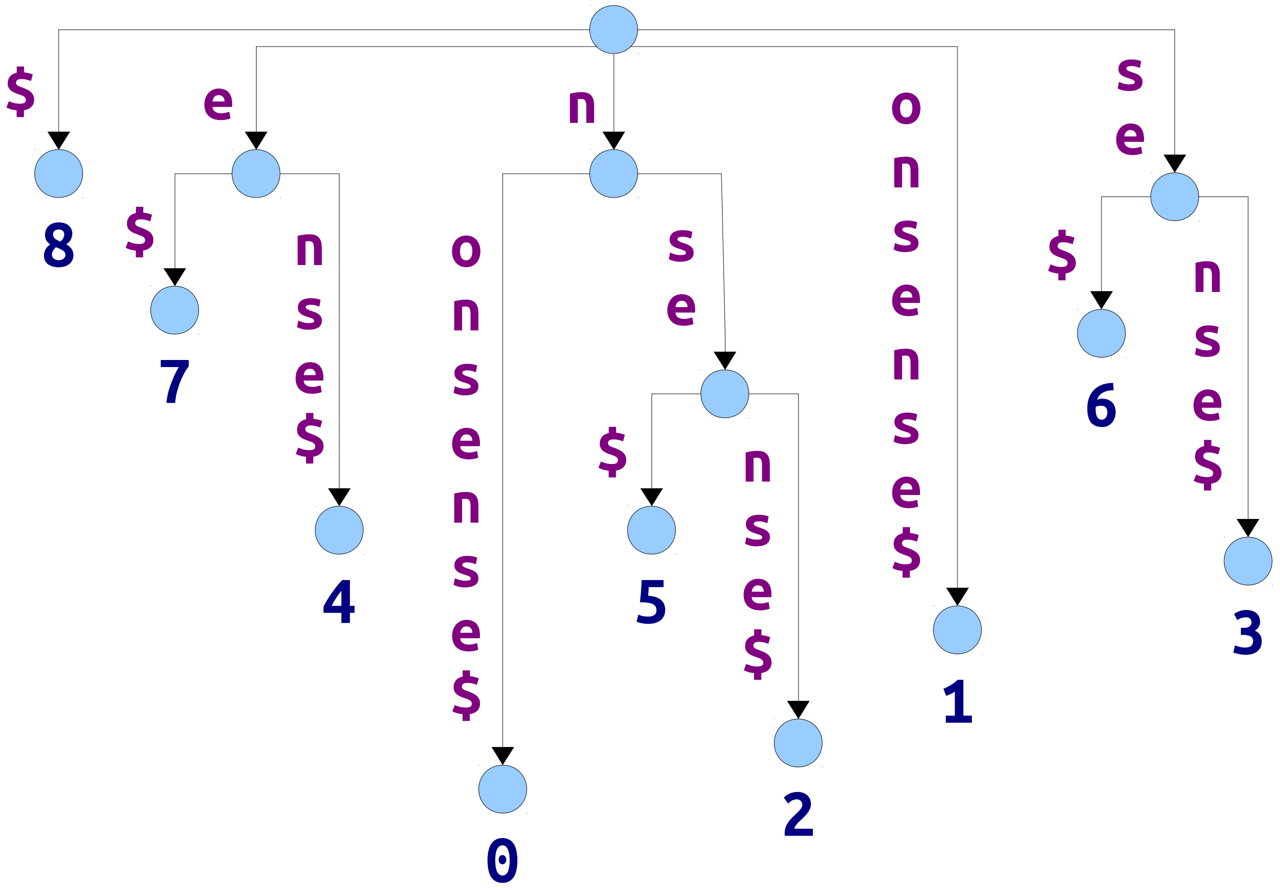
- The binary search will require O(log m) probes into the suffix array.
- Each comparison takes time O(n): have to compare P against the current suffix.
- Time for binary searching: O(n log m).
- Time to report all matches after that point: O(z).
- Total time: **O(n log m + z)**.

Why the Slowdown?

A Loss of Structure

- Many algorithms on suffix trees involve looking for internal nodes with various properties:
  - Longest repeated substring: internal node with largest string depth.
  - Longest common extension: lowest common ancestor of two nodes.
- Because suffix arrays do not store the tree structure, we lose access to this information.

Suffix Trees and Suffix Arrays

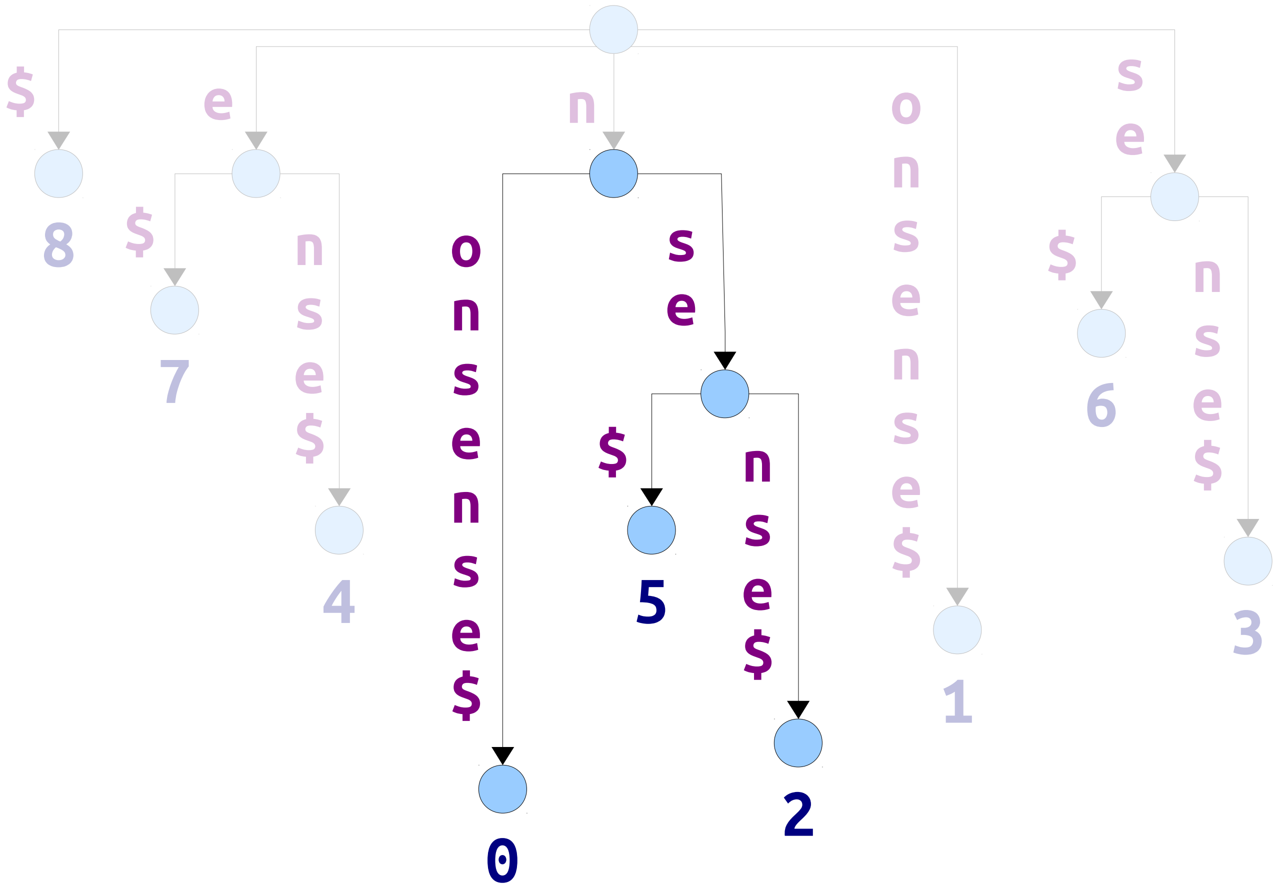


8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

Nifty Fact: The suffix array can be constructed from an ordered DFS over a suffix tree!

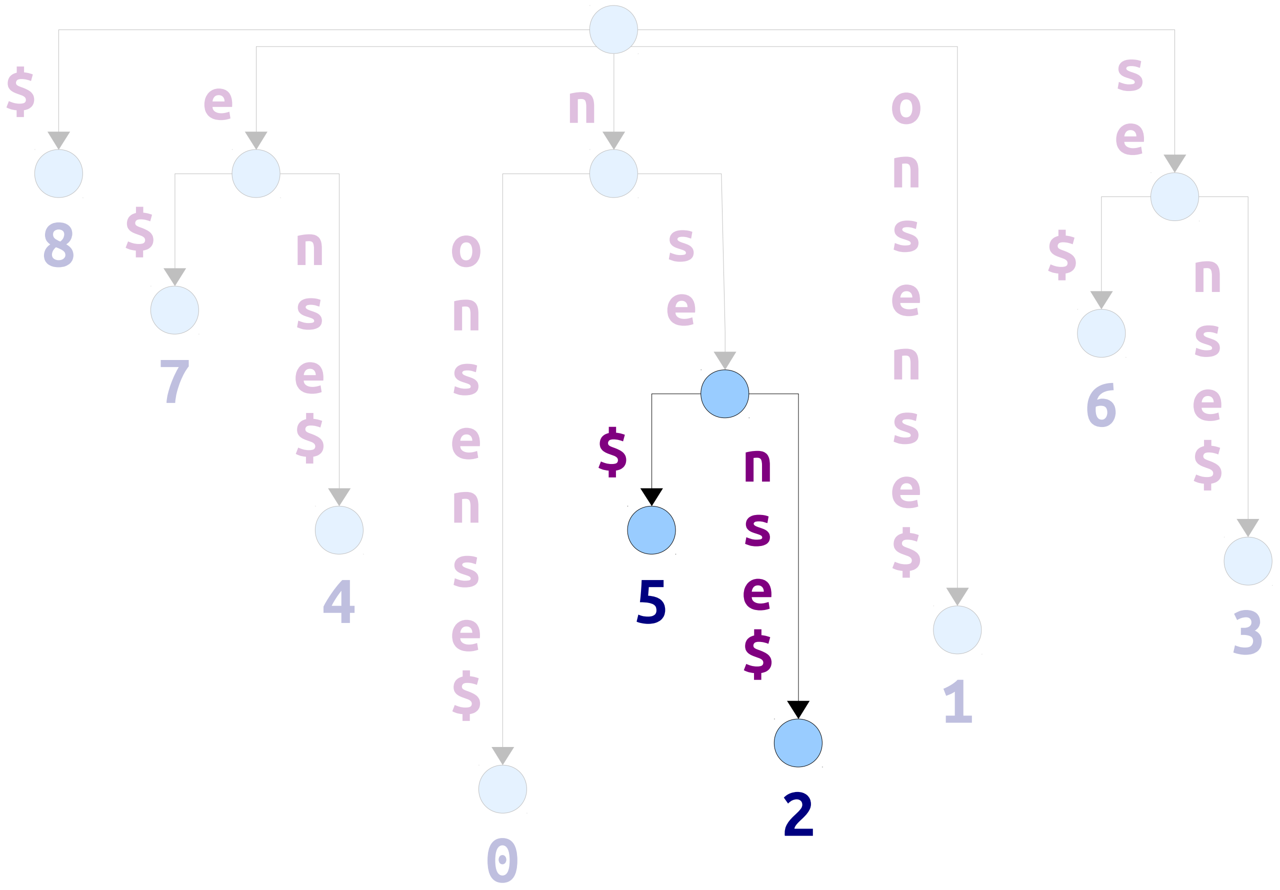
Suffix Trees and Suffix Arrays



nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

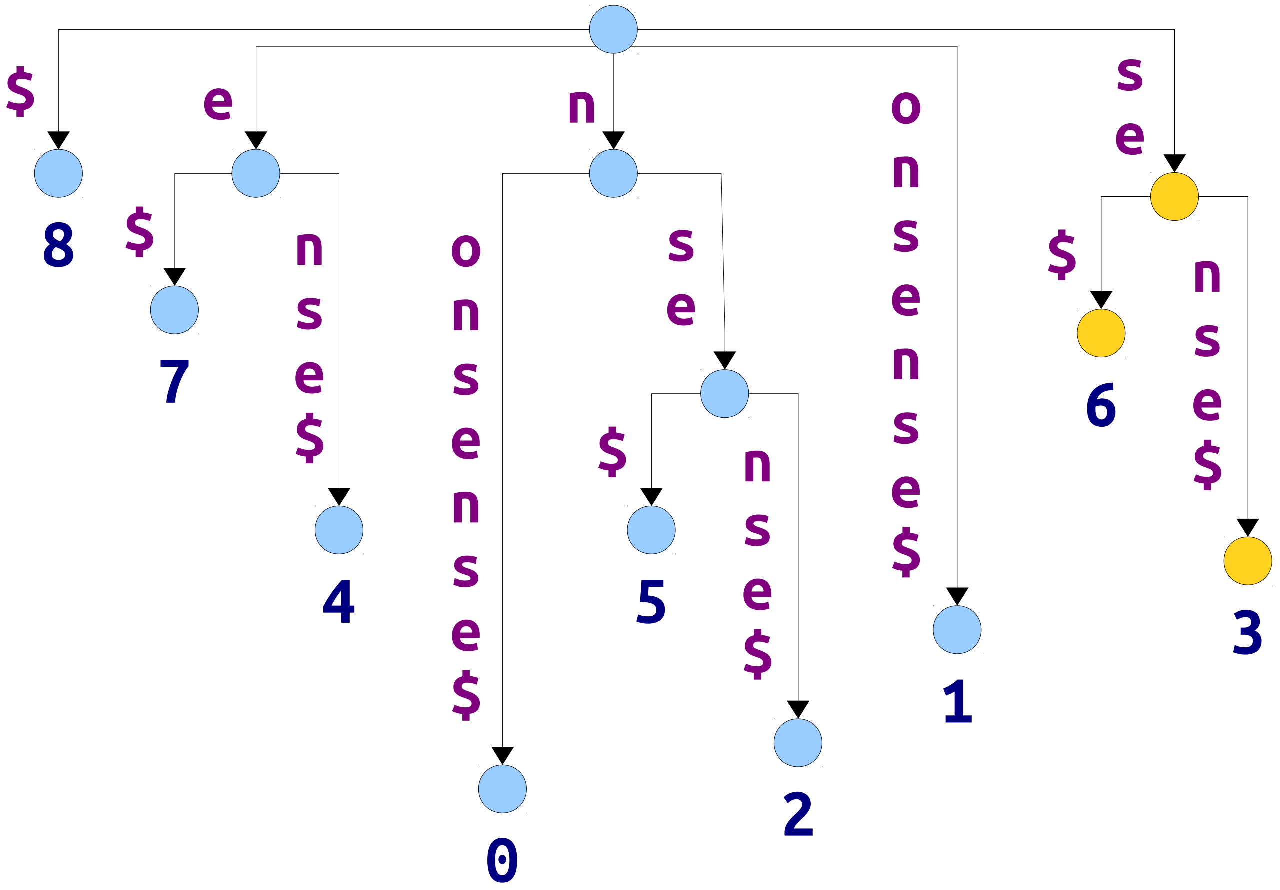
nonsense$
012345678

Nifty Fact: Adjacent strings with a common prefix correspond to subtrees in the suffix tree.

Longest Common Prefixes

- Given two strings *x* and *y*, the ***longest common prefix*** or (***LCP***) of *x* and *y* is the longest prefix of *x* that is also a prefix of *y*.
- The LCP of *x* and *y* is denoted lcp(*x*, *y*).
- LCP information is fundamentally important for suffix arrays. With it, we can implicitly recover much of the structure present in suffix trees.

Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

Nifty Fact: The lowest common ancestor of suffixes *x* and *y* has string label given by lcp(*x*, *y*).

Computing LCP Information

- ***Claim:*** There is an O(m)-time algorithm for computing LCP information on a suffix array.
- Let's see how it works.

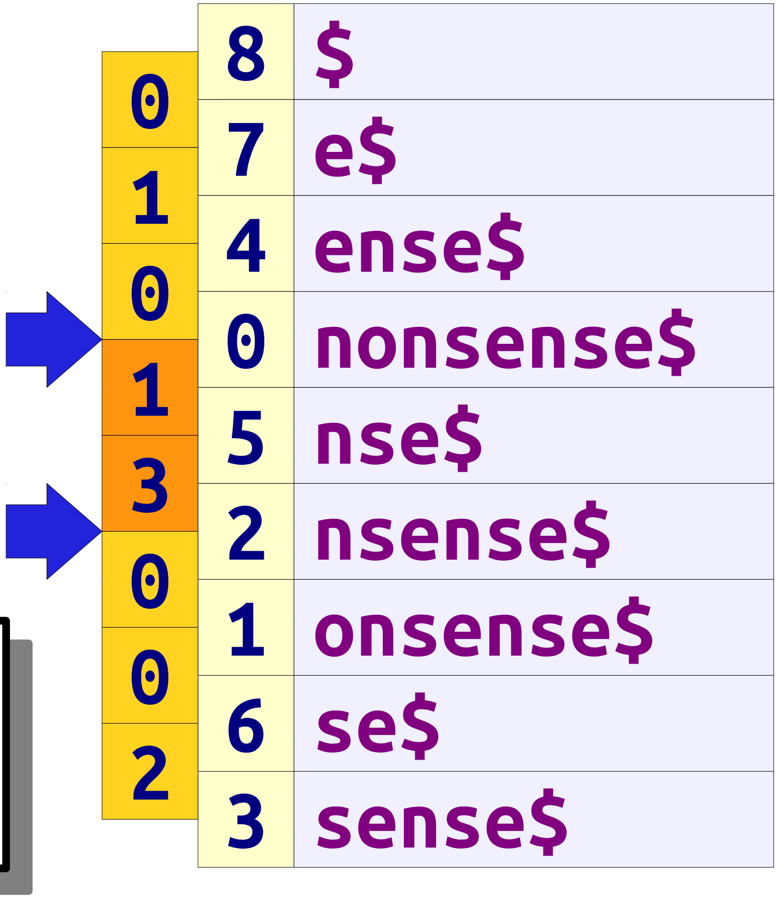
Pairwise LCP

- ***Fact:*** There is an algorithm (due to Kasai et al.) that constructs, in time O(*m*), an array of the LCPs of adjacent suffix array entries.
- The algorithm isn't that complex, but the correctness argument is a bit nontrivial.

	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

Pairwise LCP

- Some notation:
  - SA[i] is the ith suffix in the suffix array.
  - H[i] is the value of lcp(SA[i], SA[$i + 1$])



0	8	$
1	7	e$
0	4	ense$
1	0	nonsense$
3	5	nse$
0	2	nsense$
0	1	onsense$
2	6	se$
	3	sense$

Claim: For any $0 < i < j < m$:

$$\text{lcp}(\text{SA}[i], \text{SA}[j]) = \text{RMQ}_H(i, j - 1)$$

Computing LCPs

- To preprocess a suffix array to support O(1) LCP queries:
  - Use Kasai's O(m)-time algorithm to build the LCP array.
  - Build an RMQ structure over that array in time O(m) using Fischer-Heun.
  - Use the precomputed RMQ structure to answer LCP queries over ranges.
- Requires O(m) preprocessing time and only O(1) query time.

Searching a Suffix Array

- ***Recall:*** Can search a suffix array of T for all matches of a pattern P in time O(n log m + z).
- If we've done O(m) preprocessing to build the LCP information, we can speed this up.

Searching a Suffix Array

- Intuitively, simulate doing a binary search of the leaves of a suffix tree, remembering the deepest subtree you've matched so far.
- At each point, if the binary search probes a leaf outside of the current subtree, skip it and continue the binary search in the direction of the current subtree.
- To implement this on an actual suffix array, we use LCP information to implicitly keep track of where the bounds on the current subtree are.

Searching a Suffix Array

- ***Claim:*** The algorithm we just sketched runs in time O(n + log m + z).
- ***Proof idea:*** The O(log m) term comes from the binary search over the leaves of the suffix tree. The O(n) term corresponds to descending deeper into the suffix tree one character at a time. Finally, we have to spend O(z) time reporting matches.

Longest Common Extensions

Another Application: LCE

- ***Recall:*** The longest common extension of two strings T_1 and T_2 at positions i and j, denoted $\mathrm{LCE}_{T_1, T_2}(i, j)$, is the length of the longest substring of T_1 and of T_2 that begins at position i in T_1 and position j in T_2.

a	p	p	e	n	d
p	e	n	p	a	l

- Using generalized suffix trees and LCA, we have an ⟨O(m), O(1)⟩-time solution to LCE.
- ***Claim:*** There's a much easier solution using LCP.

Suffix Arrays and LCE

- ***Recall:*** $\text{LCE}_{T_1, T_2}(i, j)$ is the length of the longest common prefix of the suffix of T_1 starting at position i and the suffix of T_2 starting at position j.
- Suppose we construct a ***generalized suffix array*** for T_1 and T_2 augmented with LCP information. We can then use LCP to answer LCE queries in time O(1).
- We'll need a table mapping suffixes to their indices in the table to do this, but that's not that hard to set up.

0	1	8	$\$_1$
0	2	5	$\$_2$
1	1	7	e$\$_1$
1	2	4	e$\$_2$
4	1	4	ense$\$_1$
0	2	1	ense$\$_2$
1	1	0	nonsense$\$_1$
3	1	5	nse$\$_1$
3	2	2	nse$\$_2$
0	1	2	nsense$\$_1$
0	1	1	onsense$\$_1$
2	1	6	se$\$_1$
2	2	3	se$\$_2$
0	1	3	sense$\$_1$
	2	0	tense$\$_2$

1	nonsense$\$_2$
2	tense$\$_2$

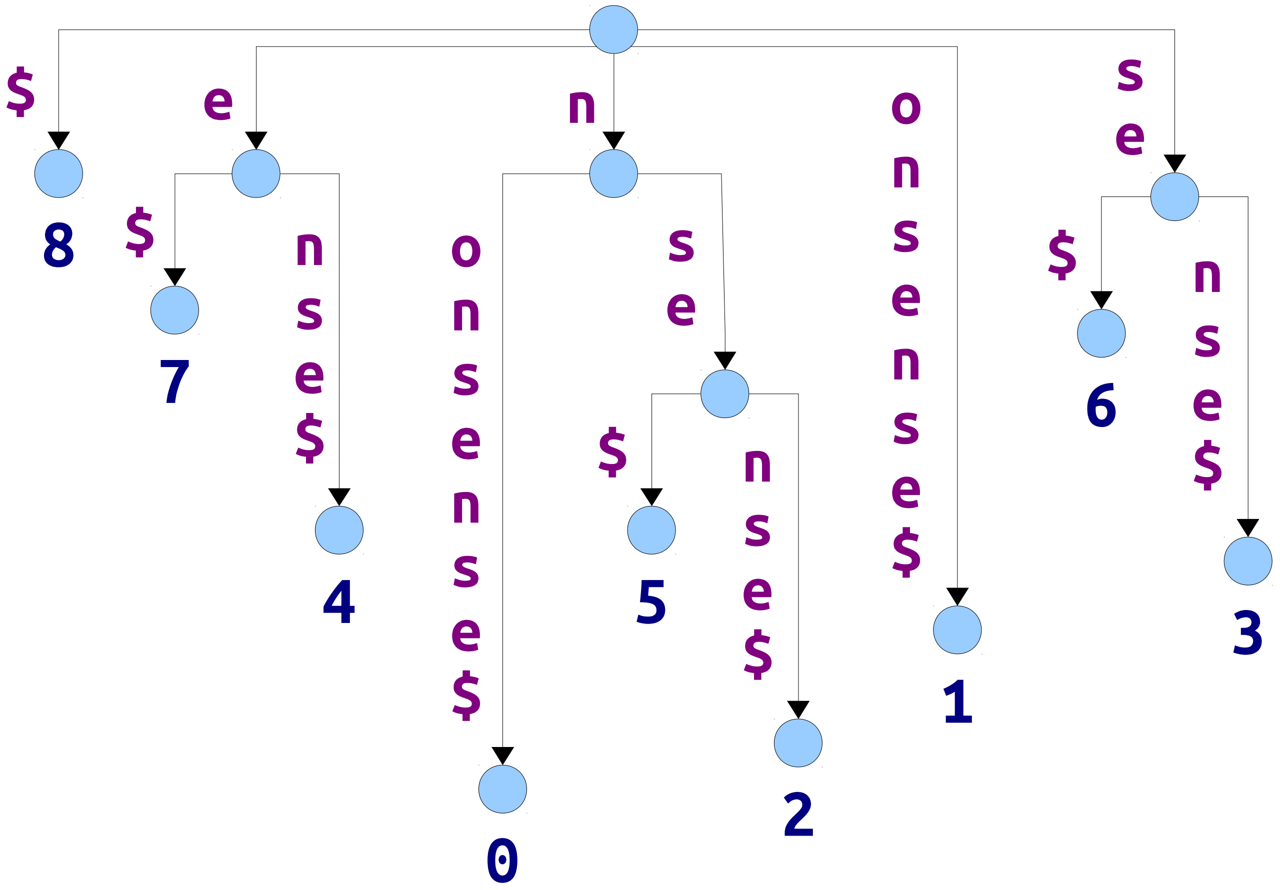
Using LCP: Constructing Suffix Trees

Constructing Suffix Trees

- Last time, I claimed it was possible to construct suffix trees in time O(m).
- We'll do this by showing the following:
  - A suffix array for T can be built in time O(m).
  - An LCP array for T can be built in time O(m).
    - Check Kasai's paper for details.
  - A suffix tree can be built from a suffix array and LCP array in time O(m).

From Suffix Arrays to Suffix Trees

Using LCP

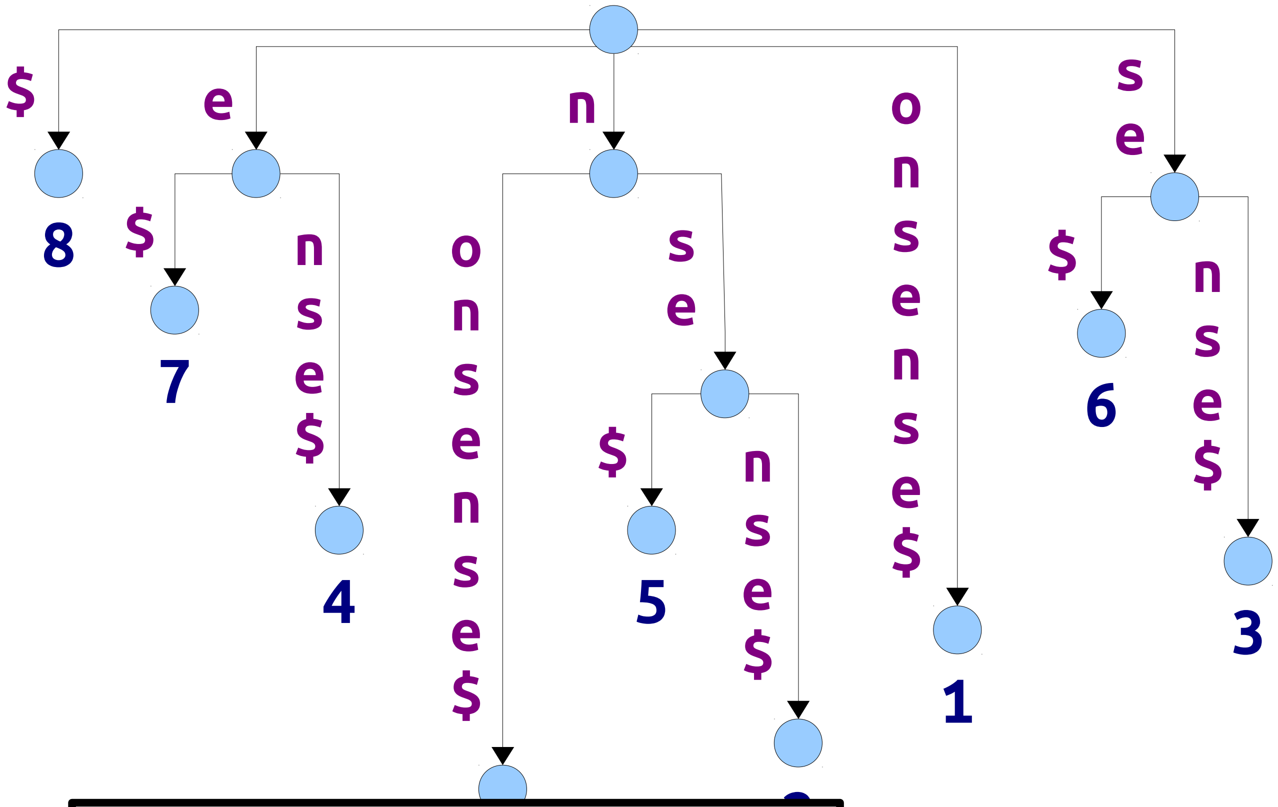


	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

nonsense$
012345678

Claim: Any 0's in the suffix array represent demarcation points between subtrees of the root node.

Using LCP

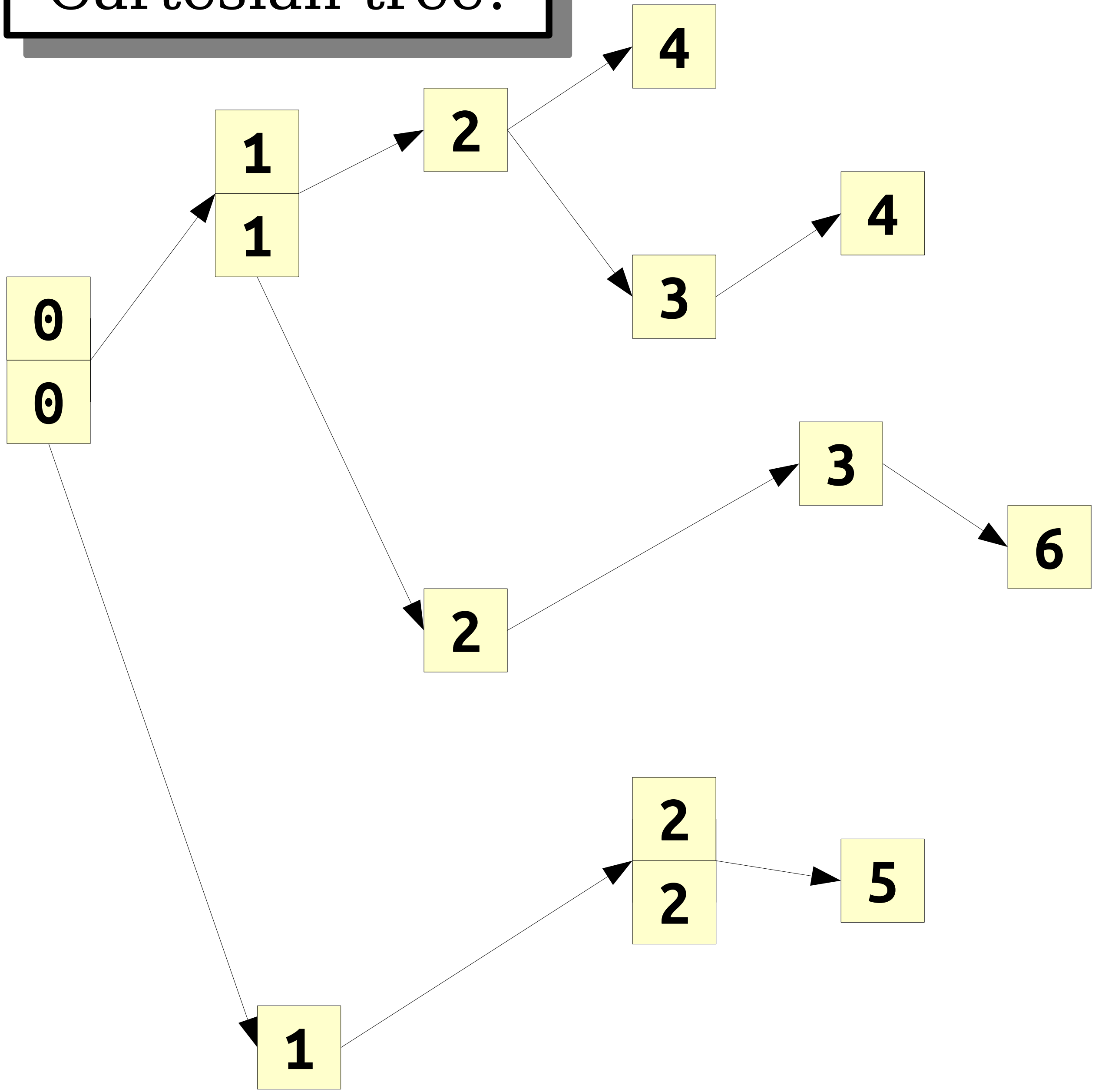


	8	$
1	7	e$
	4	ense$
1	0	nonsense$
3	5	nse$
	2	nsense$
	1	onsense$
2	6	se$
	3	sense$

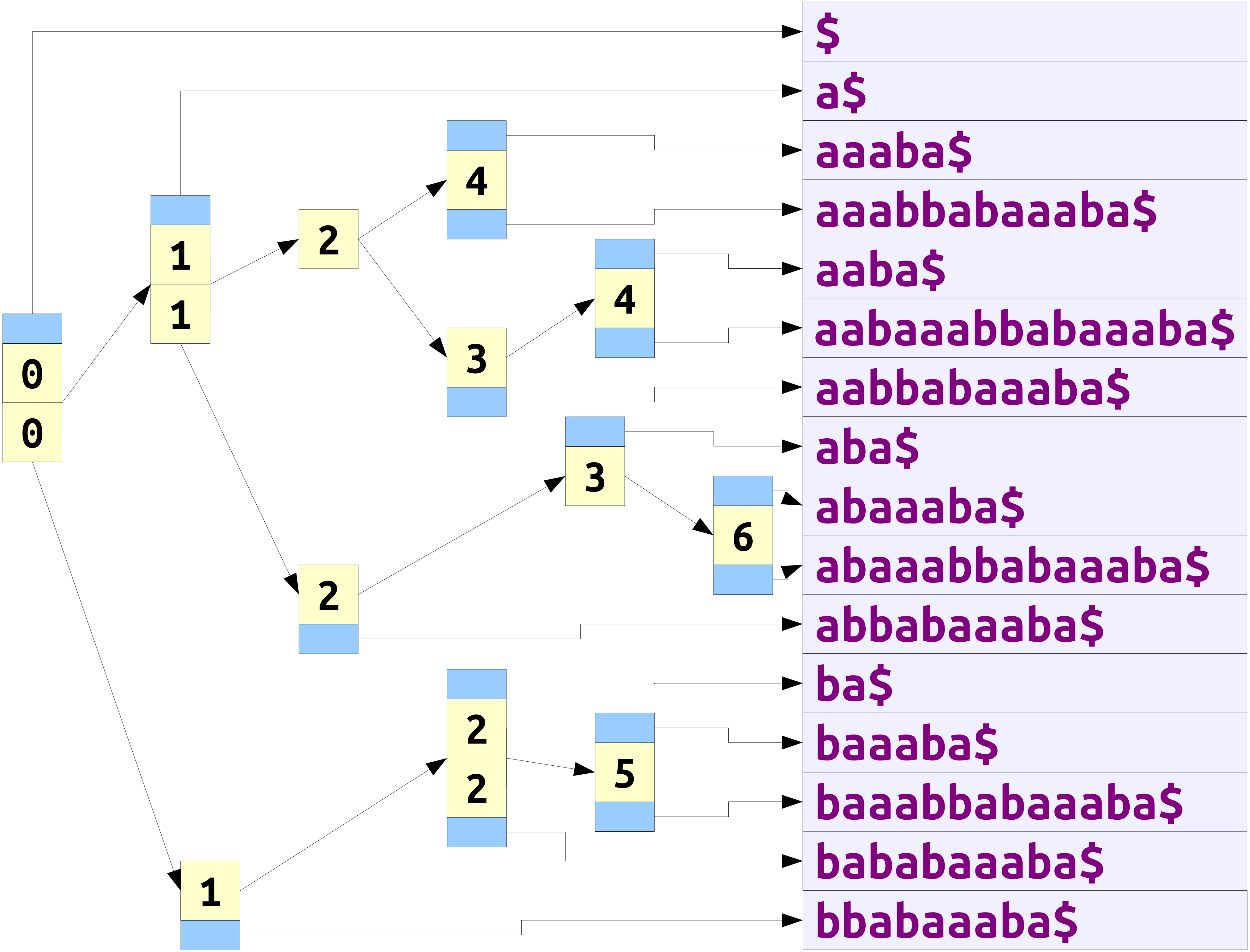
The same property holds for these subarrays, except using the subarray min instead of 0.

	$
0	a$
1	aaaba$
4	aaabbabaaaba$
2	aaba$
4	aabaaabbabaaaba$
3	aabbabaaaba$
1	aba$
3	abaaaba$
6	abaaabbabaaaba$
2	abbabaaaba$
0	ba$
2	baaaba$
5	baaabbabaaaba$
2	bababaaaba$
1	bbabaaaba$

This is a slightly modified Cartesian tree!



$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$


0
0
1
1
2
4
3
4
2
3
6
1
2
2
5
$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$

A Linear-Time Algorithm

- Construct a Cartesian tree from the LCP array, fusing together nodes with the same values if one becomes a parent of the other.
- Run a DFS over the tree and add missing children in the order in which they appear in the suffix array.
- Assign labels to the edges based on the LCP values.
- Total time: **O(m)**.

Time-Out For Announcements!

Problem Set Two

- Problem Set Two goes out today. It's due next Tuesday (April 19th) at the start of class.
    - Play around with tries, Aho-Corasick, suffix trees, and suffix arrays!
- Problem Set One has been graded. Grades are available on GradeScope.
- Solutions are available in hardcopy in lecture. They'll be in the filing cabinets in the Gates B wing (near Keith's office) if you weren't able to pick them up.
- Luna made some *excellent* graphs showing the actual performance of the RMQ data structures in practice, including charts for how common errors break the runtime bounds. Highly recommended!

Office Hours Location

- Looks like we're no longer allowed to hold office hours in the Huang Basement.
- We've moved our Monday / Tuesday office hours into Gates B26.
- Keith's office hours will still be in Gates 178.

WiCS Casual CS Dinner

- Stanford WiCS is holding the first of their biquarterly CS Casual Dinners next **Monday, April 18** from **6:30PM – 7:30PM** at the WCC.
- Highly recommended! Your perspective at this point in your CS career would be really valuable to people who are just starting out.

{w} Stanford Women
in Computer Science

WICS PRESENTS

CodeGirl Screening & Panel

Tuesday, April 12th
6:30-8:45pm @ 420-040
Popcorn provided!
RSVP at goo.gl/forms/kw6ad4f0KN

Come watch a thrilling, heartfelt documentary that follows high school-aged girls from around the world as they compete in the Technovation Challenge and work to better their communities through technology and collaboration.

The screening will be followed by a panel on closing the gender gap in the tech industry, featuring girls from the film and representatives from Technovation.

Check out the trailer at **www.codegirlmovie.com**

WICS.STANFORD.EDU **FACEBOOK.COM/STANFORDWICS** **WICS-BOARD@LISTS.STANFORD.EDU**

HackOverflow

- HackOverflow is this Saturday, April 16, from 10:00AM – 10:00PM in the Huang Basement.
- It's a great hackathon for first-timers. Highly recommended!

DiversityBase: Interested?

- DiversityBase is a joint effort by SOLE, SBSE, AISES, and FLIP with a focus on computer science.
- They're looking for people to take on leadership positions. This is a phenomenal organization and it would be a great place to make a huge impact.
- Interested? Apply here:

http://goo.gl/forms/50ObFGs5KS

LOFT Coder Summit

Presented By:

Infosys
FOUNDATION USA

Saturday, May 14th, 2016

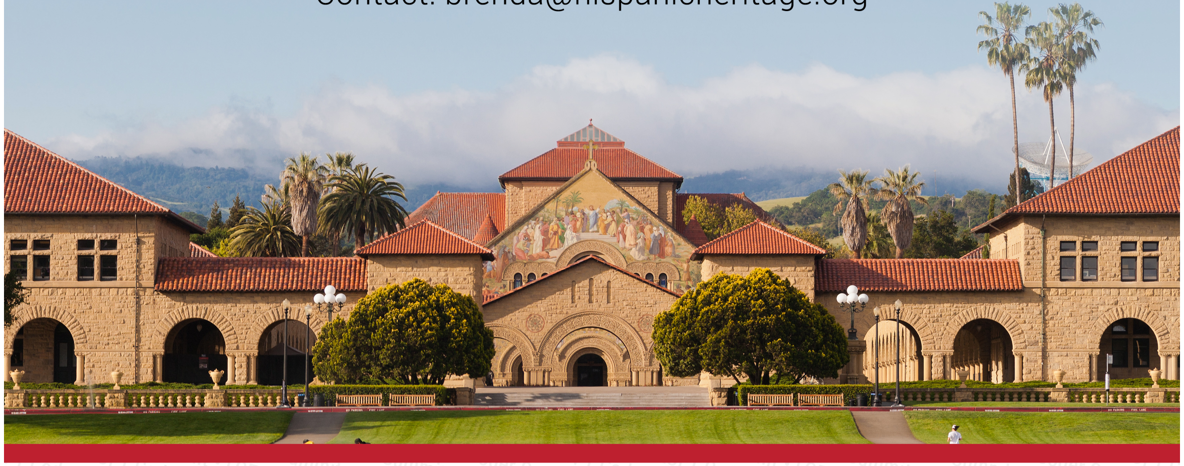
Stanford University
Stanford, CA

The summit is a free one-day event of:
Workshops
Internship Opportunities
Networking

Please join us in redefining the landscape of computer technology.

RSVP Here: lcsrsvp.com

www.loftcsl.org
Contact: brenda@hispanicheritage.org



Hispanic Heritage FOUNDATION

LOFT

CSL
Code as a Second Language

SOLE
Society of Latino Engineers

Back to CS166!

The Hard Part: Building Suffix Arrays

A Naïve Algorithm

- Here's a simple algorithm for building a suffix array:
  - Construct all the suffixes of the string in time $\Theta(m^2)$.
  - Sort those suffixes using heapsort or mergesort.
    - Makes $O(m \log m)$ comparisons, but each comparison takes $O(m)$ time.
    - Time required: $O(m^2 \log m)$.
- Total time: $\boldsymbol{O(m^2 \log m)}$.
- ***Can we do better?***

Radix Sort

- ***Radix sort*** is a fast sorting algorithm for strings and integers.
- It's a powerful primitive for building other algorithms and data structures – and comes up *all the time* in job interviews.
- In case you haven't seen it before (it's only intermittently taught in CS161), let's start with a quick radix sort review.

Analyzing Radix Sort

- Suppose there are t total strings with maximum length k, drawn from alphabet Σ.
- Time to set up initial buckets: $\Theta(|\Sigma|)$.
- Time to distribute strings elements each round: $O(t)$.
- Time to collect strings each round: $O(t + |\Sigma|)$.
- Number of rounds: $O(k)$
- Runtime: $\mathbf{O(\boldsymbol{k}(\boldsymbol{t} + |\Sigma|))}$.

Speeding Up with Radix Sort

- What happens if we use radix sort instead of heapsort in our original suffix array algorithm?
  - Number of strings: $\Theta(m)$.
  - String length: $\Theta(m)$.
  - Number of characters: $|\Sigma|$.
- Runtime is therefore $\boldsymbol{\Theta(m^2 + m|\Sigma|)}$
- Assuming $|\Sigma| = O(m)$, the runtime is $\Theta(m^2)$, a log factor faster than before.
- ***Can we do better?***

Radix Sort

- ***Useful observation:*** it's possible to sort t strings in time O(t) if
  - the strings all have a constant length, and
  - the alphabet size is O(t).
- We're going to use this observation in a little bit, but make a note of it for now.

The DC3 Algorithm

DC3

- One of the simplest and fastest algorithms for building suffix arrays is called DC3 (***D***ifference ***C***over, size ***3***).
- It's a masterpiece of an algorithm – it's clever, brilliant, and not that hard to code up.
- It's also quite nuanced and tricky.
- We're going to spend the rest of today working through the details. You'll then play around with it on the problem set.

Some Assumptions

- Assume the initial input alphabet consists of a set of integers 0, 1, 2, …, |Σ| - 1.
- If this isn't the case, we can always sort the letters and replace each with its rank.
- Assuming that |Σ| = O(1), this doesn't affect the runtime.

Some Terminology

- Define T_k to be the positions in T whose indices are equal to k mod 3.
  - T_0 is the set of positions that are multiples of three.
  - T_1 is the set of positions that follow the positions in T_0.
  - T_2 is the set of positions that follow the positions in T_1.

m	o	n	s	o	o	n	n	o	m	n	o	m	s	$

DC3, Intuitively

- At a high-level, DC3 works as follows:
  - Recursively get the sorted order of all suffixes in T_1 and T_2.
  - Using this information, efficiently sort the suffixes in T_0.
  - Merge the two lists of sorted suffixes (the suffixes in T_0 and the suffixes in T_1/T_2) together to form the full suffix array.
- The details are beautiful, but tricky.

DC3, Intuitively

At a high-level, DC3 works as follows:

- Recursively get the sorted order of all suffixes in T_1 and T_2.

Using this information, efficiently sort the suffixes in T_0.

Merge the two lists of sorted suffixes (the suffixes in T_0 and the suffixes in T_1/T_2) together to form the full suffix array.

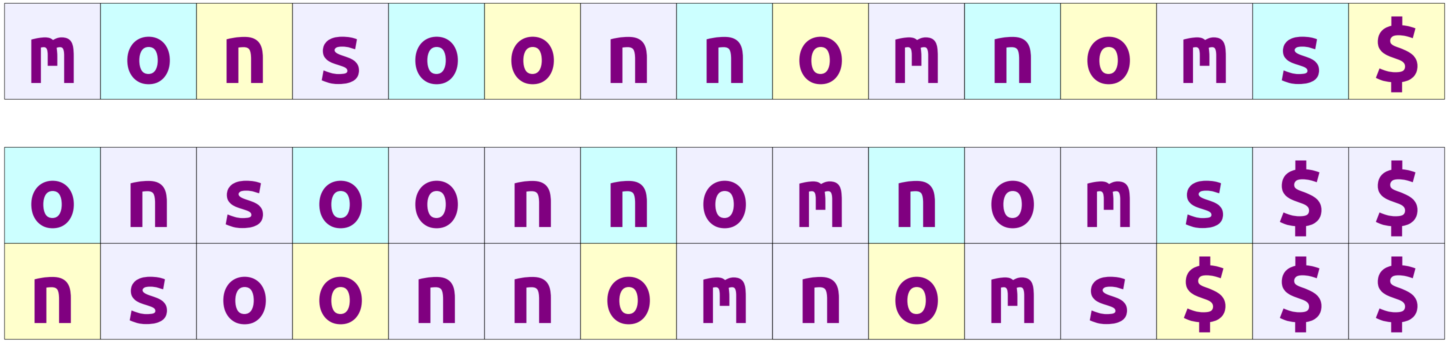
The details are beautiful, but tricky.

The First Step

- Our objective is to get the relative rankings of the suffixes at positions T_1 and T_2.
- High-level idea:
  - Construct a new string based on suffixes starting at positions in T_1 and T_2.
  - Compute the suffix array of that string, recursively.
  - Use the resulting suffix array to deduce the orderings of the suffixes from T_1 and T_2.

Embiggening Our String

- Form two new strings from *T*$ by dropping off the first character and first two characters and padding with extra $ markers.
- Then, concatenate those strings together.

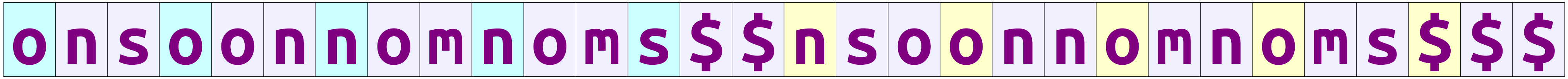


m	o	n	s	o	o	n	n	o	m	n	o	m	s	$

o	n	s	o	o	n	n	o	m	n	o	m	s	$	$
n	s	o	o	n	n	o	m	n	o	m	s	$	$	$

Embiggening Our String

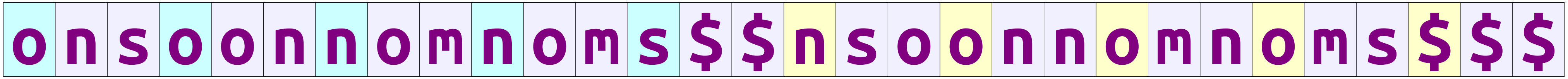
- Form two new strings from *T*\$ by dropping off the first character and first two characters and padding with extra \$ markers.
- Then, concatenate those strings together.



o	n	s	o	o	n	n	o	m	n	o	m	s	$	$	n	s	o	o	n	n	o	m	n	o	m	s	$	$	$

Um, Why?

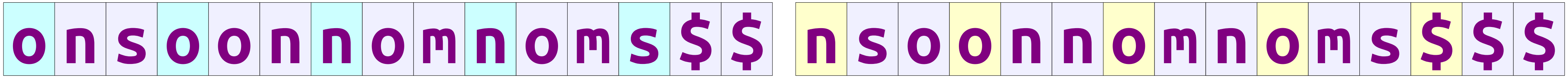
- ***Claim:*** The relative order of the suffixes in the first half of the string starting at positions in T_1 and the suffixes in the second half of the string at positions in T_2 is the same as the relative order of those suffixes in T.
- ***Intuition:*** Strings within the same half are relatively ordered. Strings across the two halves are "protected" by the endmarkers.



o	n	s	o	o	n	n	o	m	n	o	m	s	$	$	n	s	o	o	n	n	o	m	n	o	m	s	$	$	$

So, Um...

... we just doubled the size of our input string. You're not supposed to do that in a divide-and-conquer algorithm.

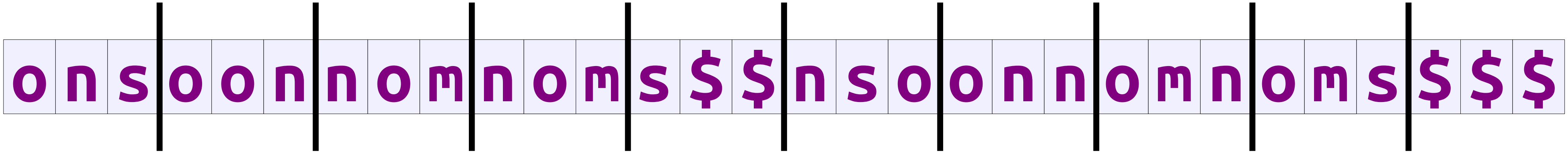


o	n	s	o	o	n	n	o	m	n	o	m	s	$	$

n	s	o	o	n	n	o	m	n	o	m	s	$	$	$

Playing with Blocks

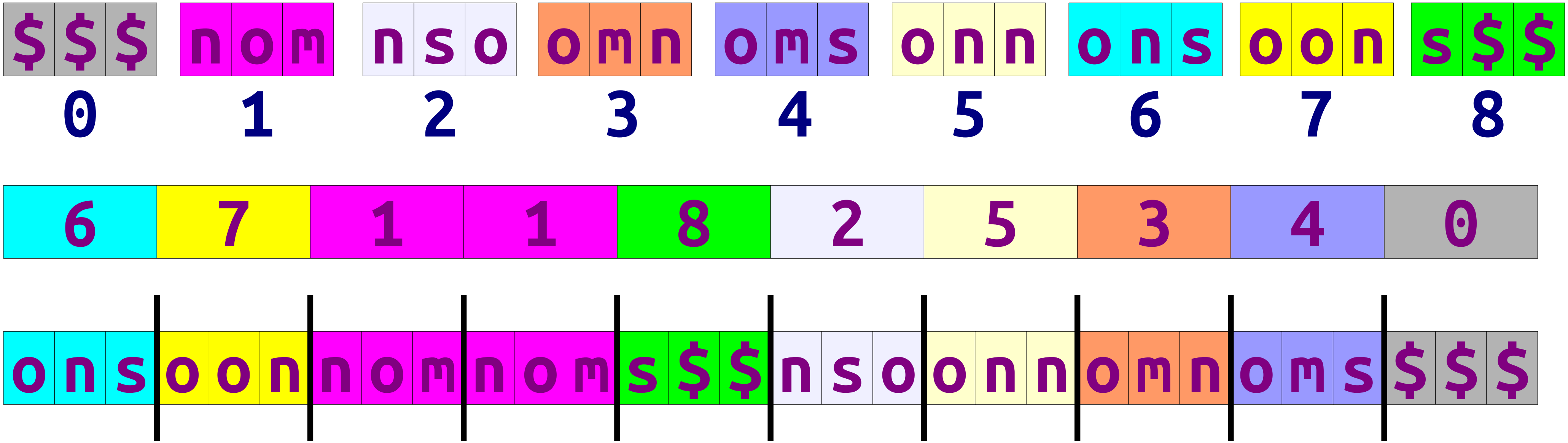
- ***Key Insight:*** Break the resulting string apart into blocks of size three.
- Think about what happens if we compare two suffixes starting at the beginning of a block:
    - Since the suffixes are distinct, there's a mismatch at some point.
    - All blocks prior to that point must be the same.
    - The differing block of three is the tiebreaker.



o	n	s	o	o	n	n	o	m	n	o	m	s	$	$	n	s	o	o	n	n	o	m	n	o	m	s	$	$	$

The Recursive Step

- ***The Trick:*** Treat each block of three characters as its own character.
- Determine the relative ordering of those characters by an O(m)-time radix sort.
- Replace each block of three characters with the rank of its “metacharacter.”
- Recursively compute the suffix array of the resulting string.

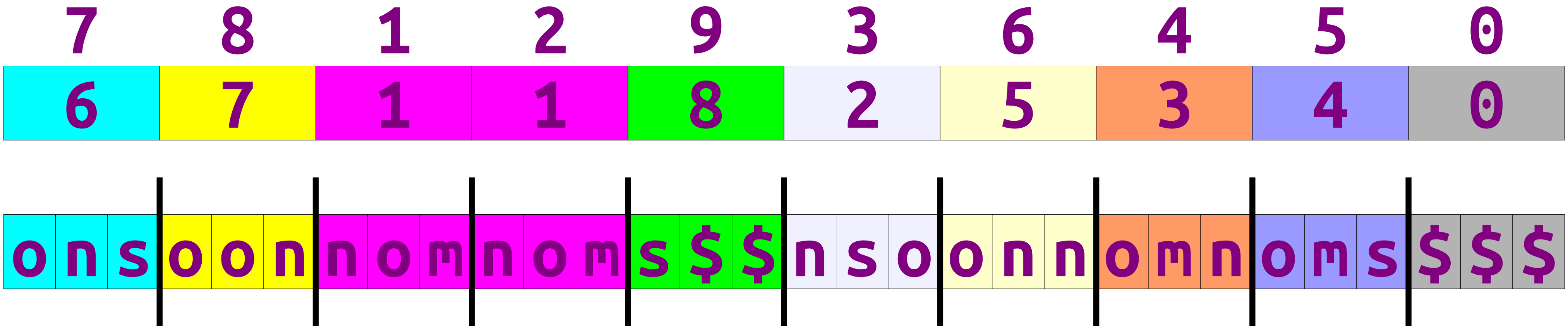


$$$	nom	nso	omn	oms	onn	ons	oon	s$$
0	1	2	3	4	5	6	7	8

6	7	1	1	8	2	5	3	4	0
ons	oon	nom	nom	s$$	nso	onn	omn	oms	$$$

The Recursive Step

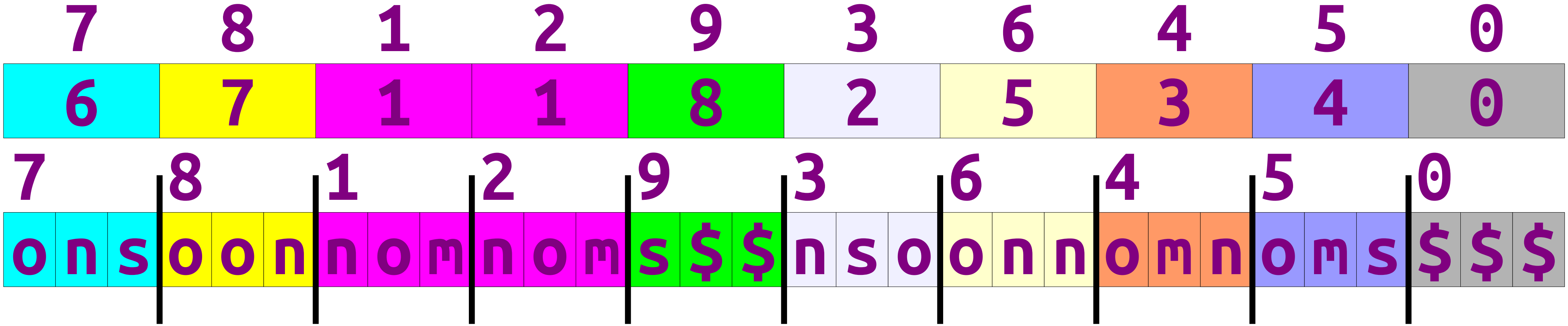
- ***The Trick:*** Treat each block of three characters as its own character.
- Determine the relative ordering of those characters by an O(m)-time radix sort.
- Replace each block of three characters with the rank of its "metacharacter."
- Recursively compute the suffix array of the resulting string.



7	8	1	2	9	3	6	4	5	0
6	7	1	1	8	2	5	3	4	0
ons	oon	nom	nom	s$$	nso	onn	omn	oms	$$$

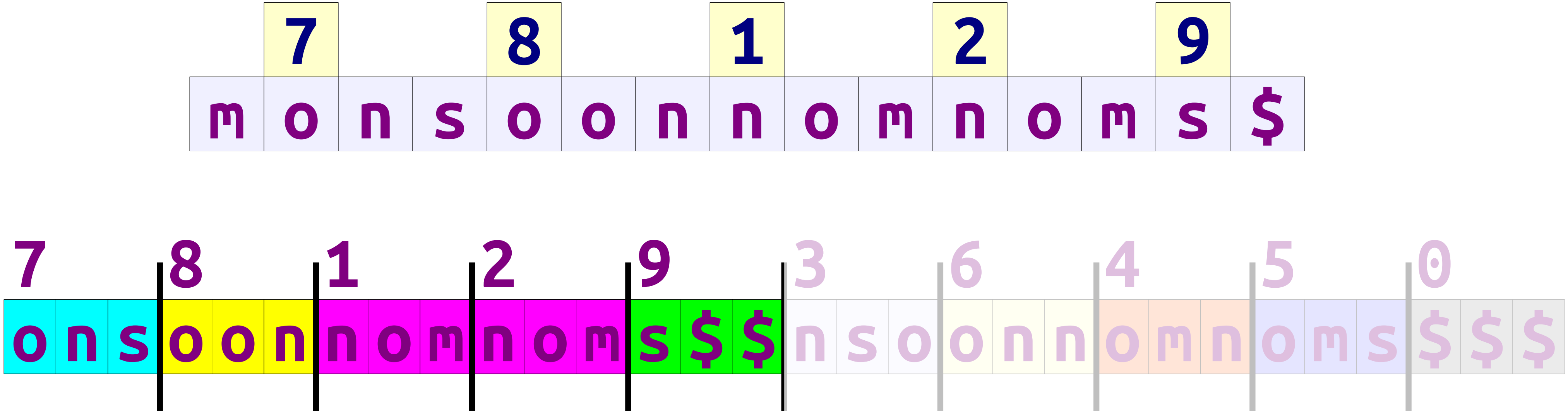
The Recursive Step

- Once we have this suffix array, we can use it to get the suffixes from T_1 and T_2 into sorted order.



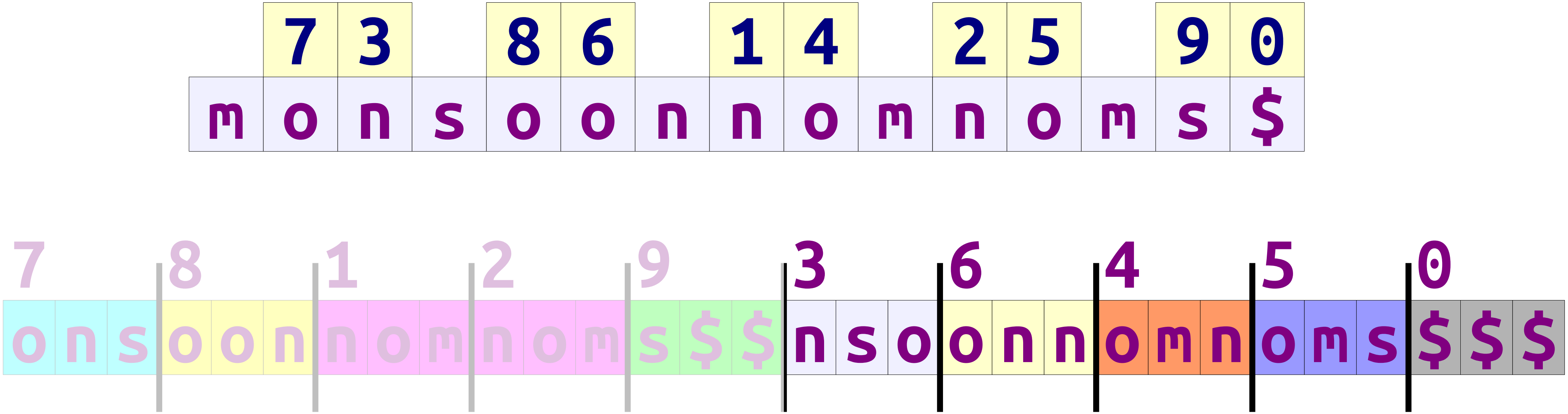
The Recursive Step

- Once we have this suffix array, we can use it to get the suffixes from T_1 and T_2 into sorted order.



The Recursive Step

- Once we have this suffix array, we can use it to get the suffixes from T_1 and T_2 into sorted order.



Ranking T_1 and T_2

- We spend a total of O(m) work in this step doubling the array, grouping it into blocks of size 3, radix sorting it, and converting the result of the call into meaningful data.
- We also make a recursive call on an array of size $2m$ / 3.
- Total work: O(m), plus a recursive call on an array of size $2m$ / 3.

DC3, Intuitively

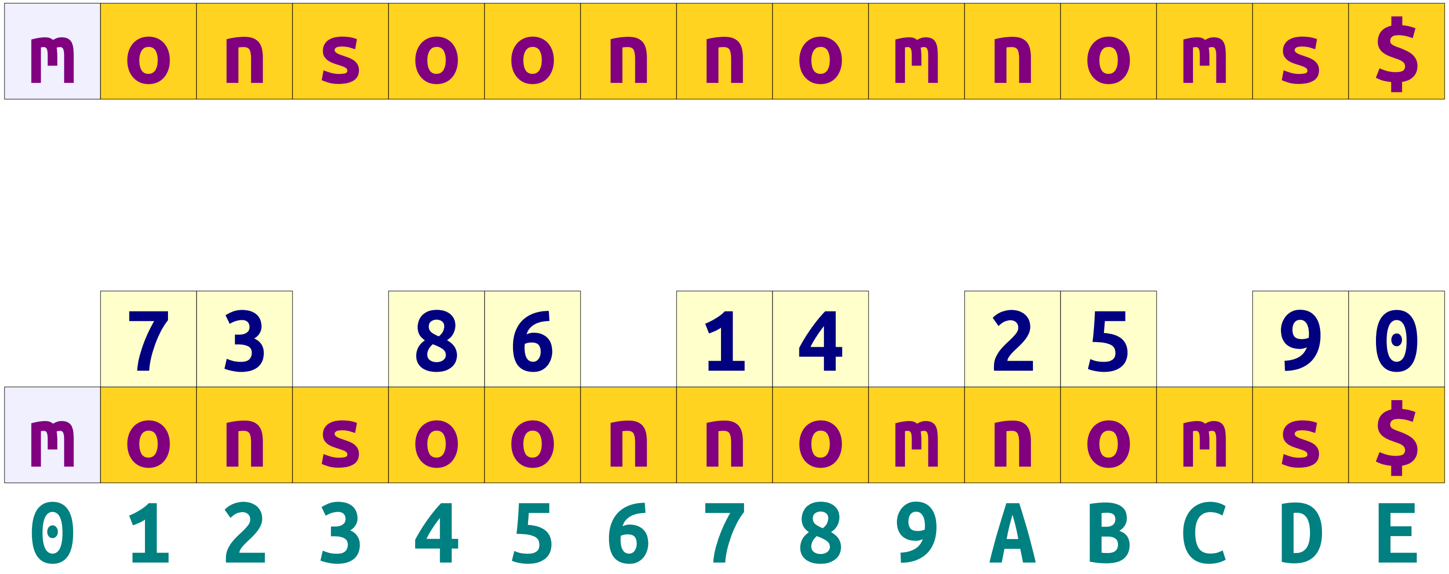
At a high-level, DC3 works as follows:

- Recursively get the sorted order of all suffixes in T_1 and T_2.
- Using this information, efficiently sort the suffixes in T_0.
- Merge the two lists of sorted suffixes (the suffixes in T_0 and the suffixes in T_1/T_2) together to form the full suffix array.

The details are beautiful, but tricky.

A Beautiful Insight

- ***Claim:*** If we know the relative ordering of suffixes at positions T_1 and T_2, we can determine the relative order of suffixes in positions T_0.
- ***Idea:*** Use a modified radix sort!

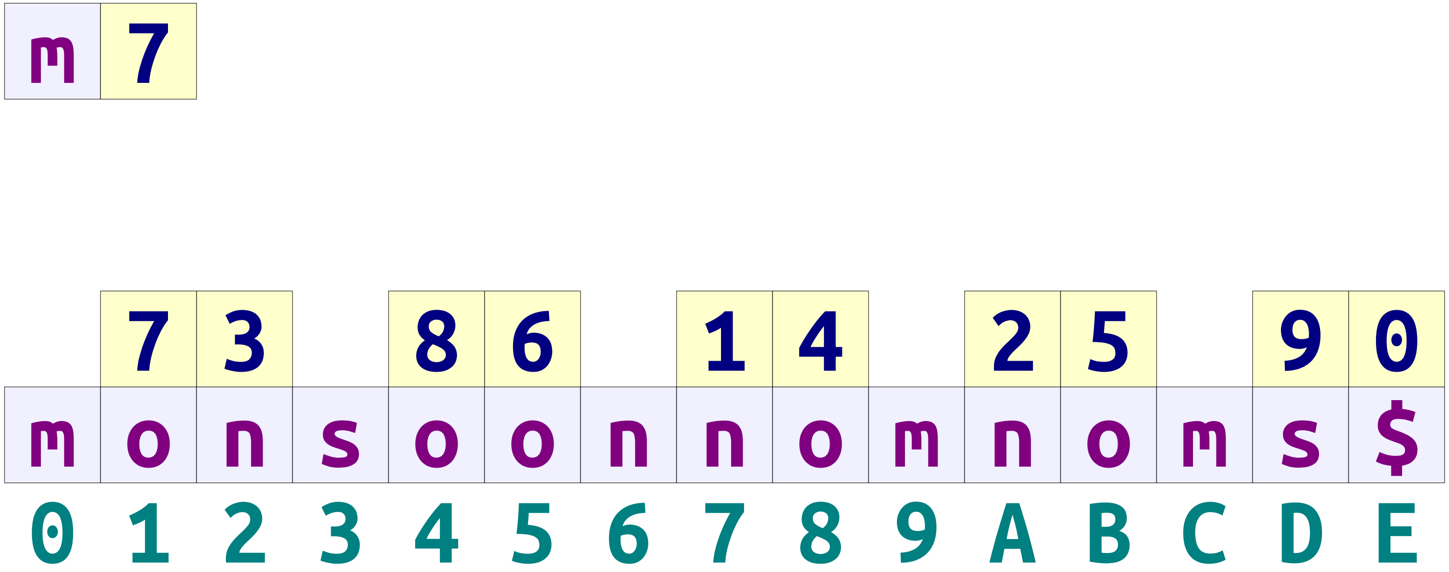


m	o	n	s	o	o	n	n	o	m	n	o	m	s	$

	7	3		8	6		1	4		2	5		9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

A Beautiful Insight

- ***Claim:*** If we know the relative ordering of suffixes at positions T_1 and T_2, we can determine the relative order of suffixes in positions T_0.
- ***Idea:*** Use a modified radix sort!



m	7

	7	3		8	6		1	4		2	5		9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

A Beautiful Insight

- ***Claim:*** If we know the relative ordering of suffixes at positions T_1 and T_2, we can determine the relative order of suffixes in positions T_0.
- ***Idea:*** Use a modified radix sort!

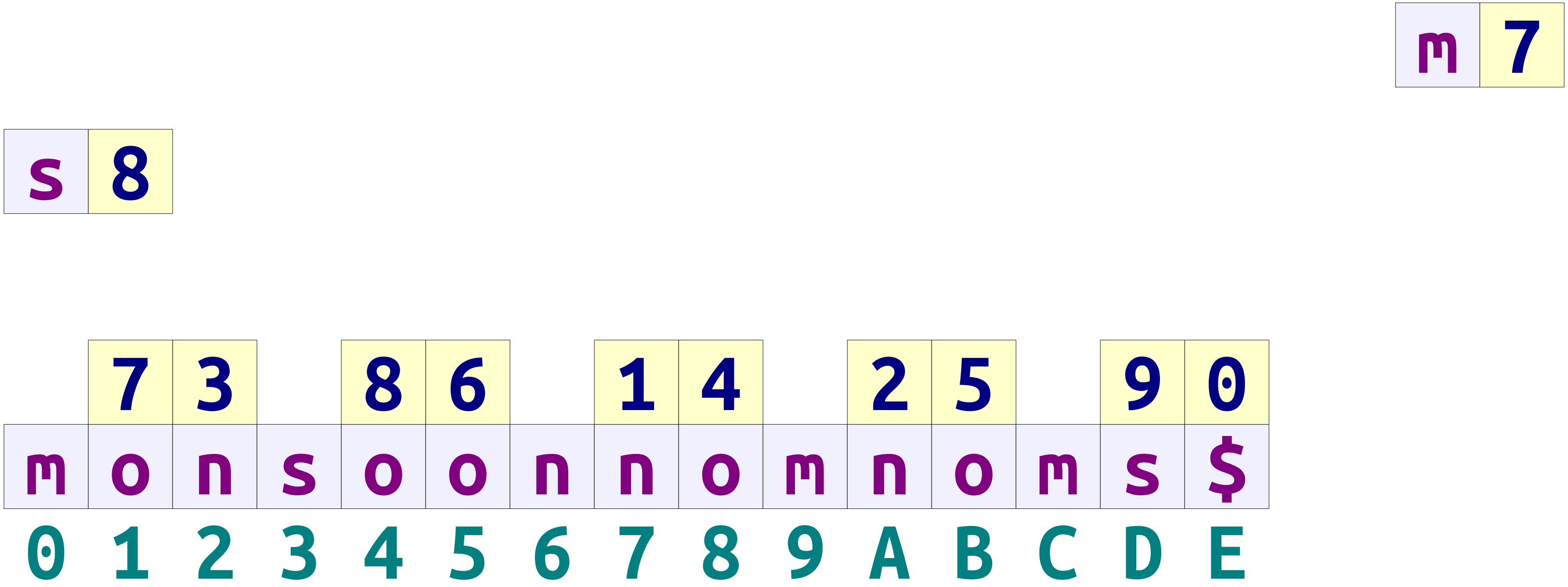
m	7

s	o	o	n	n	o	m	n	o	m	s	$

	7	3		8	6		1	4		2	5		9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

A Beautiful Insight

- ***Claim:*** If we know the relative ordering of suffixes at positions T_1 and T_2, we can determine the relative order of suffixes in positions T_0.
- ***Idea:*** Use a modified radix sort!



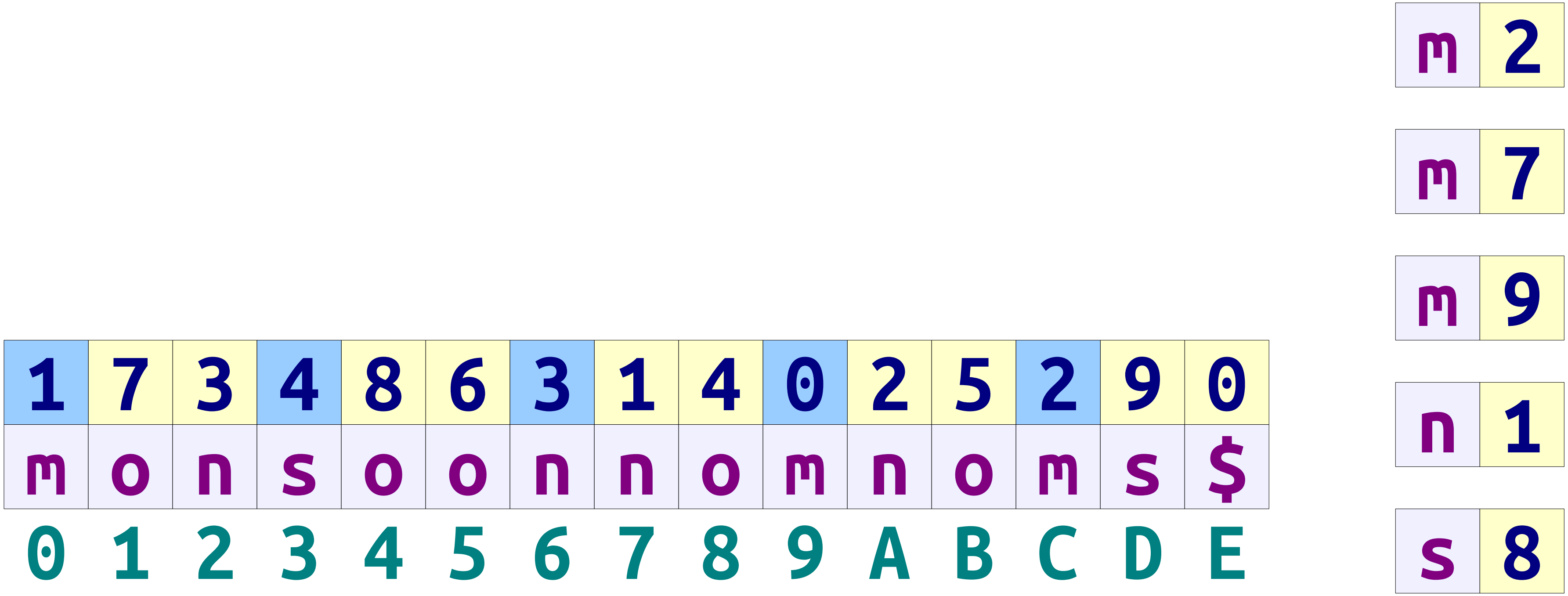
m 7

s 8

	7	3		8	6		1	4		2	5		9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

A Beautiful Insight

- ***Claim:*** If we know the relative ordering of suffixes at positions T_1 and T_2, we can determine the relative order of suffixes in positions T_0.
- ***Idea:*** Use a modified radix sort!



1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

m	2
m	7
m	9
n	1
s	8

Sorting T_0

- To sort T_0, we do the following:
    - For each position in T_0, form a pair of the letter at that position and the index of the suffix right after it (which is in T_1).
    - These pairs are effectively strings drawn from an alphabet of size Σ + m.
    - Radix sort them in time O(m).

DC3, Intuitively

At a high-level, DC3 works as follows:

Recursively get the sorted order of all suffixes in T_1 and T_2.

Using this information, efficiently sort the suffixes in T_0.

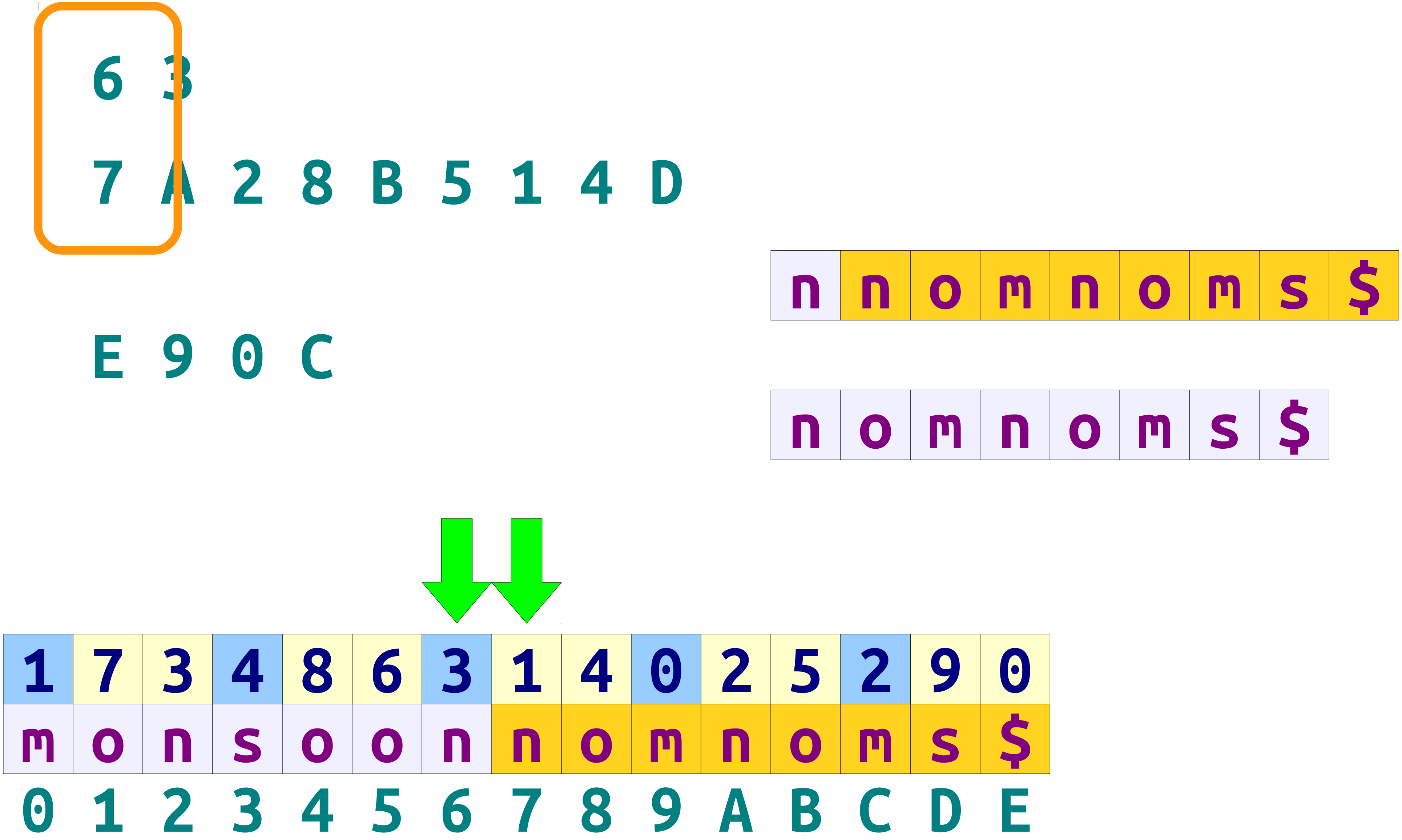
- Merge the two lists of sorted suffixes (the suffixes in T_0 and the suffixes in T_1/T_2) together to form the full suffix array.

The details are beautiful, but tricky.

Merging the Lists

- At this point, we have two sorted lists:
  - A sorted list of all the suffixes in T_0.
  - A sorted list of all the suffixes in T_1 and T_2.
- We also know the relative order of any two suffixes in T_1 and T_2.
- How can we merge these lists together?

The Merging



6 3

7 A 2 8 B 5 1 4 D

E 9 0 C

n	n	o	m	n	o	m	s	$

n	o	m	n	o	m	s	$

1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

Key idea: We know the relative ordering of the suffixes at positions that are congruent to 1 or 2 mod 3.

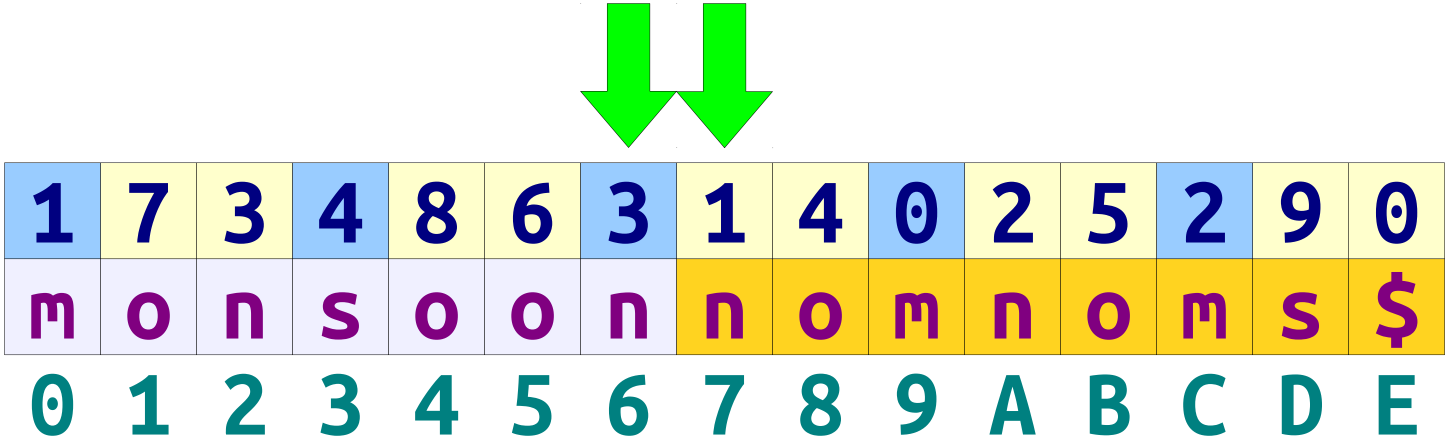
6 3

7 A 2 8 B 5 1 4 D

E 9 0 C

n	1

n	o	m	n	o	m	s	$



1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

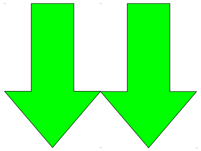
6 3

7 A 2 8 B 5 1 4 D

E 9 0 C

n	1

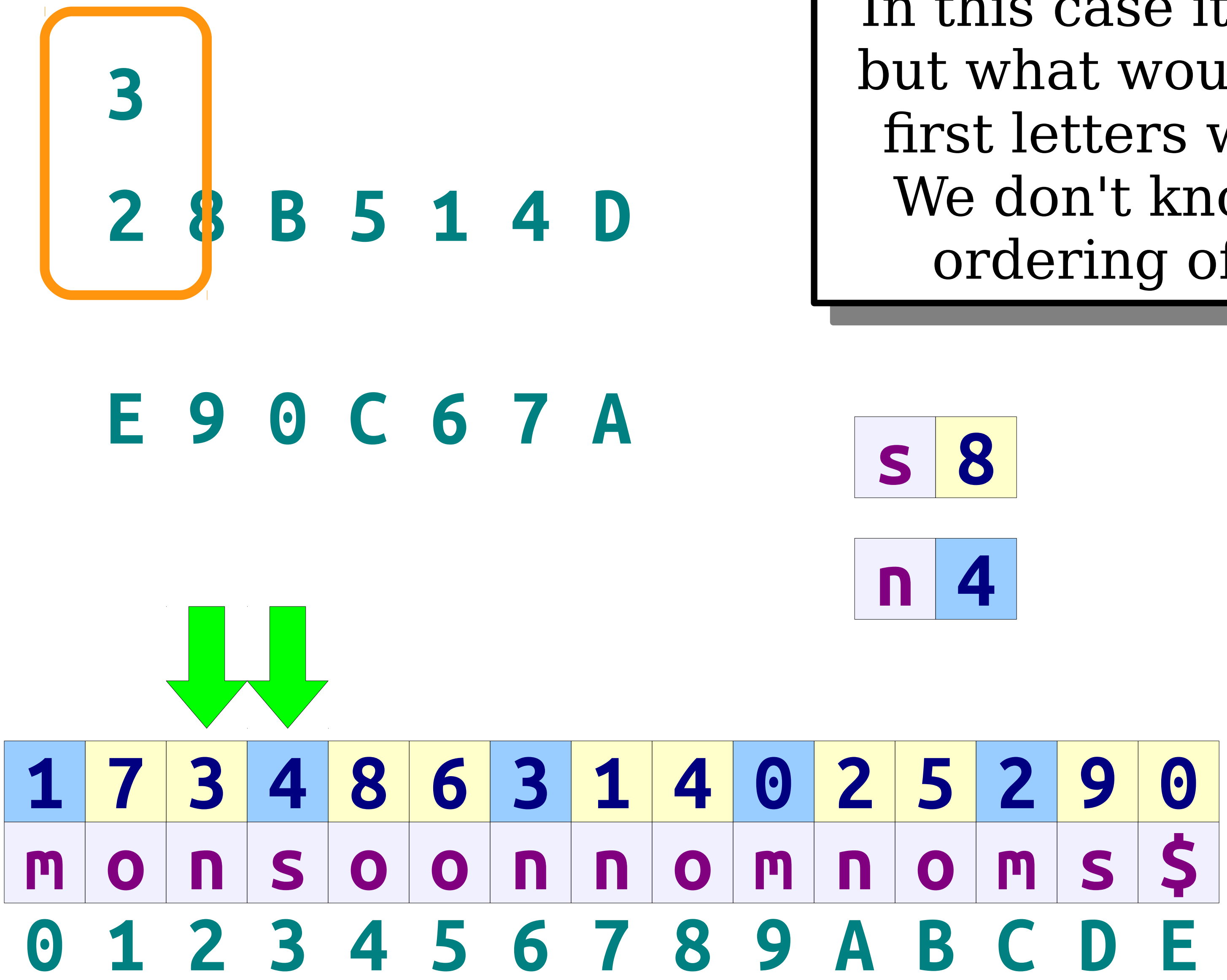
n	4



1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

In this case it doesn't matter, but what would happen if the first letters were the same? We don't know the relative ordering of the suffixes.



3

2 8 B 5 1 4 D

E 9 0 C 6 7 A

s	8
n	4

1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

These can be ranked regardless of whether the first two characters are the same.

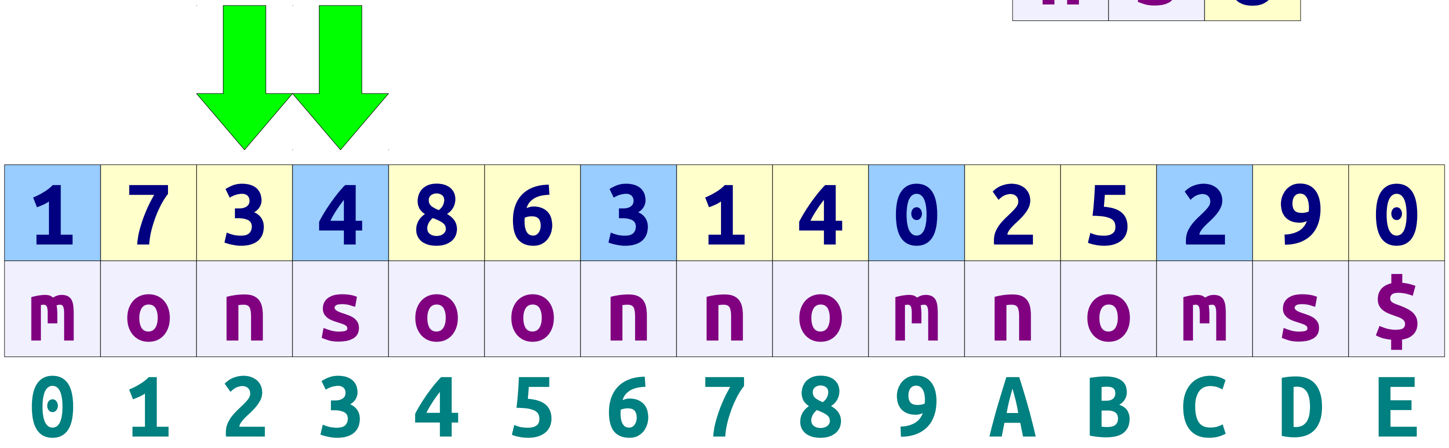
3

2 8 B 5 1 4 D

E 9 0 C 6 7 A

s	o	6

n	s	8



1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

E 9 0 C 6 7 A 2 8 B 5 1 4 D 3

2	B	7	E	C	A	4	5	8	1	6	9	3	D	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

- Comparing any two suffixes requires at most O(1) work because we can use the existing ranking of the suffixes in T_1 and T_2 to "truncate" long suffixes.
- There are a total of m suffixes to merge.
- Total runtime: **O($\boldsymbol{m}$)**.

The Overall Runtime

- The recursive step to sort T_1 and T_2 takes time $\Theta(m)$ plus the cost of a recursive call on an input of size $2m / 3$.
- Using T_1 and T_2 to sort T_0 takes time $\Theta(m)$.
- Merging T_0, T_1, and T_2 takes time $\Theta(m)$.
- Recurrence relation:

$$R(m) = R(2m / 3) + O(m)$$

- Via the Master Theorem, we see that the overall runtime is $\boldsymbol{\Theta(m)}$.

The Overall Algorithm

- Although this algorithm has a lot of tricky details, it's actually not that tough to code it up.
- The original paper gives a two-page C++ implementation of the entire algorithm.
- And because we're Decent Human Beings, we're not going to ask you to write it up on your own. ☺

Questions to Ponder

- This algorithm is extremely clever and has lots of interlocking moving parts.
  - Why is the number 3 so significant?
  - Why did we have to double the length of the string before grouping into blocks?
- You'll explore some of these questions in the problem set.

How Did Anyone Invent This?

- This algorithm can seem totally magical and confusing the first time you see it.
- As with most algorithms, this one was based on a lot of prior work.
- In 1997, Martin Farach published an algorithm (now called ***Farach's algorithm***) for directly building a suffix tree in time O(m). It involved many of the same techniques (just sort suffixes at some specific positions, use that to fill in the missing suffixes, then merge the results), but has a lot more details because it works directly on suffix trees rather than arrays.
- The algorithm itself is a bit tricky but is totally beautiful. It would make for a really fun final project!

More to Explore

- There are a number of other data structures in the family of suffix trees and suffix arrays.
  - The ***suffix automaton*** or ***DAWG*** is a minimal-state DFA for all the suffixes of a string *T*. It always has size O(|*T*|), and this is not obvious!
  - A ***factor oracle*** is a relaxed automaton that matches all the substrings of some string *T*, plus possibly some spurious matches.
  - The ***Burrows-Wheeler transform*** is a technique related to suffix arrays that was originally developed for data compression.
- Any of these would be make for great final project topics.

Summary

- Suffix trees are a compact, flexible, powerful structure for answering questions on strings.
- Suffix arrays give a space-efficient alternative to suffix trees that have a slight time tradeoff.
- LCP arrays link suffix trees and suffix arrays and can be built in time O(m).
- Suffix arrays can be constructed in time O(m).
- Suffix trees can be constructed in time O(m) from a suffix array and LCP array.

Next Time

- **Balanced Trees**
  - B-trees, 2-3-4 trees, and red/black trees.
- **Where the heck did red/black trees come from?**
  - There's an *amazing* answer to this question. Trust me.