

Suffix Arrays

Original slide from <https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/04/Slides04.pdf>

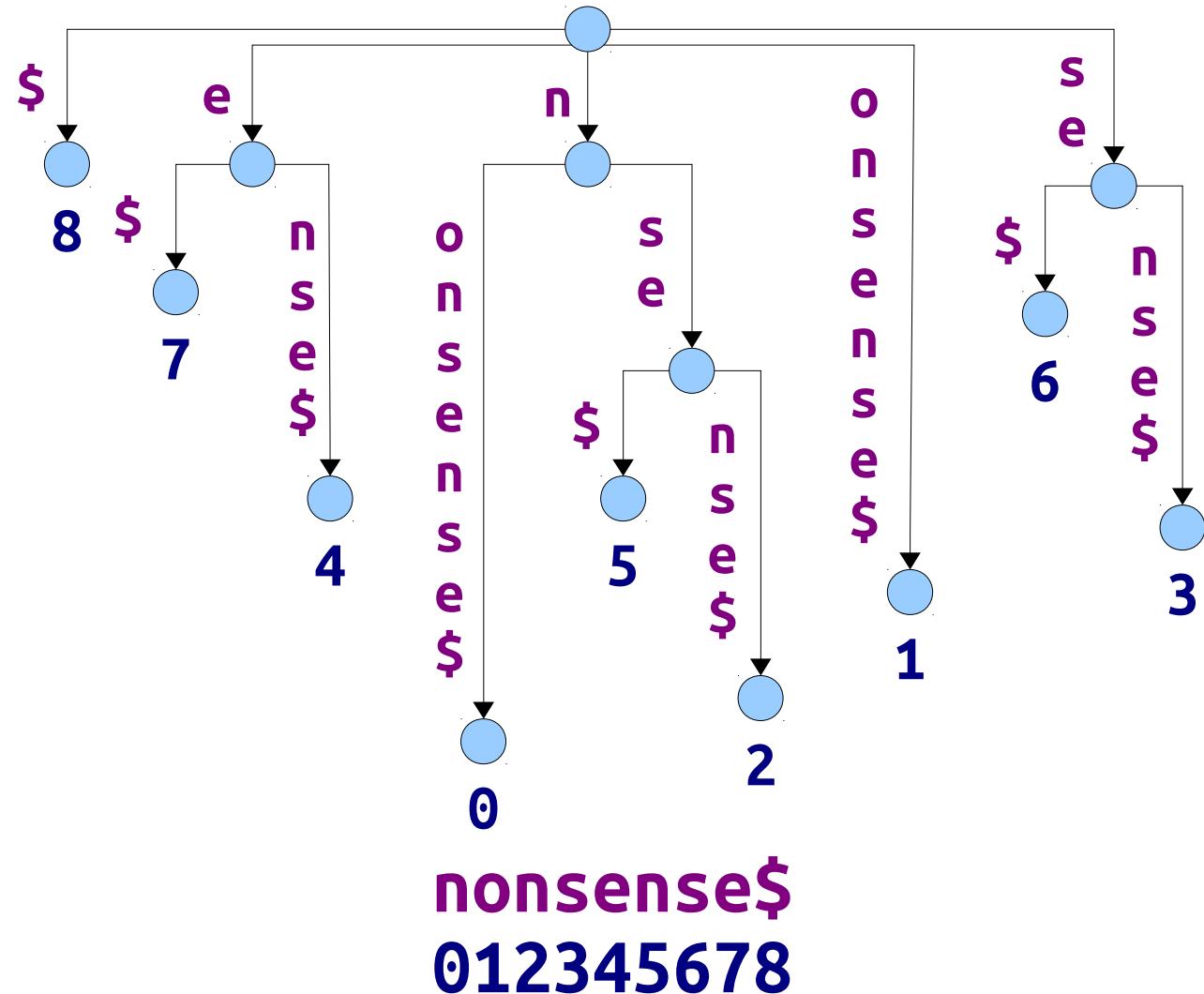
Outline for Today

- **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

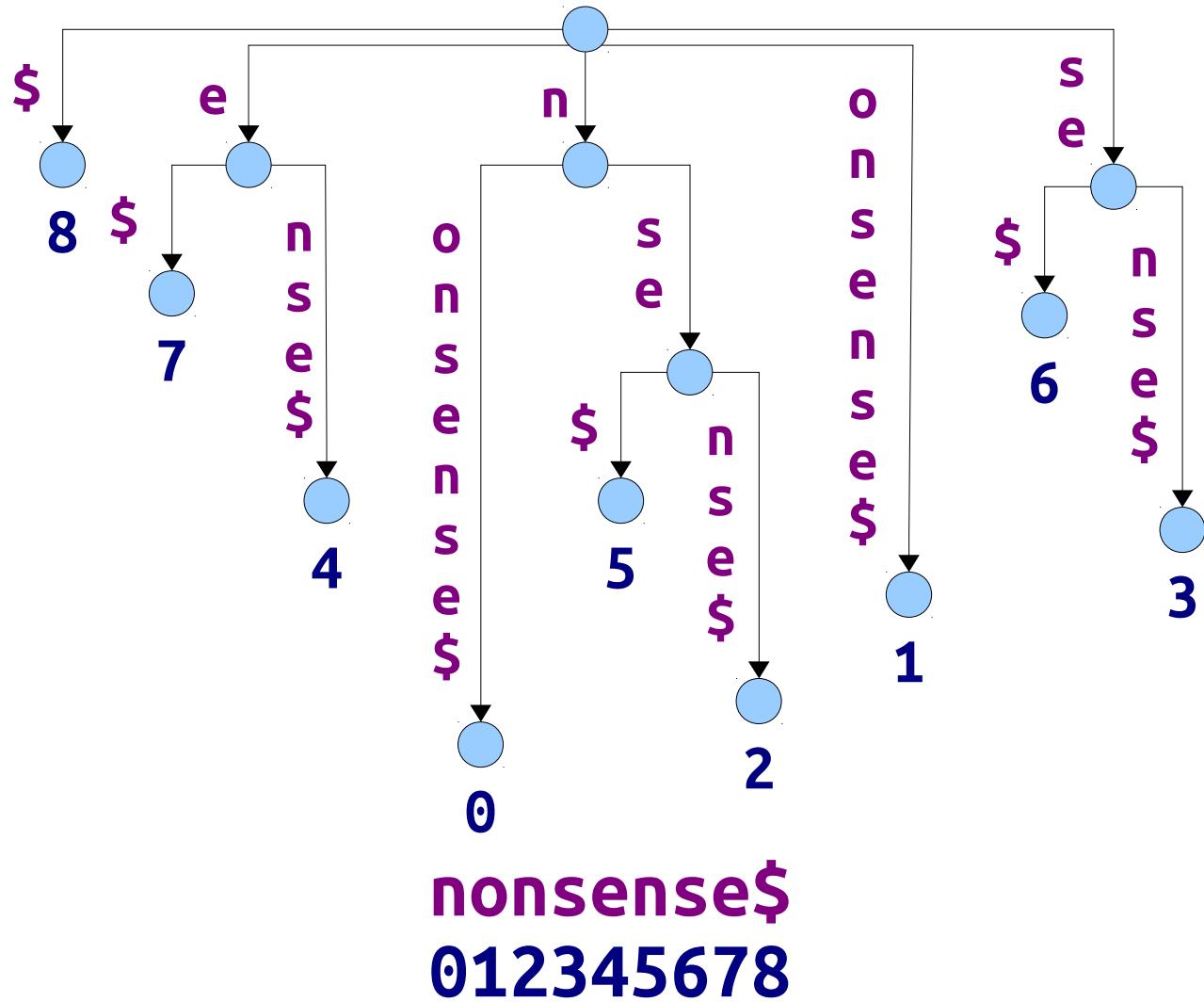
Suffix Trees

- 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 $\Sigma = \{A, C, G, T, \$\}$.
- 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.

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

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	ononsense $\$$
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	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

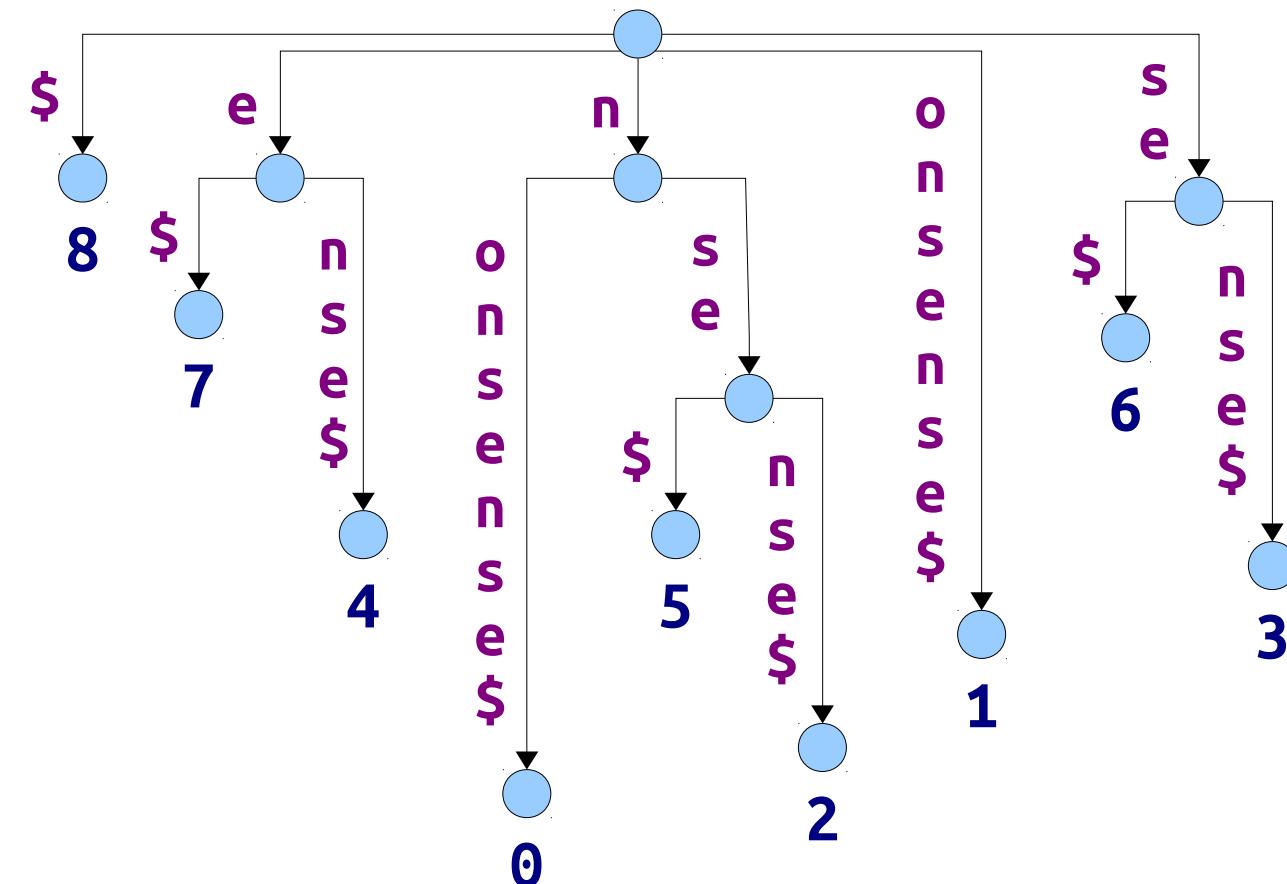
- The binary search will require $O(\log m)$ probes into the suffix array.
- Each comparison takes time $O(n)$: have to compare P against the current suffix.
- Time for binary searching: $O(n \log m)$.
- Time to report all matches after that point: $O(z)$.
- Total time: **$O(n \log m + z)$** .

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.

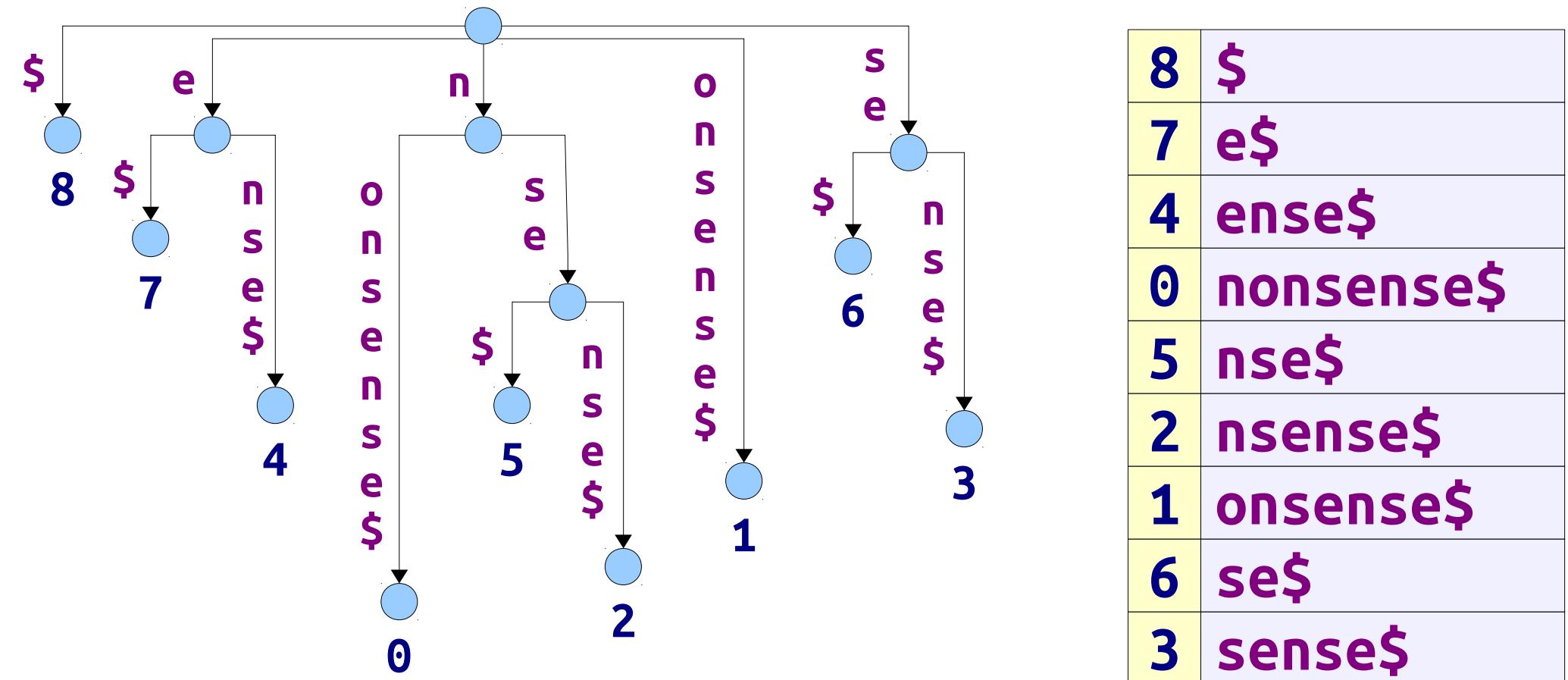
Suffix Trees and Suffix Arrays



**nonsense\$
012345678**

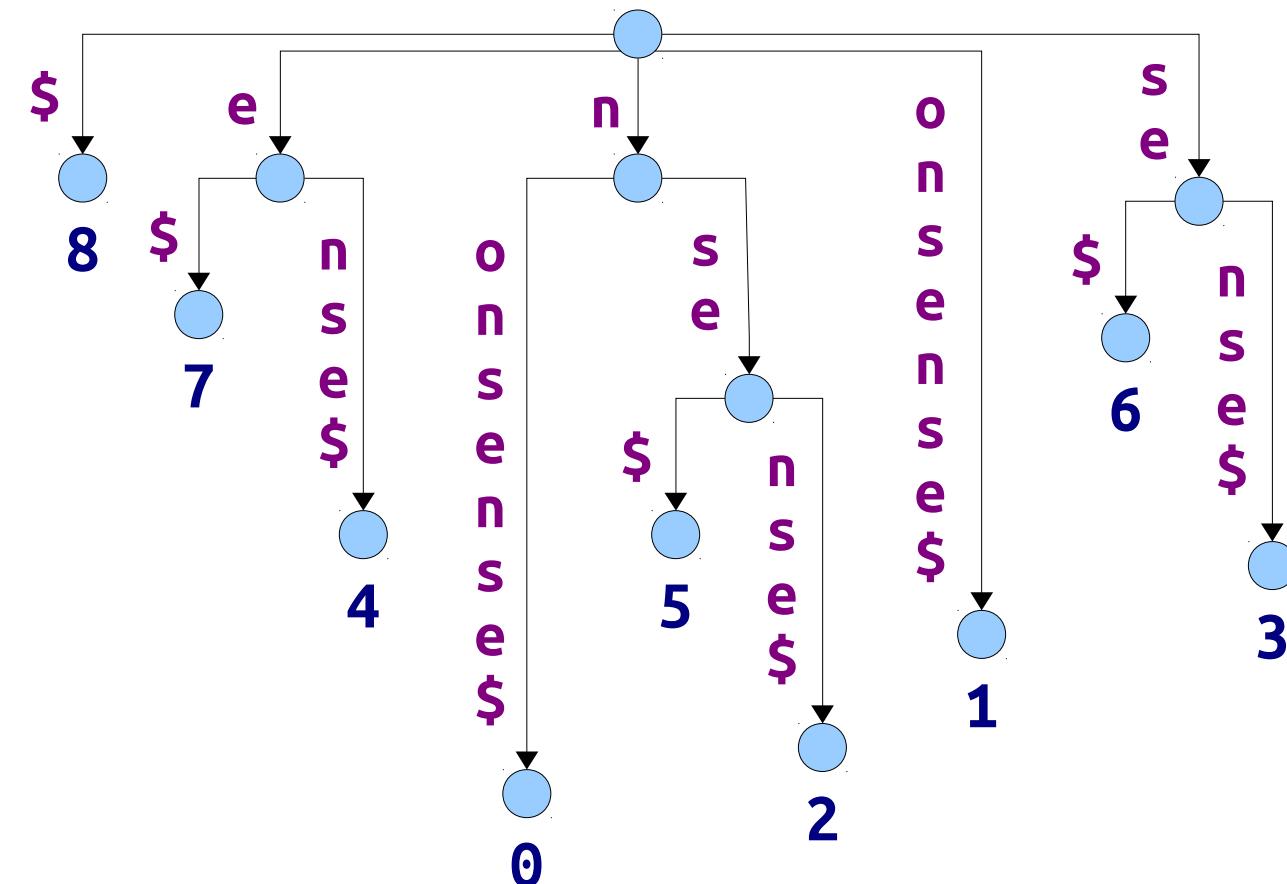
8	\$
7	e\$
4	ense\$
0	nonsense\$
5	nse\$
2	nsense\$
1	onsense\$
6	se\$
3	sense\$

Suffix Trees and Suffix Arrays



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

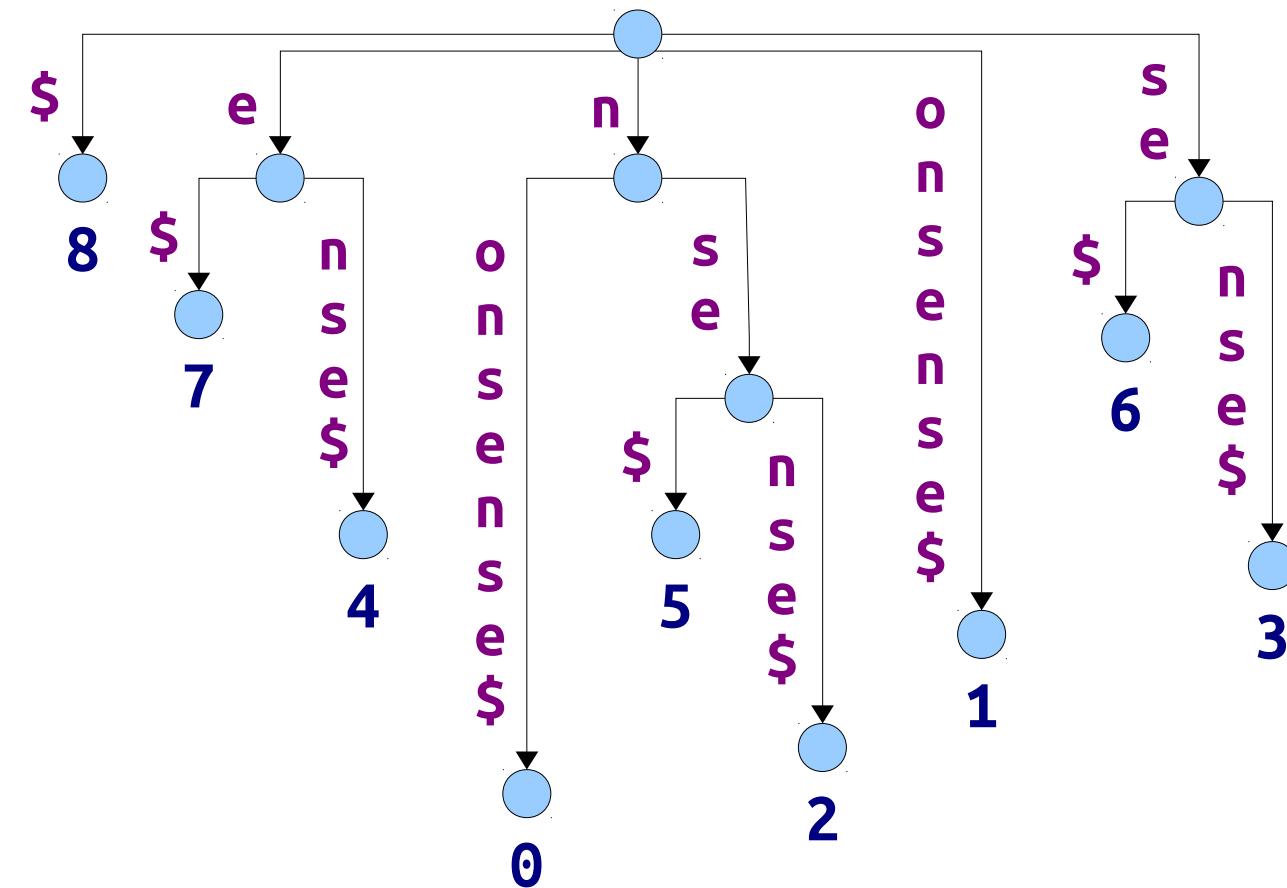
Suffix Trees and Suffix Arrays



**nonsense\$
012345678**

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

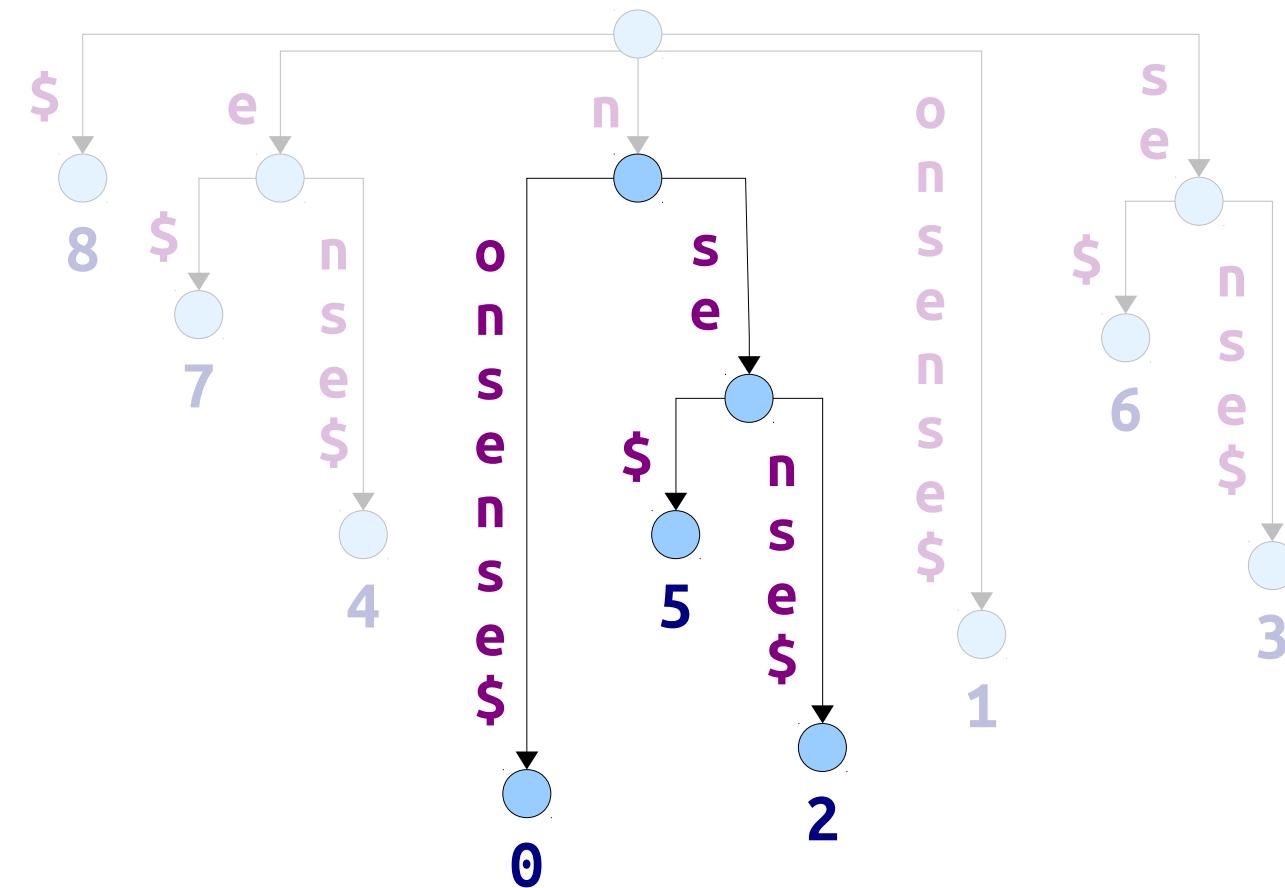
Suffix Trees and Suffix Arrays



**nonsense\$
012345678**

8	\$
7	e\$
4	ense\$
0	<u>nonsense\$</u>
5	<u>nse\$</u>
2	<u>n</u> sense\$
1	ononsense\$
6	se\$
3	sense\$

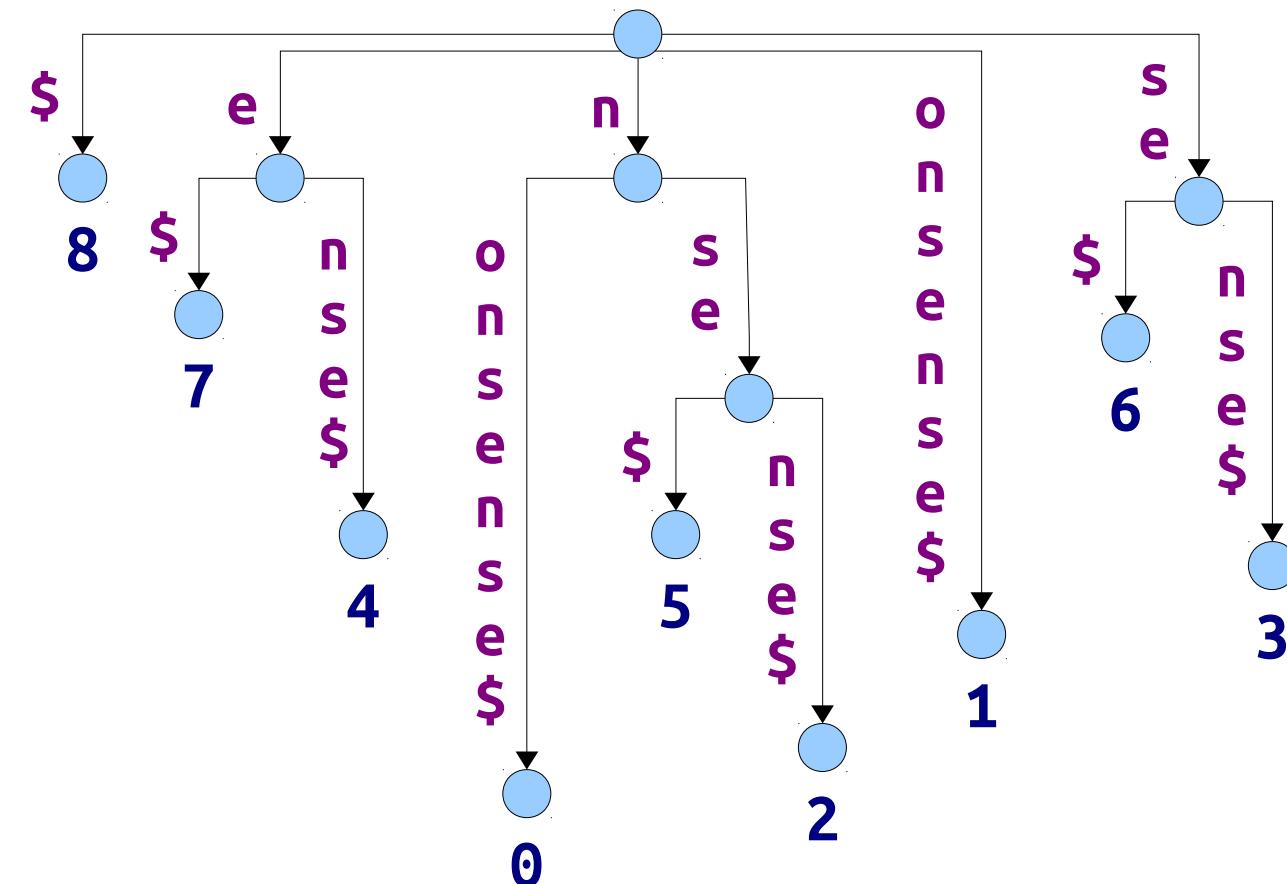
Suffix Trees and Suffix Arrays



nonsense\$
012345678

8	\$
7	e\$
4	ense\$
0	<u>nonsense\$</u>
5	<u>nse\$</u>
2	<u>n\$ense\$</u>
1	on\$ense\$
6	se\$
3	sense\$

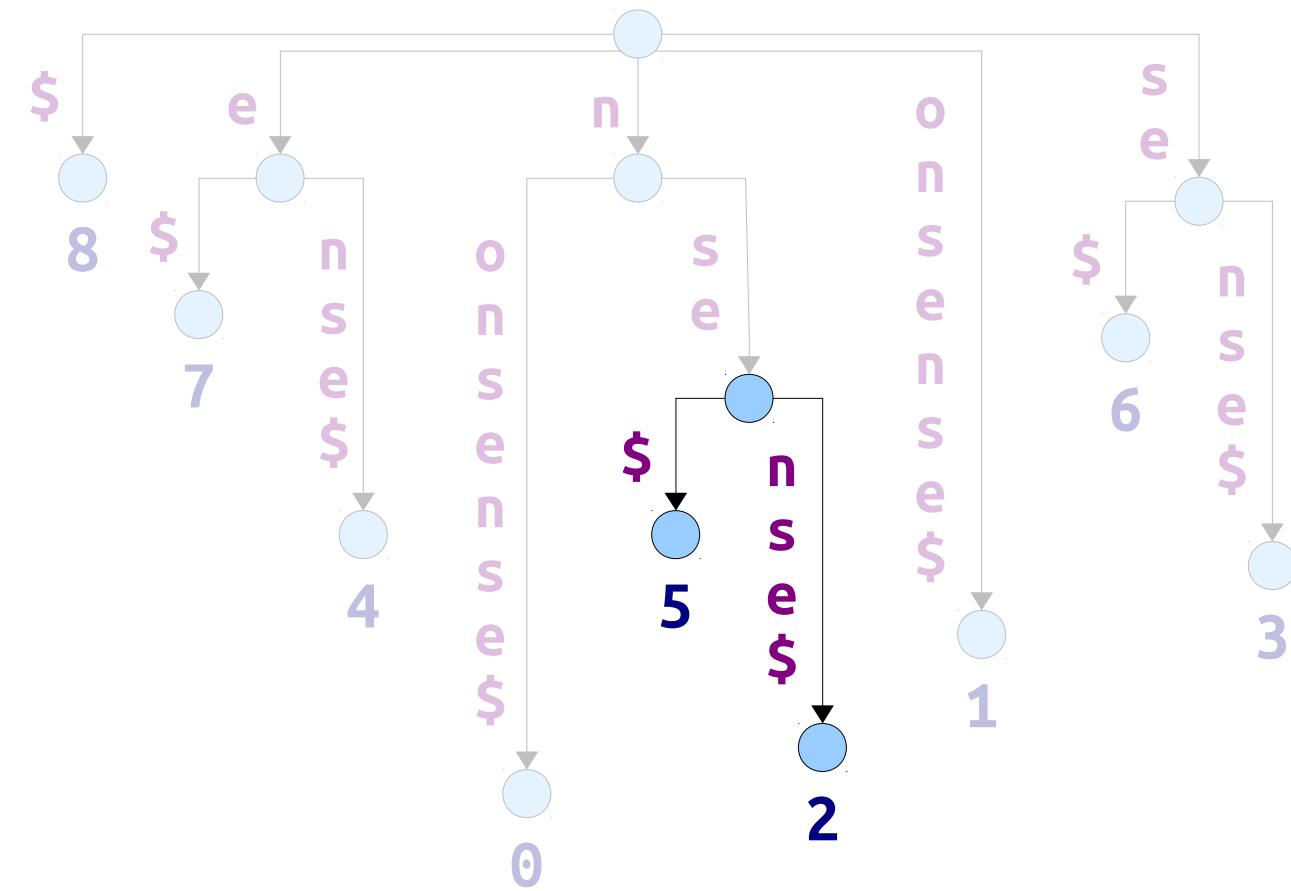
Suffix Trees and Suffix Arrays



**nonsense\$
012345678**

8	\$
7	e\$
4	ense\$
0	nonsense\$
5	<u>nse\$</u>
2	<u>n</u> sense\$
1	on sense\$
6	se\$
3	sense\$

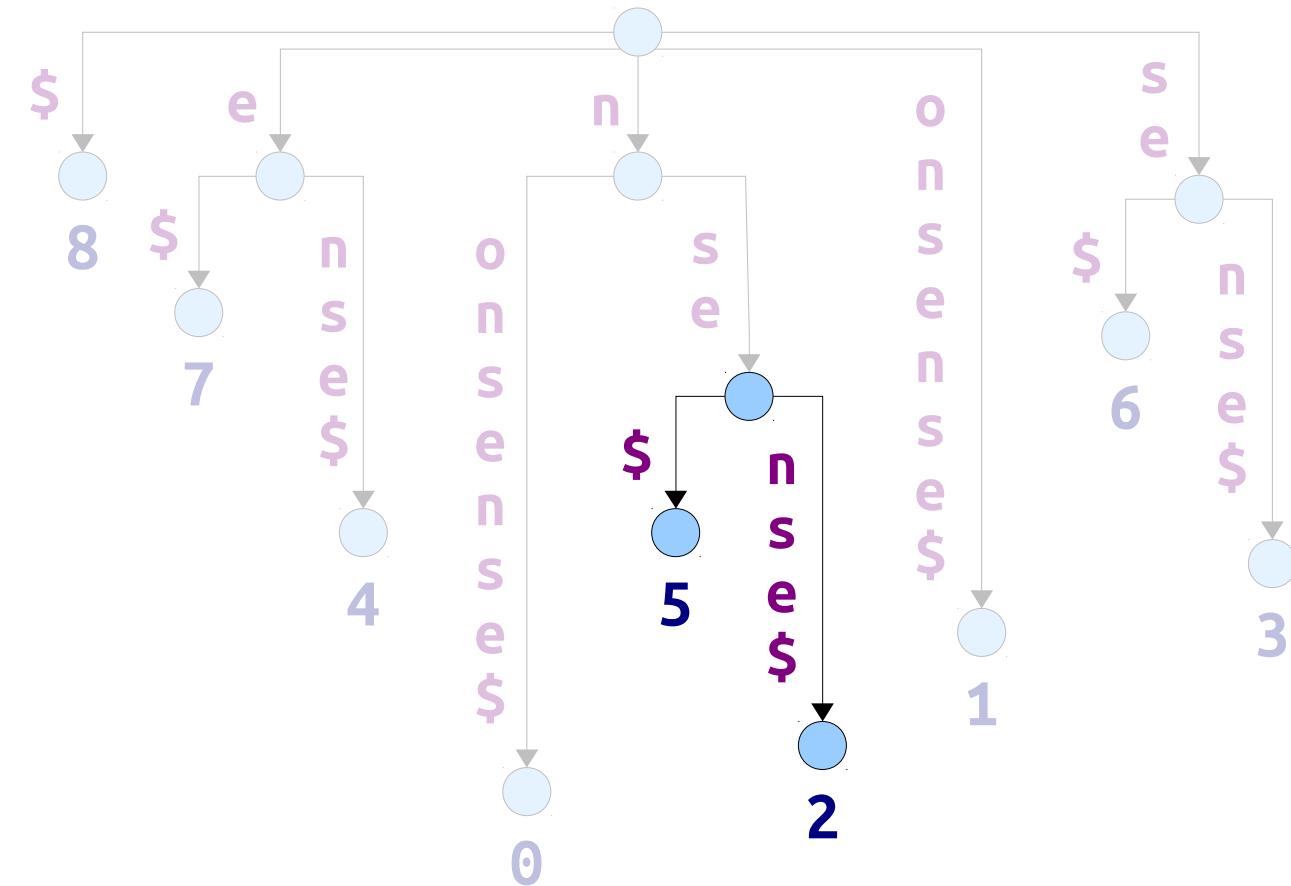
Suffix Trees and Suffix Arrays



**nonsense\$
012345678**

8	\$
7	e\$
4	ense\$
0	nonsense\$
5	<u>nse\$</u>
2	<u>n</u> sense\$
1	on sense\$
6	se\$
3	sense\$

Suffix Trees and Suffix Arrays



nonsense\$
012345678

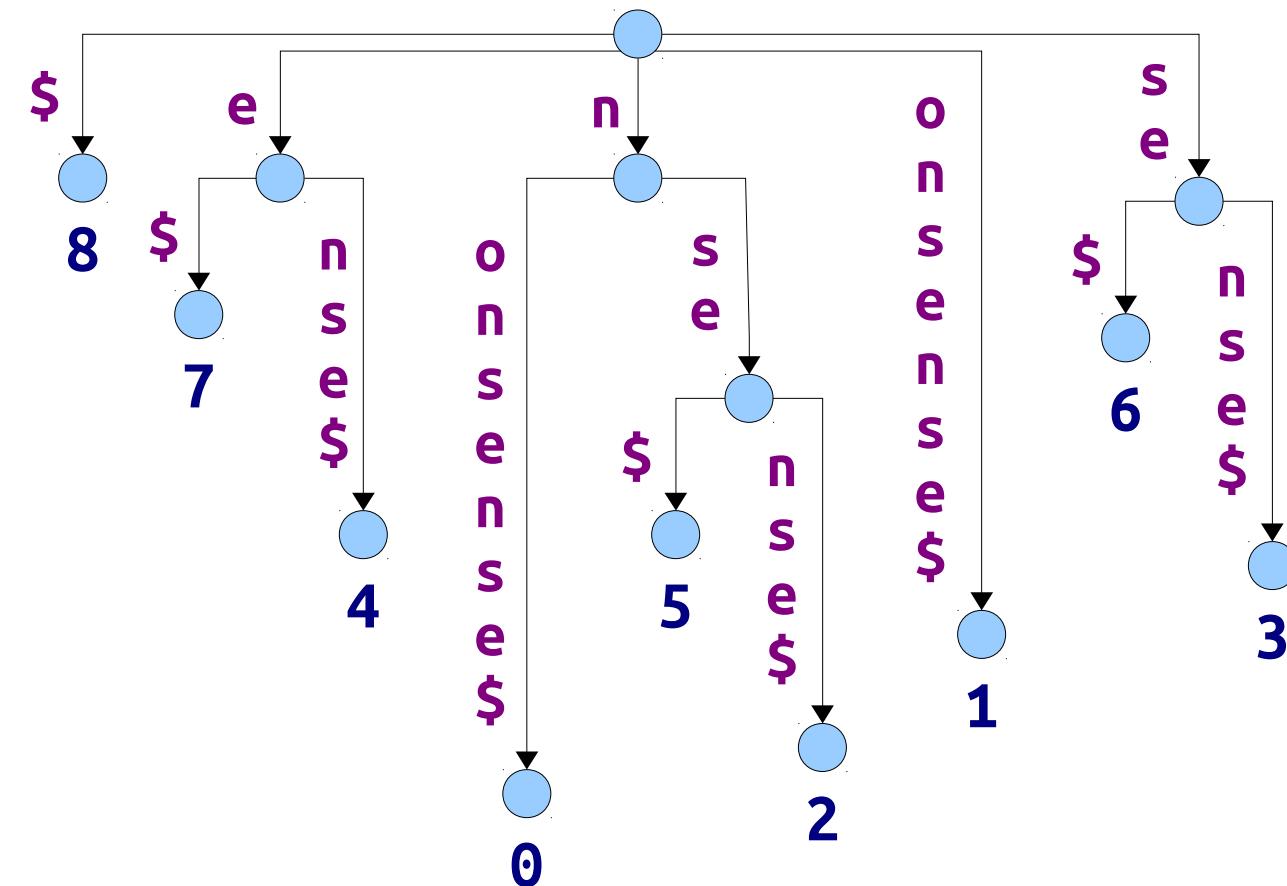
8	\$
7	e\$
4	ense\$
0	nonsense\$
5	<u>nse\$</u>
2	<u>n</u> sense\$
1	on <u>sense\$</u>
6	se\$
3	sense\$

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 $\text{lcp}(x, y)$.
- LCP information is fundamentally important for suffix arrays. With it, we can implicitly recover much of the structure present in suffix trees.

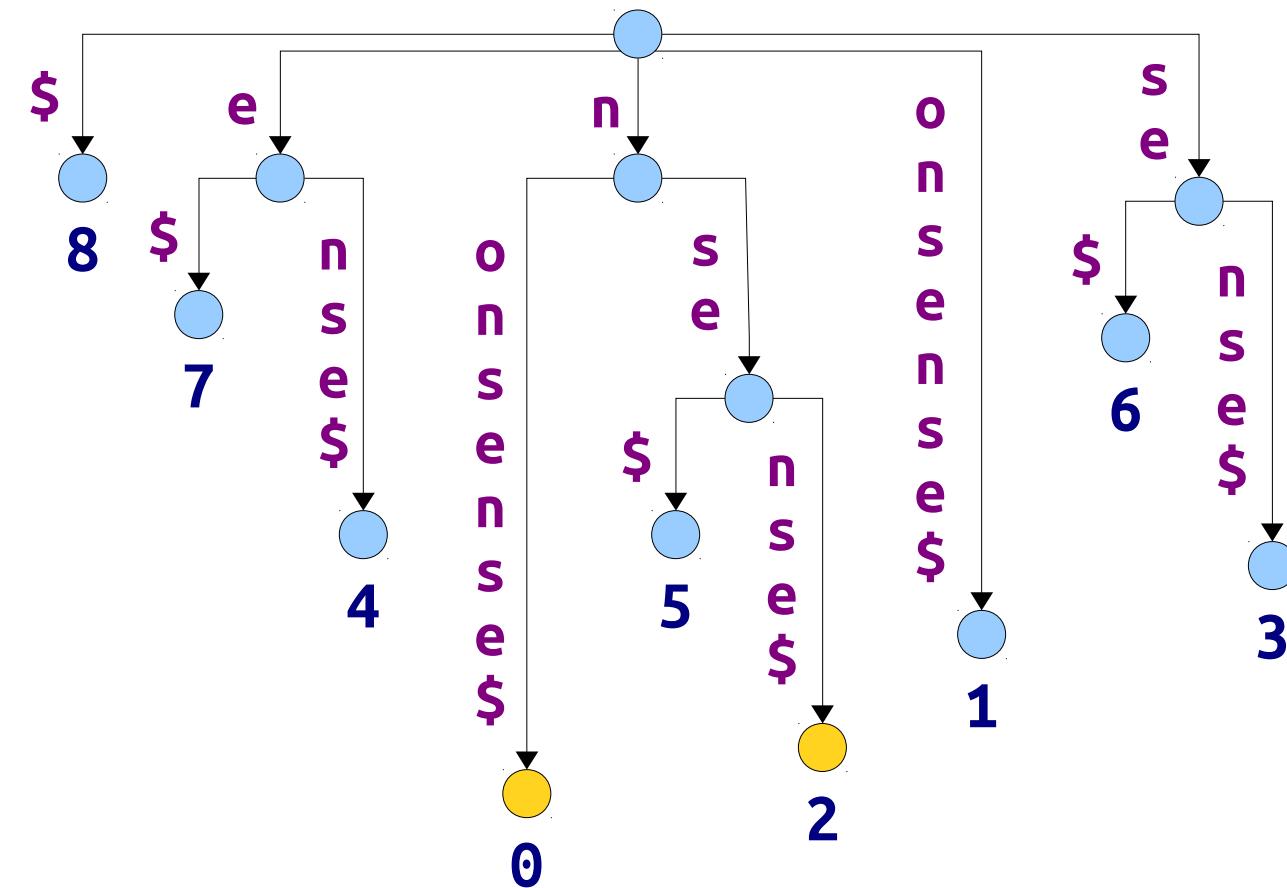
Suffix Trees and Suffix Arrays



nonsense\$
012345678

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

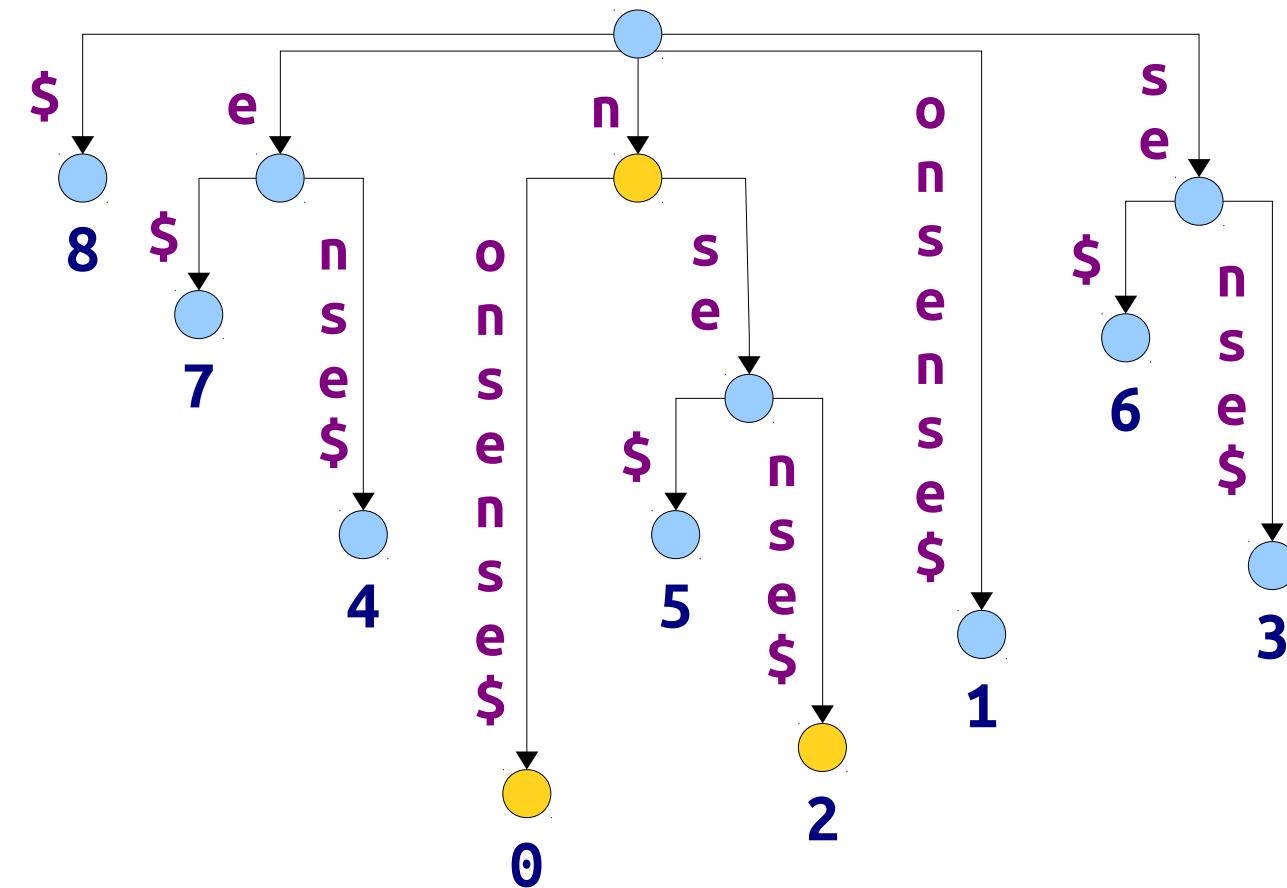
Suffix Trees and Suffix Arrays



nonsense\$
012345678

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

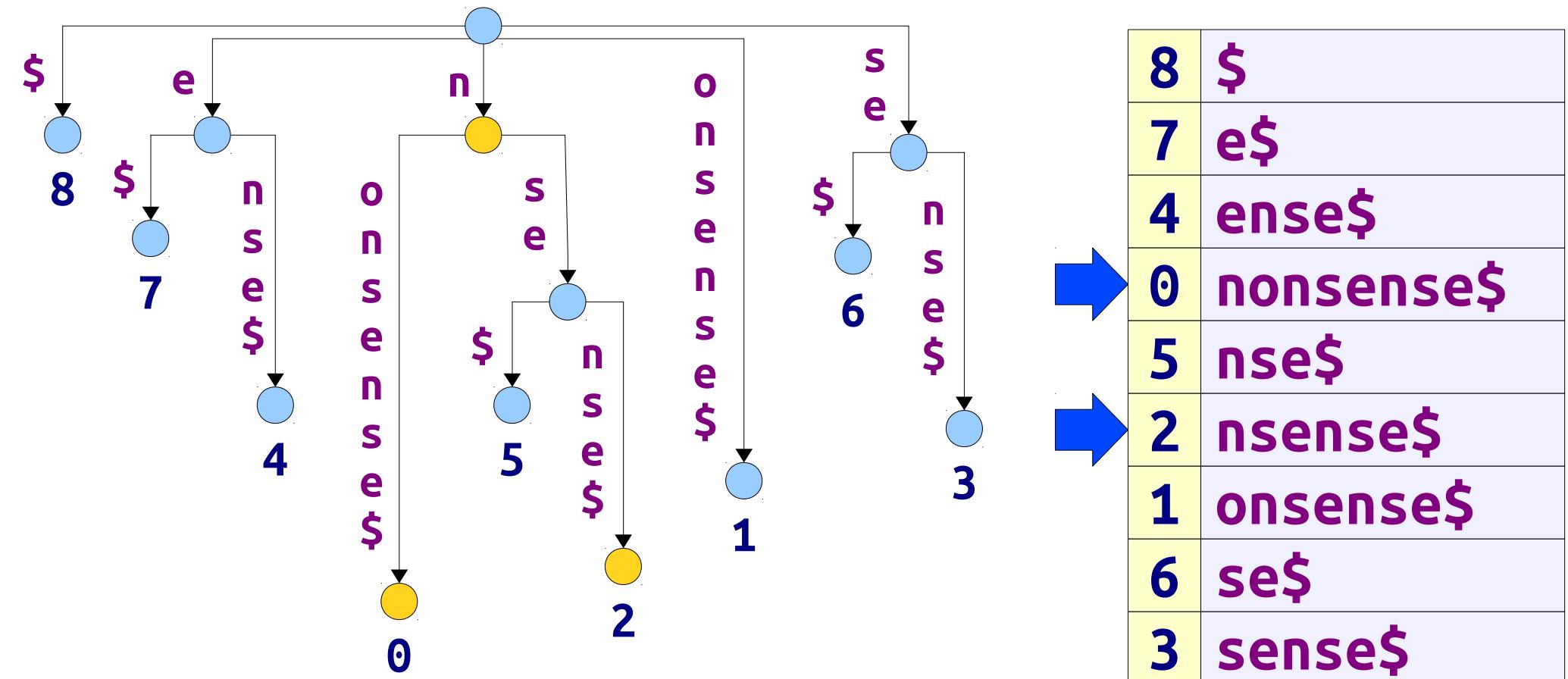
Suffix Trees and Suffix Arrays



nonsense\$
012345678

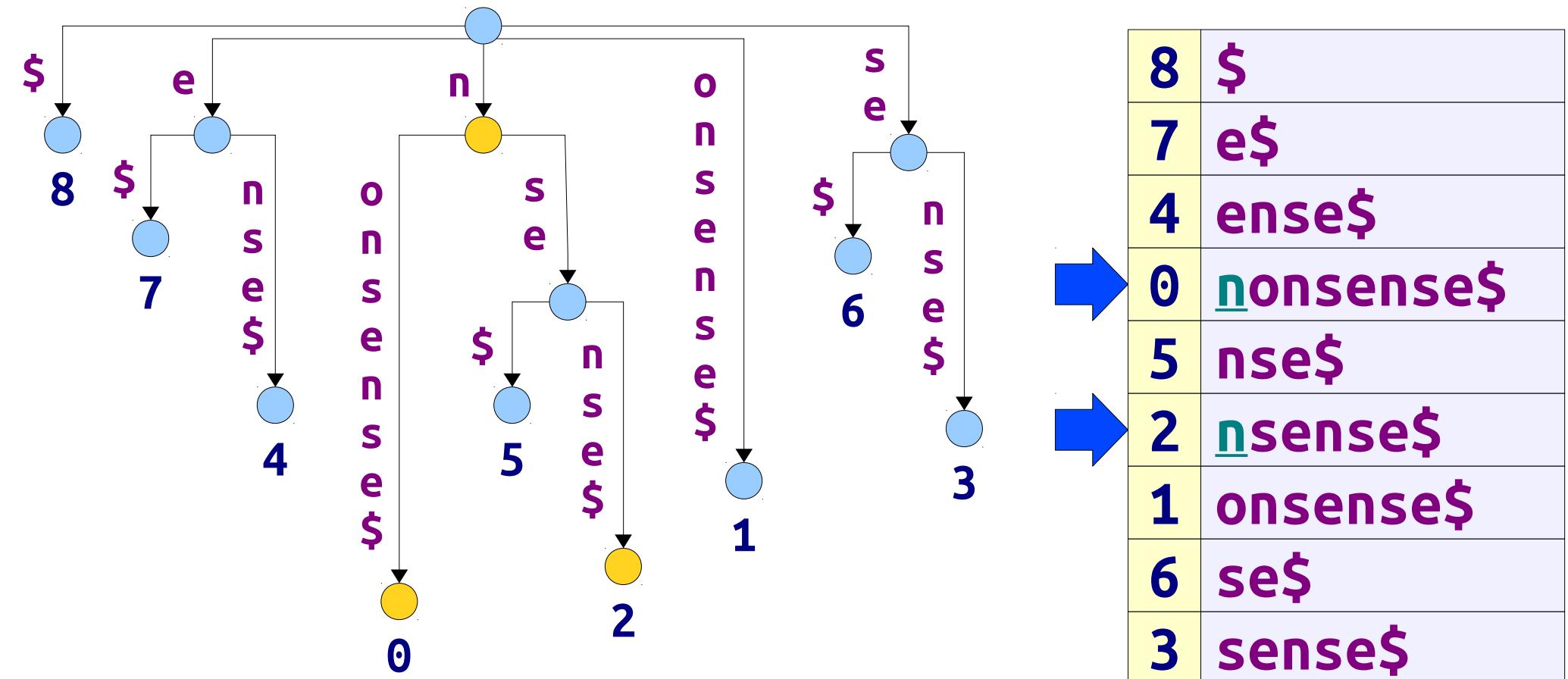
8	\$
7	e\$
4	ense\$
0	nonsense\$
5	nse\$
2	nsense\$
1	onsense\$
6	se\$
3	sense\$

Suffix Trees and Suffix Arrays



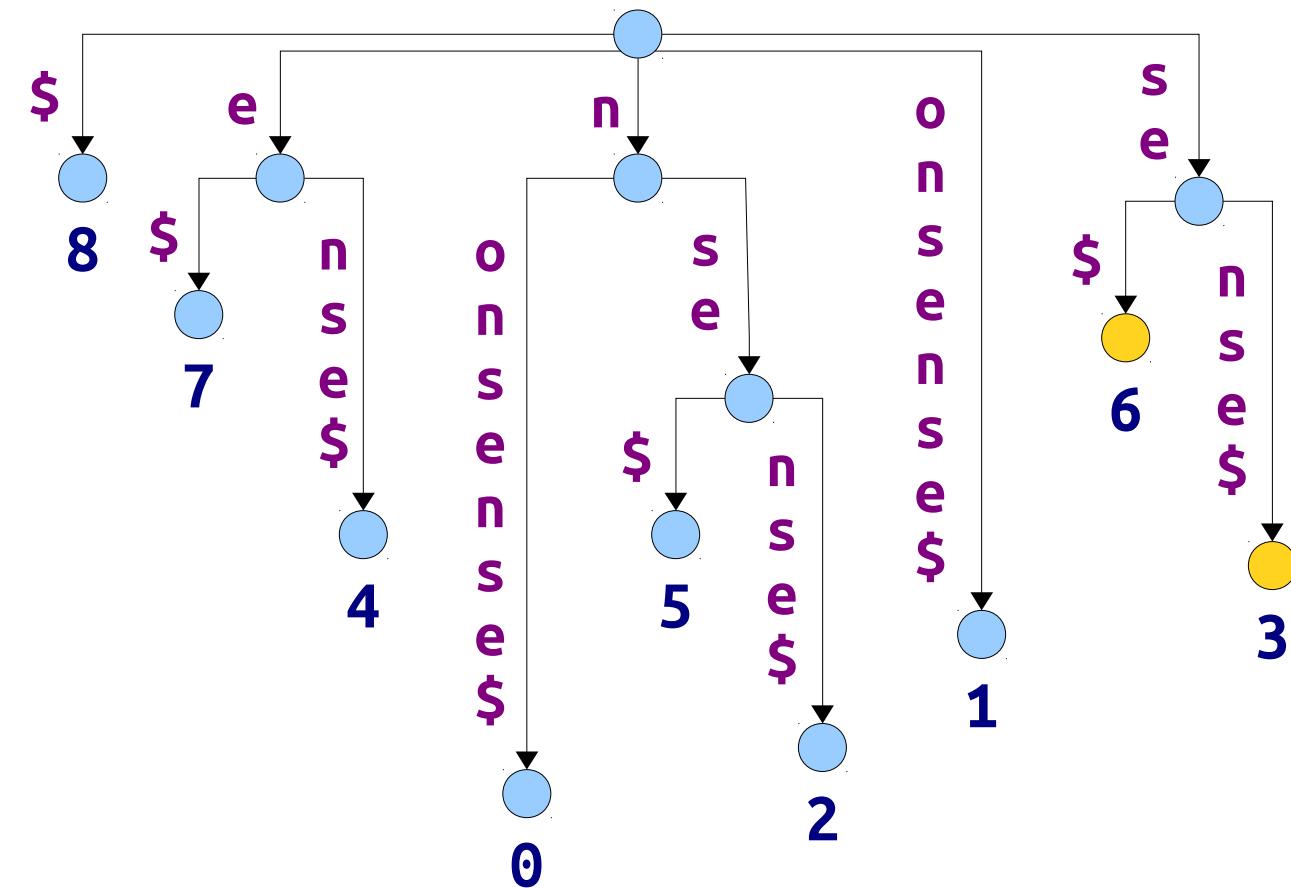
nonsense\$
012345678

Suffix Trees and Suffix Arrays



nonsense\$
012345678

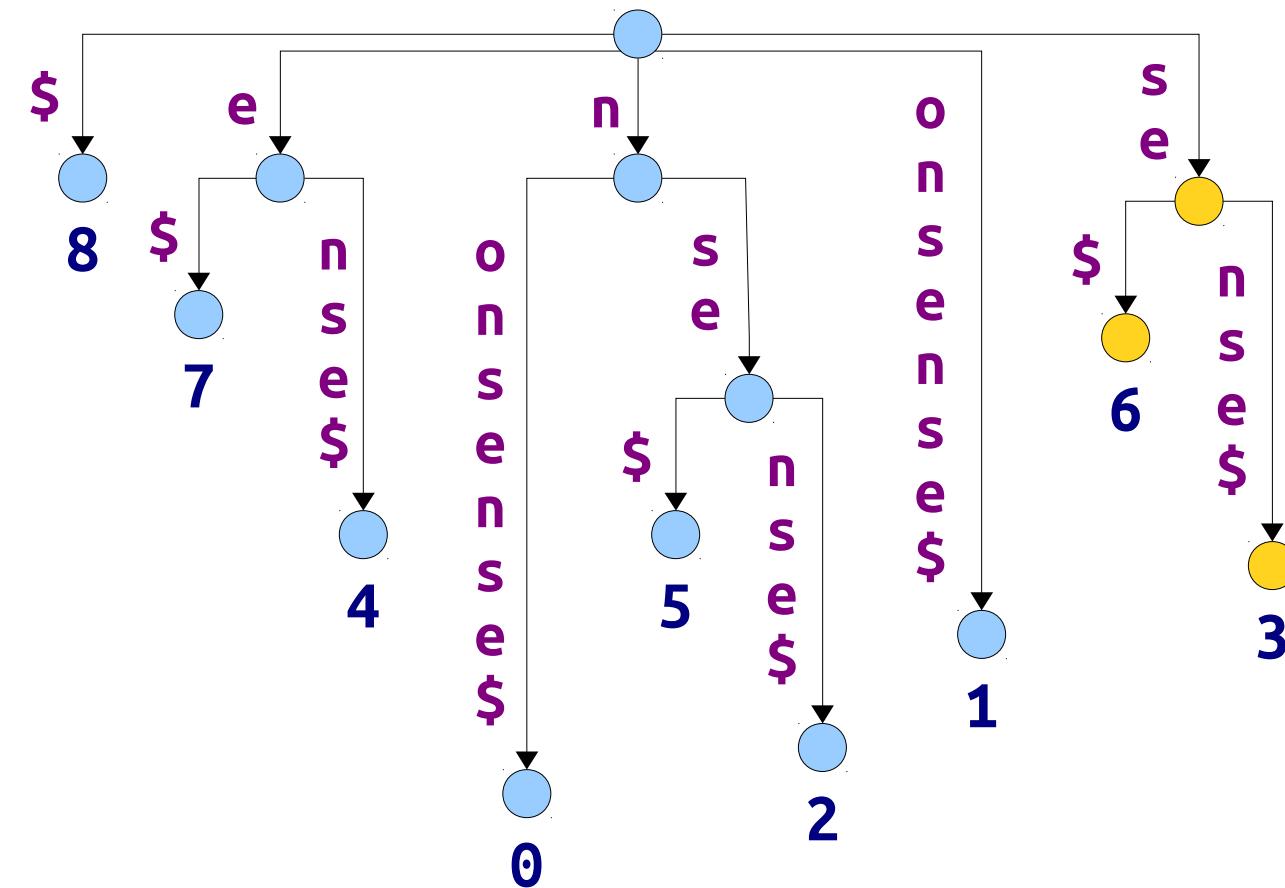
Suffix Trees and Suffix Arrays



nonsense\$
012345678

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

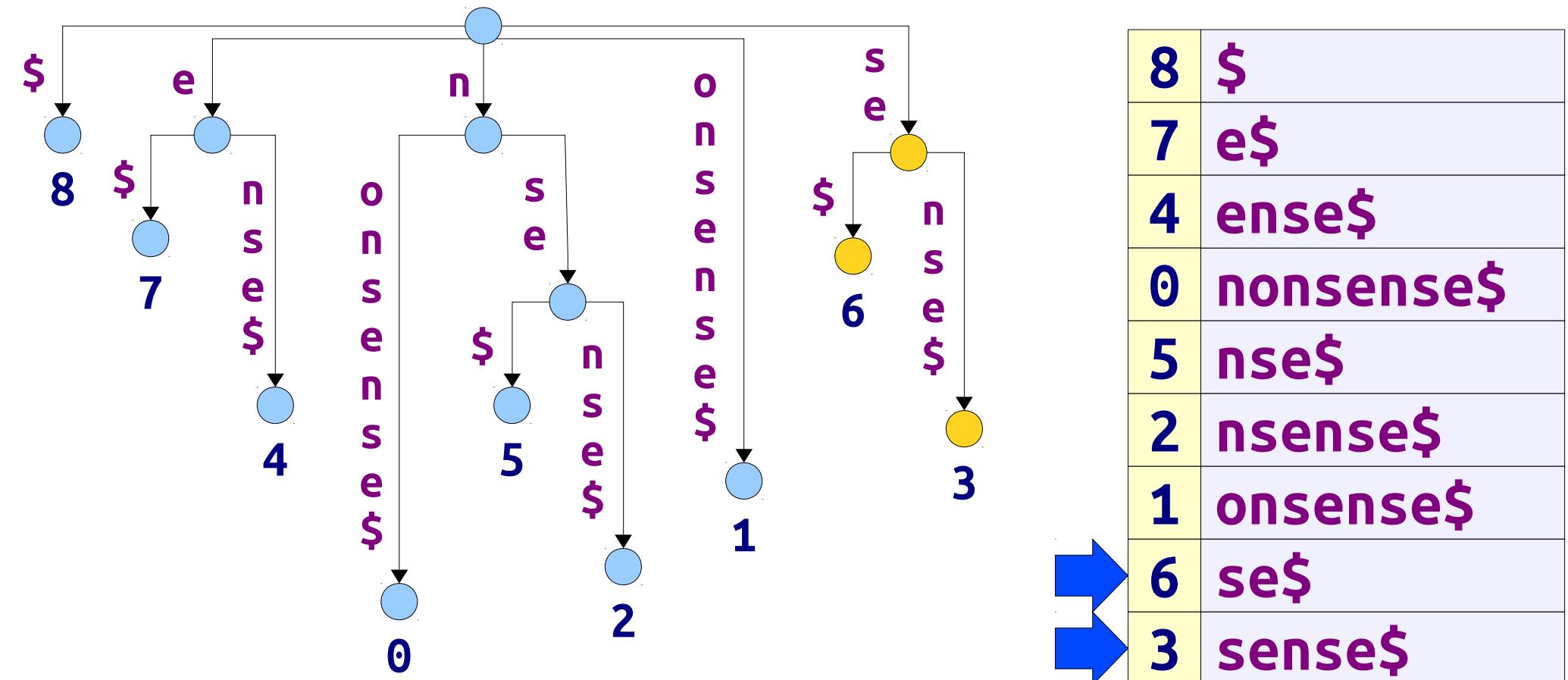
Suffix Trees and Suffix Arrays



nonsense\$
012345678

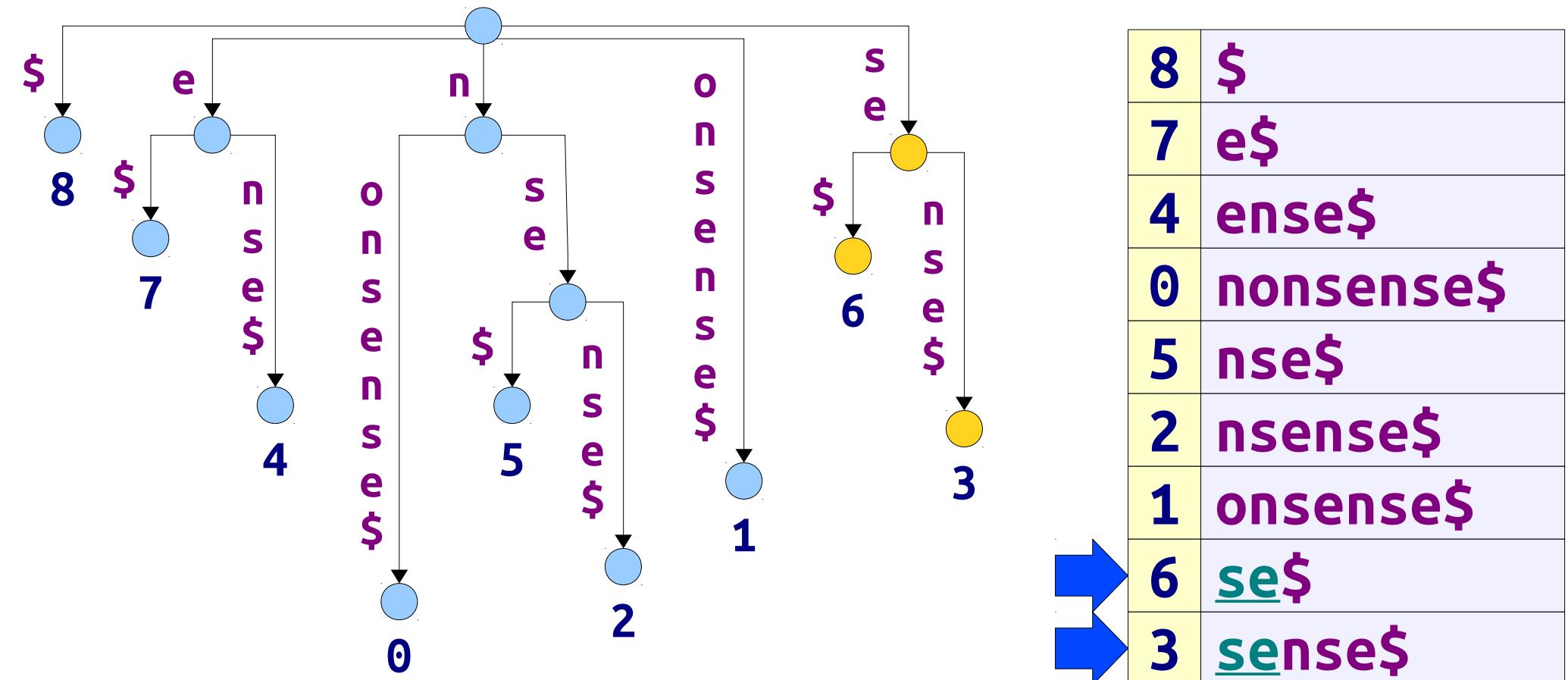
8	\$
7	e\$
4	ense\$
0	nonsense\$
5	nse\$
2	nsense\$
1	onsense\$
6	se\$
3	sense\$

Suffix Trees and Suffix Arrays



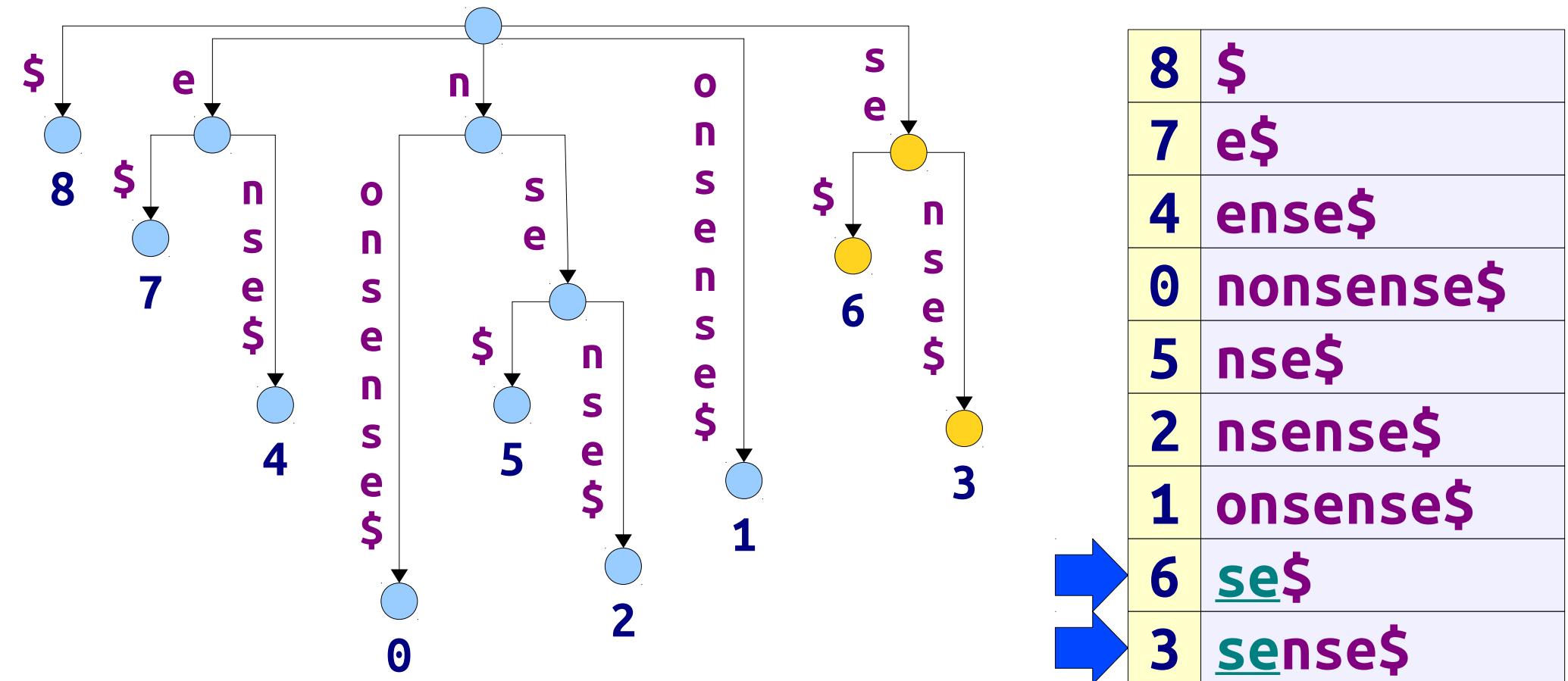
nonsense\$
012345678

Suffix Trees and Suffix Arrays



nonsense\$
012345678

Suffix Trees and Suffix Arrays



nonsense\$
012345678

Nifty Fact: The lowest common ancestor of suffixes x and y has string label given by $\text{lcp}(x, y)$.

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.

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

Pairwise LCP

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

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

Pairwise LCP

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

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

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

0	8	\$
1	7	e\$
0	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 i th suffix in the suffix array.
 - $H[i]$ is the value of $\text{lcp}(SA[i], SA[i + 1])$

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

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

The diagram illustrates the construction of an RMQ table from a suffix array. On the left, a vertical column of suffixes is shown, each preceded by a yellow index. Blue arrows point from these indices to the corresponding entries in a table on the right. The table has 10 rows, indexed from 0 to 9 on the left. The first row contains the character '\$'. Subsequent rows contain suffixes starting from 'e\$' up to 'sense\$'. The value in the second column of each row is the RMQ value for the range indicated by the index in the first column.

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

Pairwise LCP

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

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

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

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

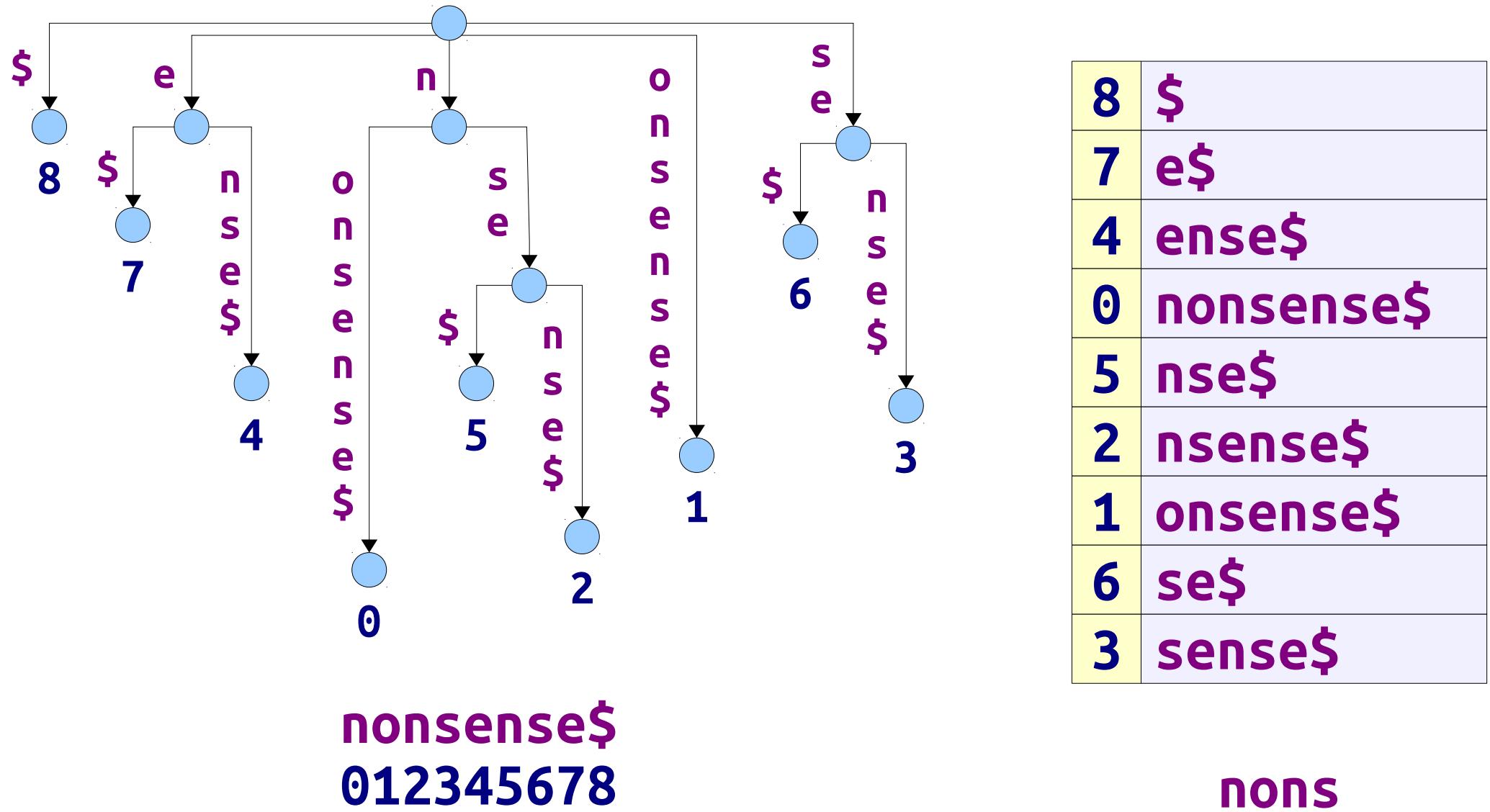
Computing LCPs

- 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.

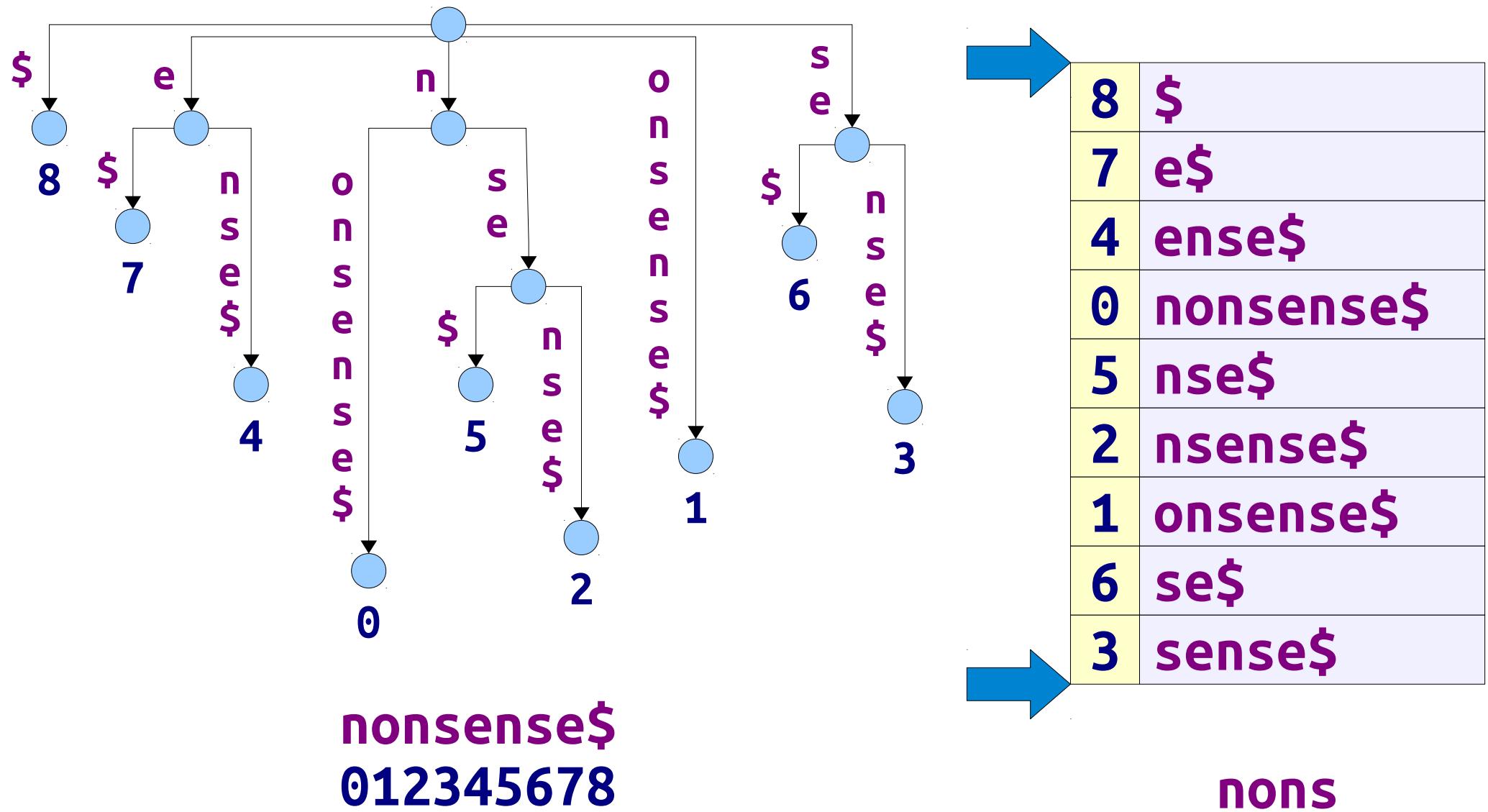
Searching a Suffix Array

- **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.

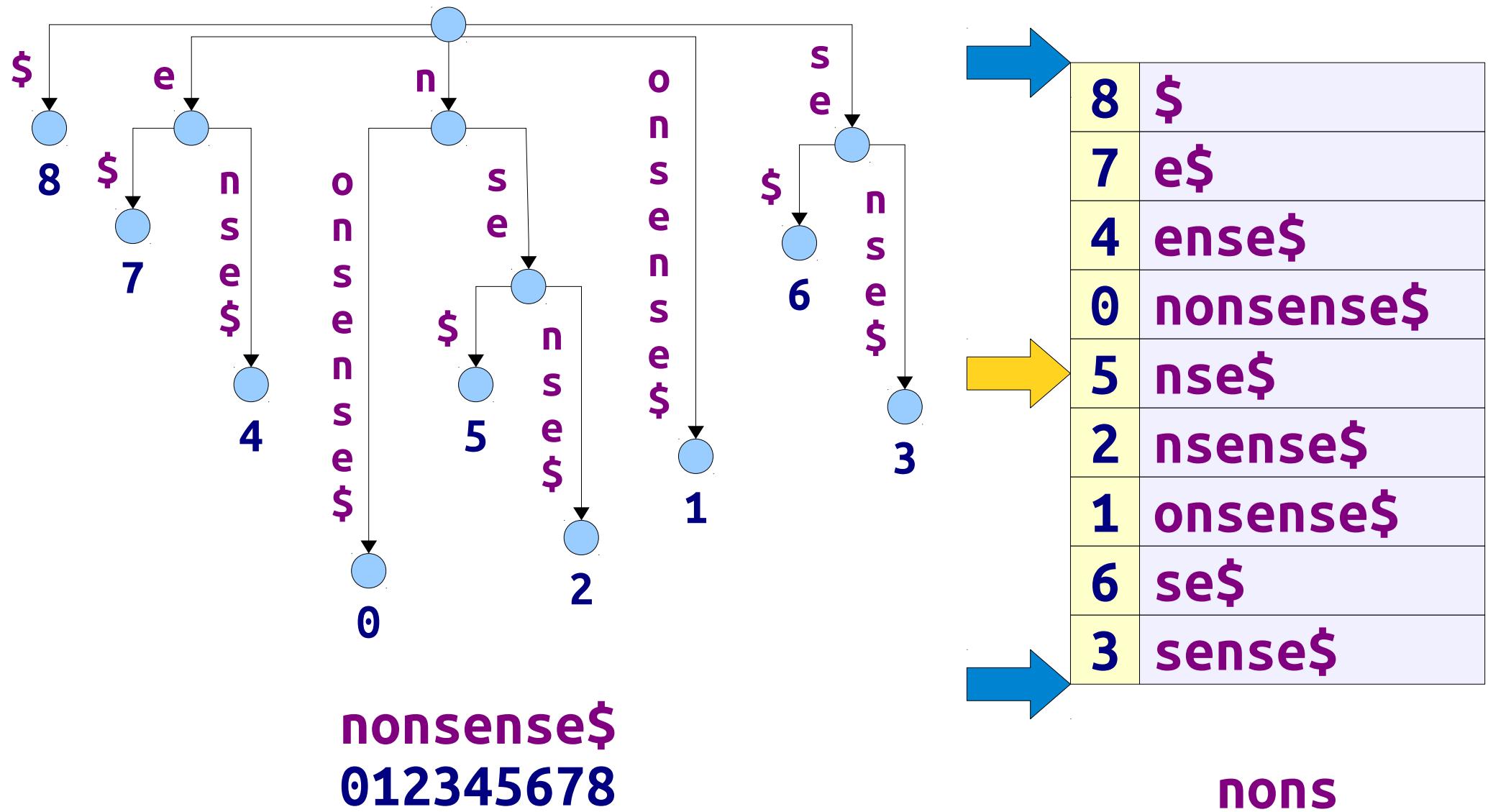
Searching a Suffix Array



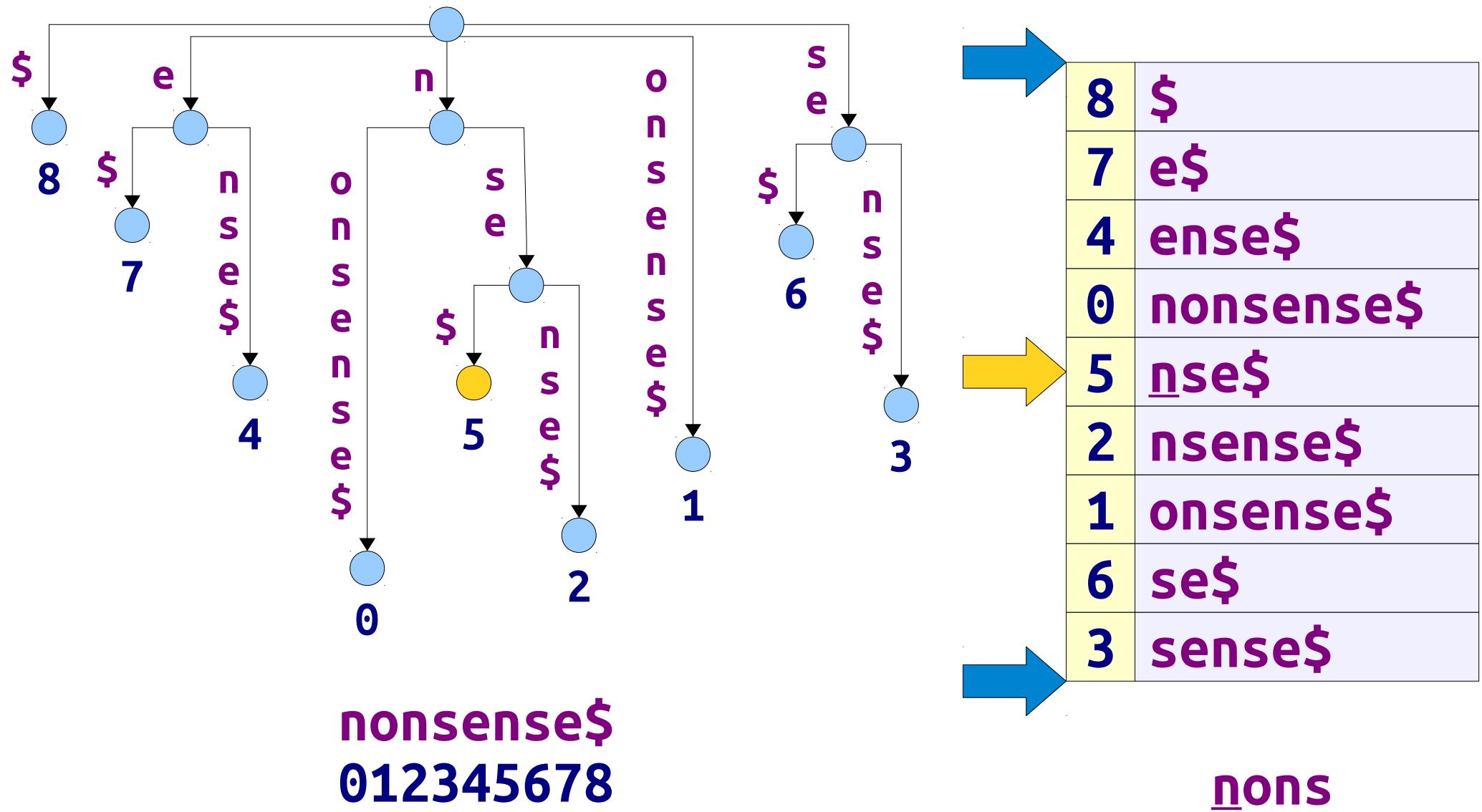
Searching a Suffix Array



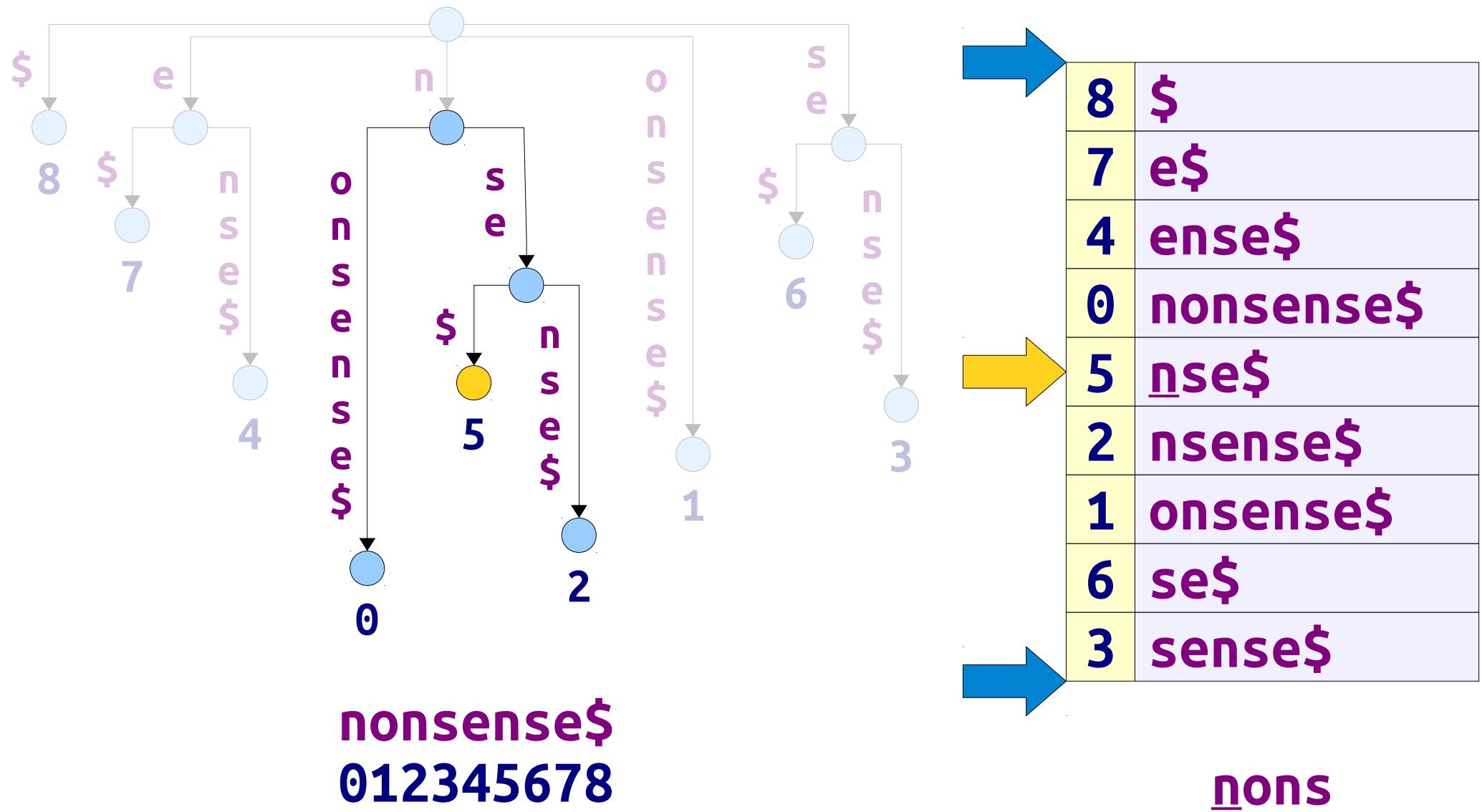
Searching a Suffix Array



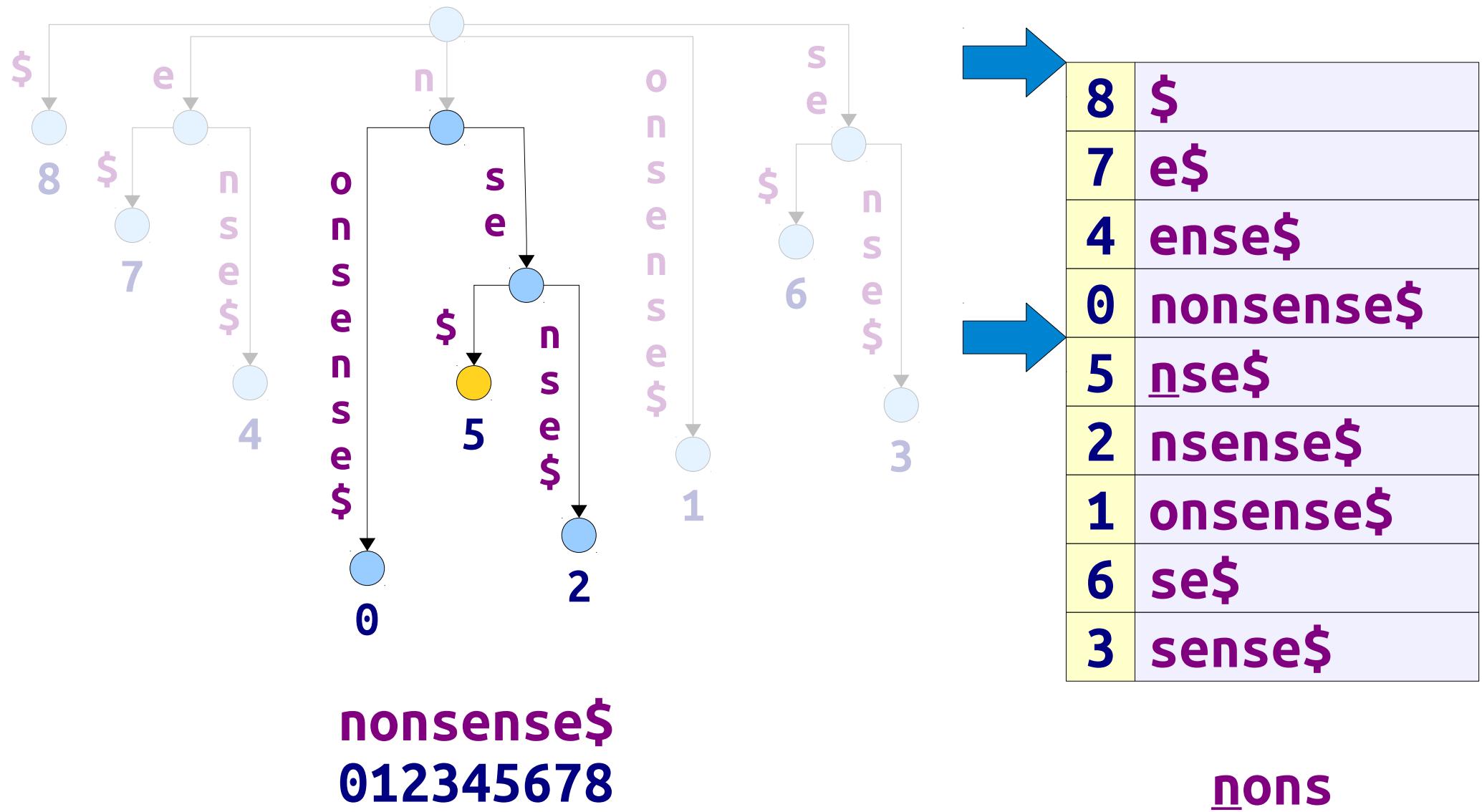
Searching a Suffix Array



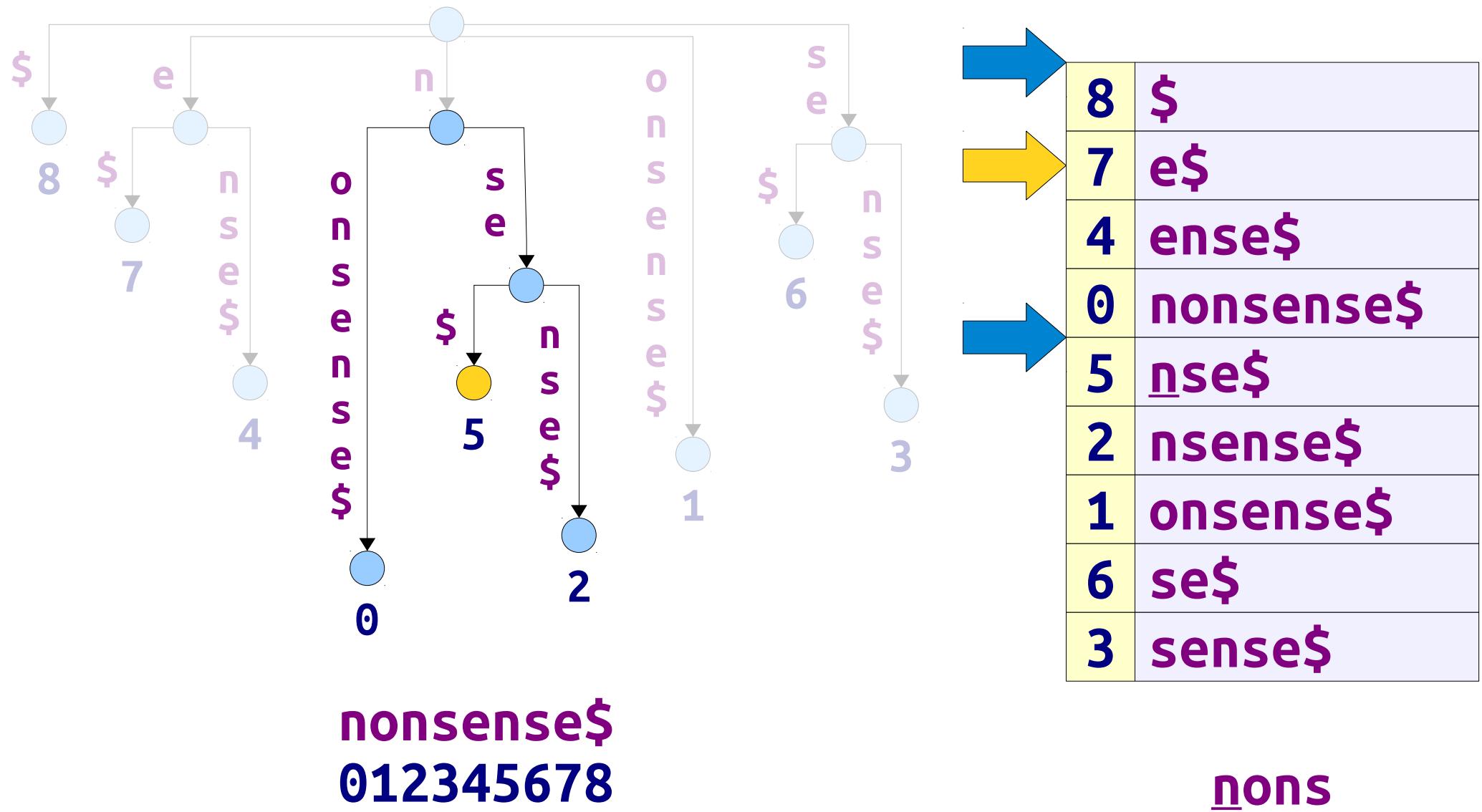
Searching a Suffix Array



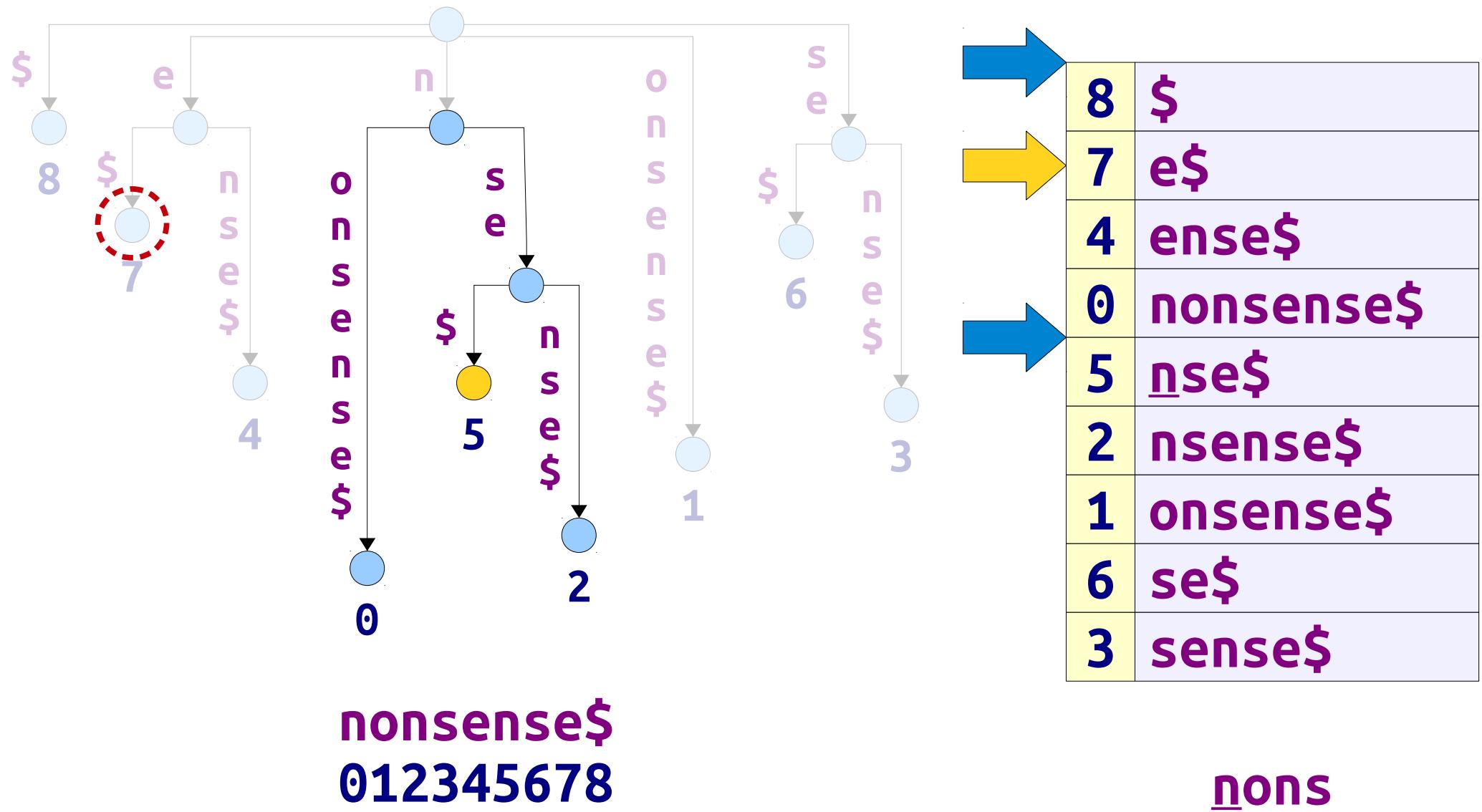
Searching a Suffix Array



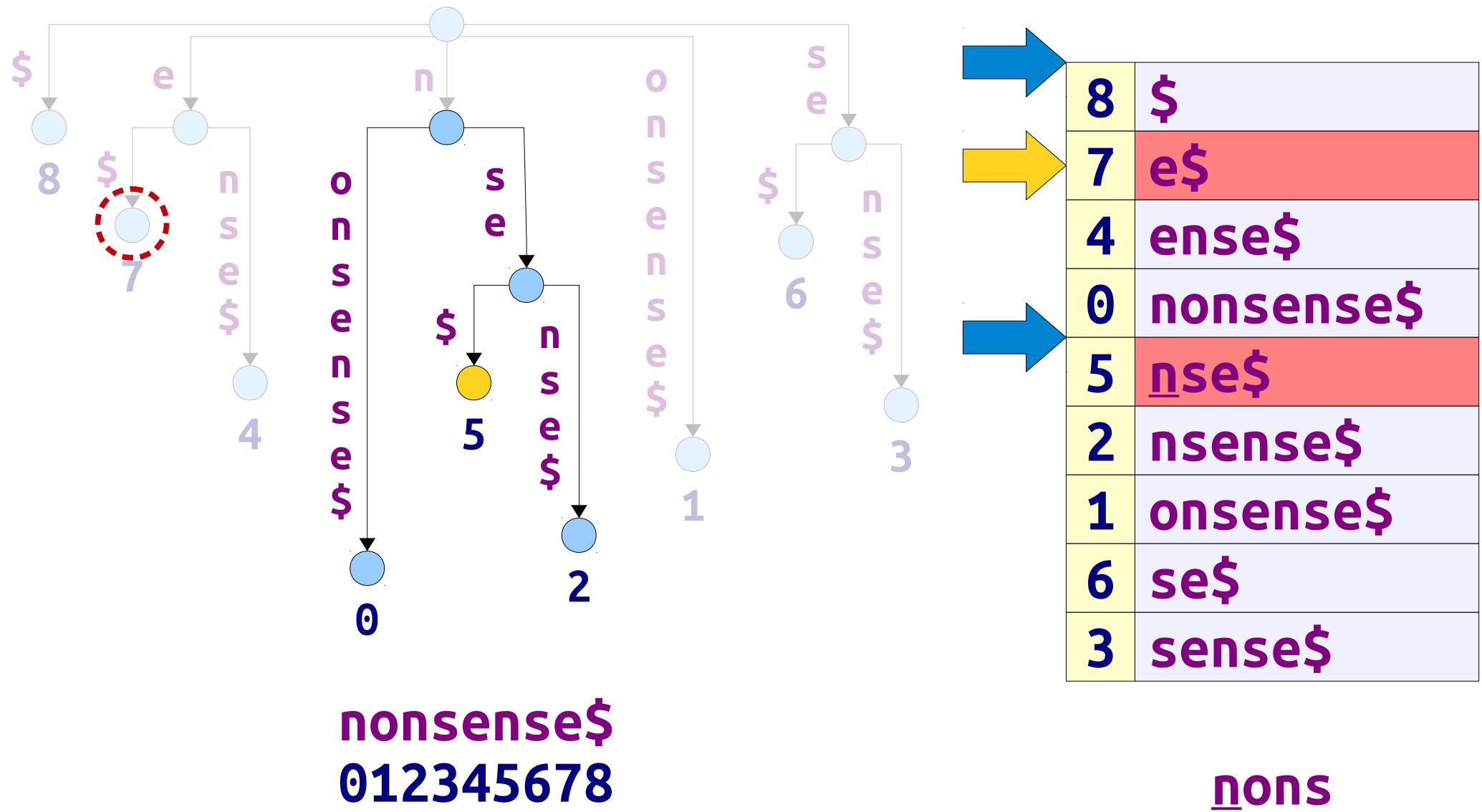
Searching a Suffix Array



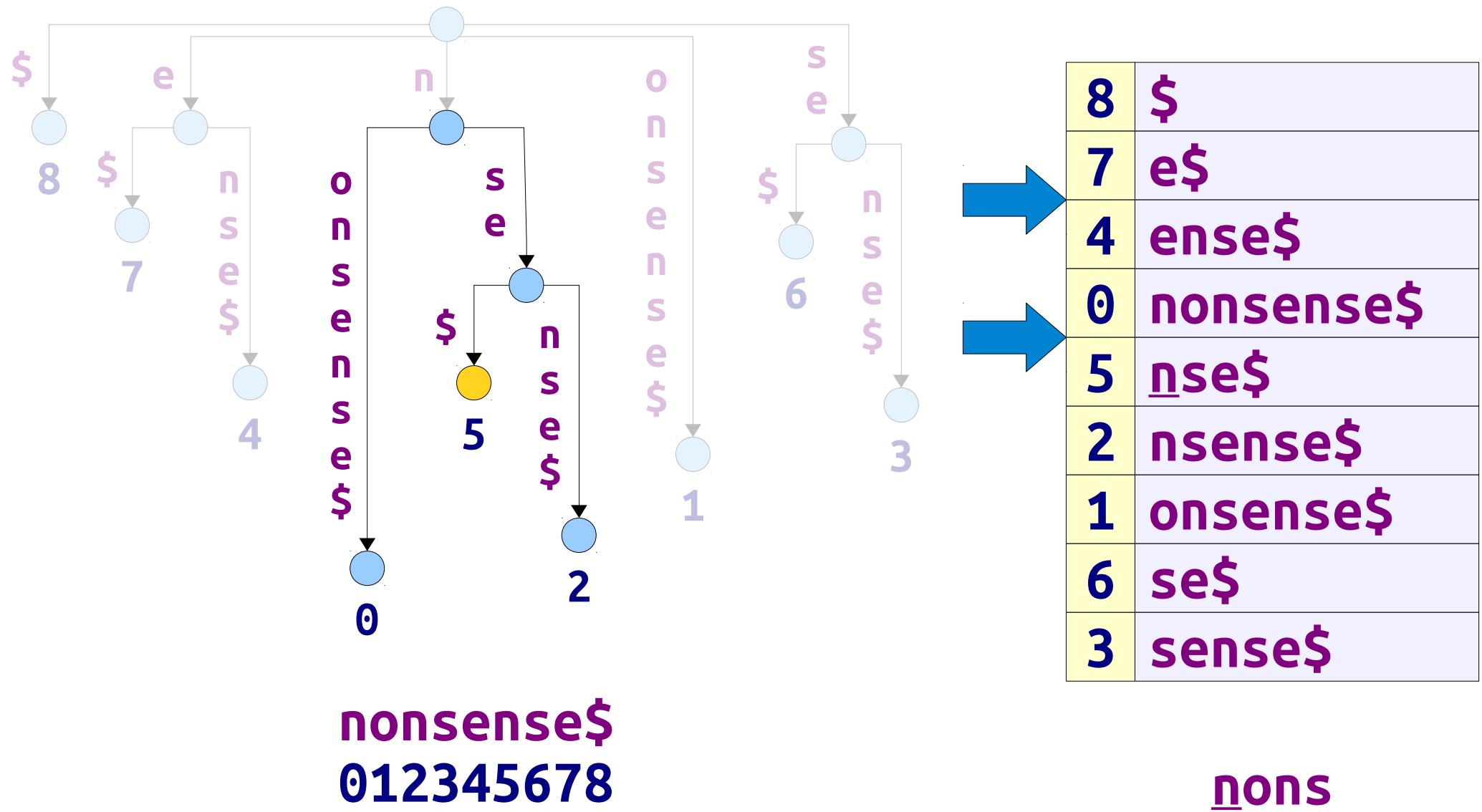
Searching a Suffix Array



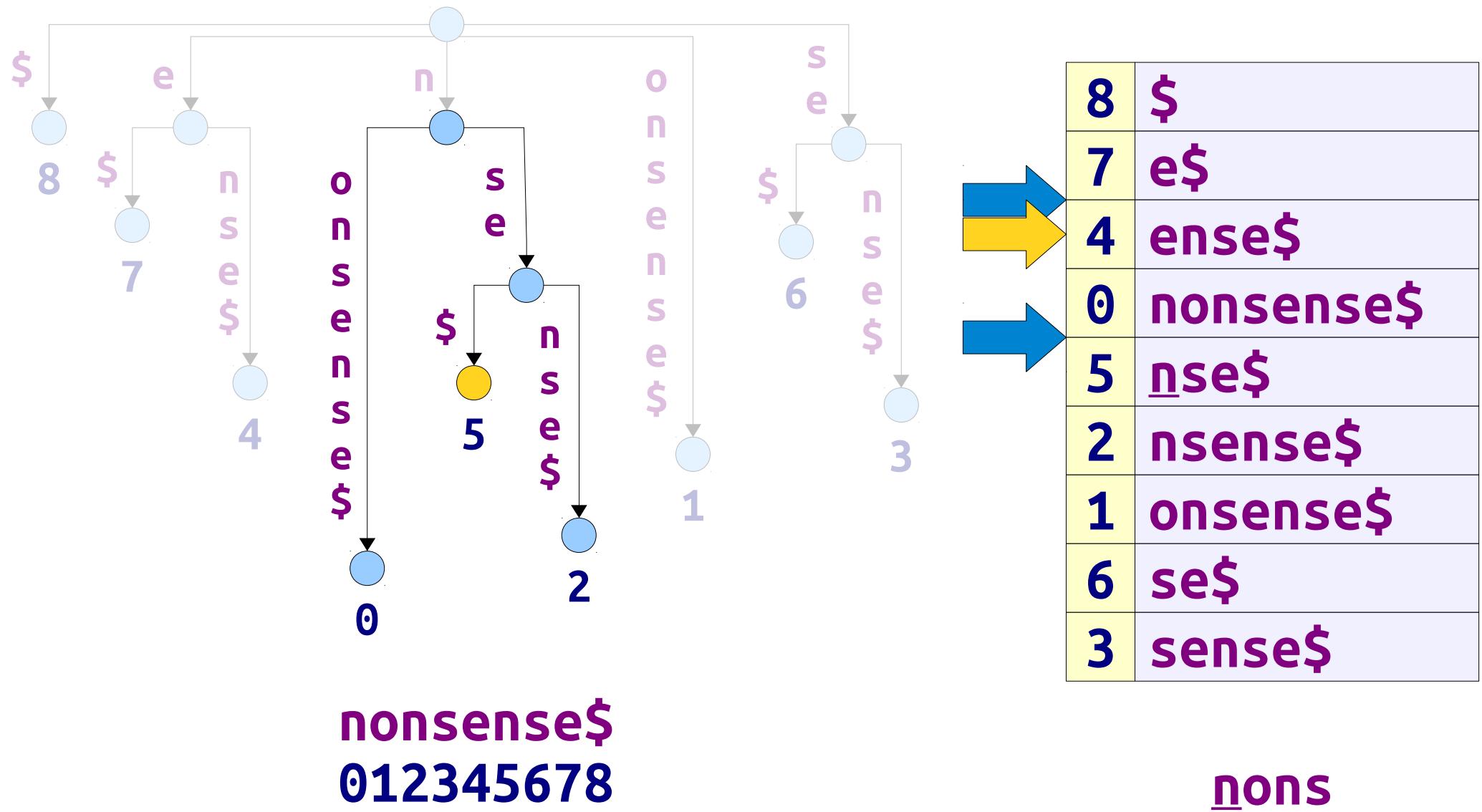
Searching a Suffix Array



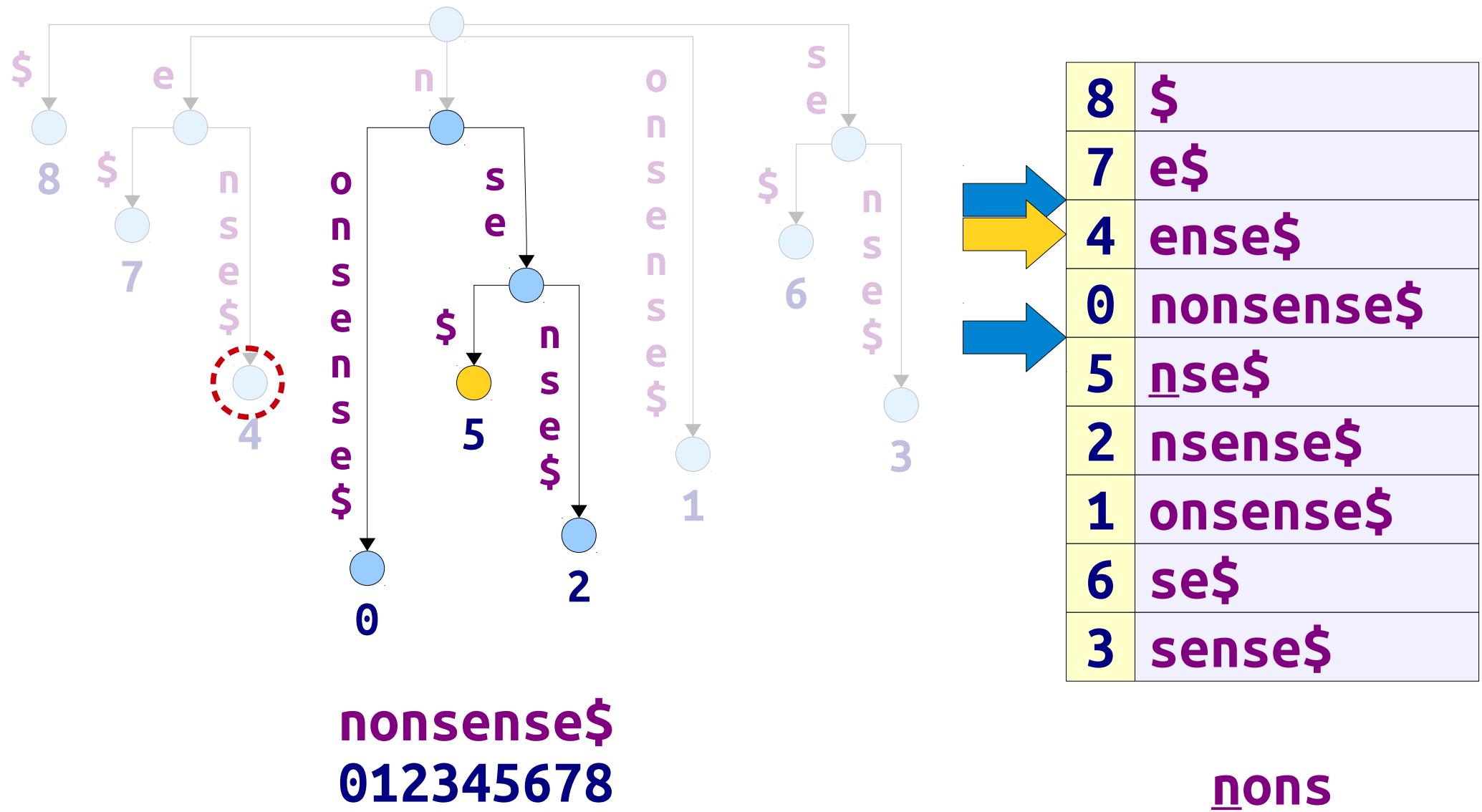
Searching a Suffix Array



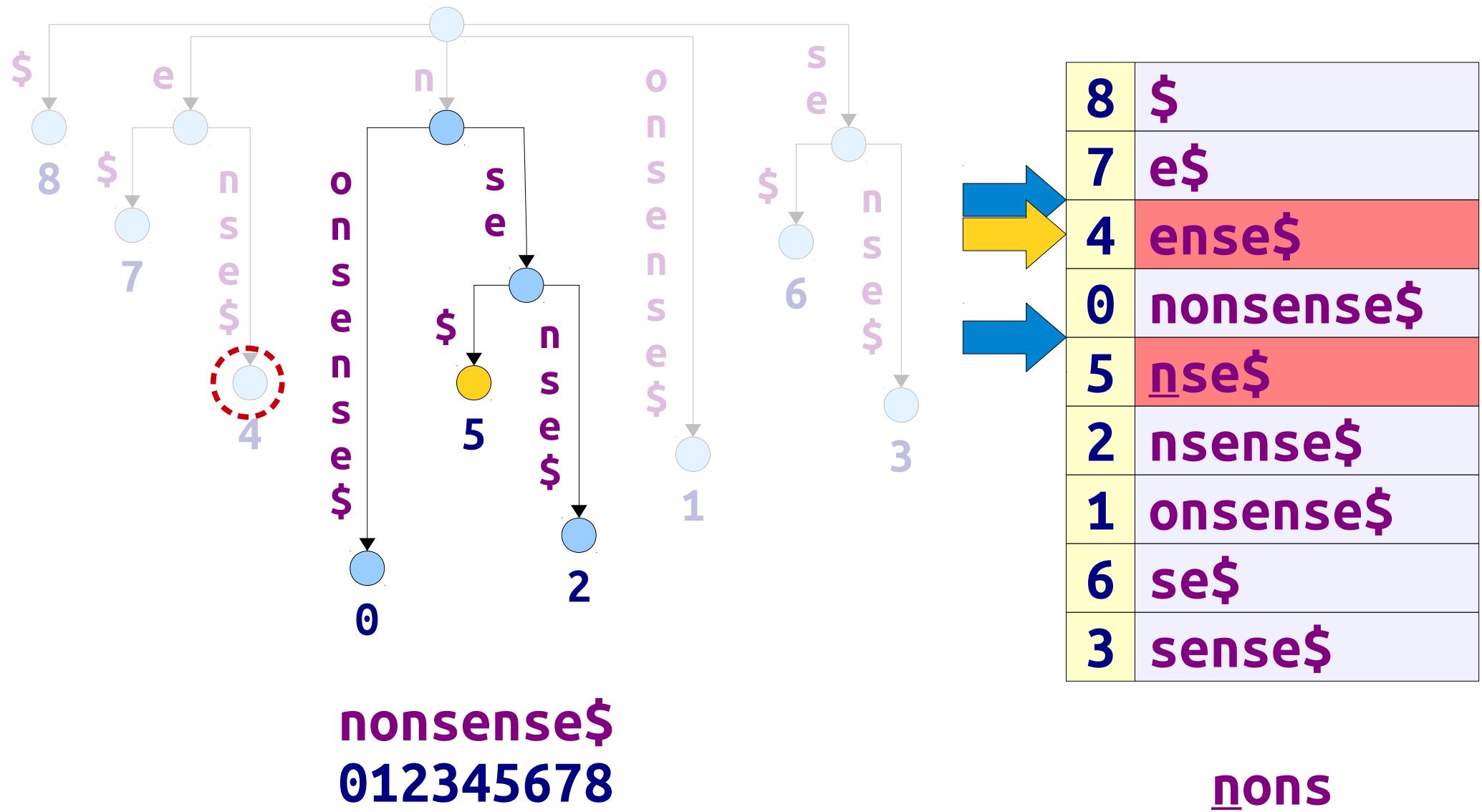
Searching a Suffix Array



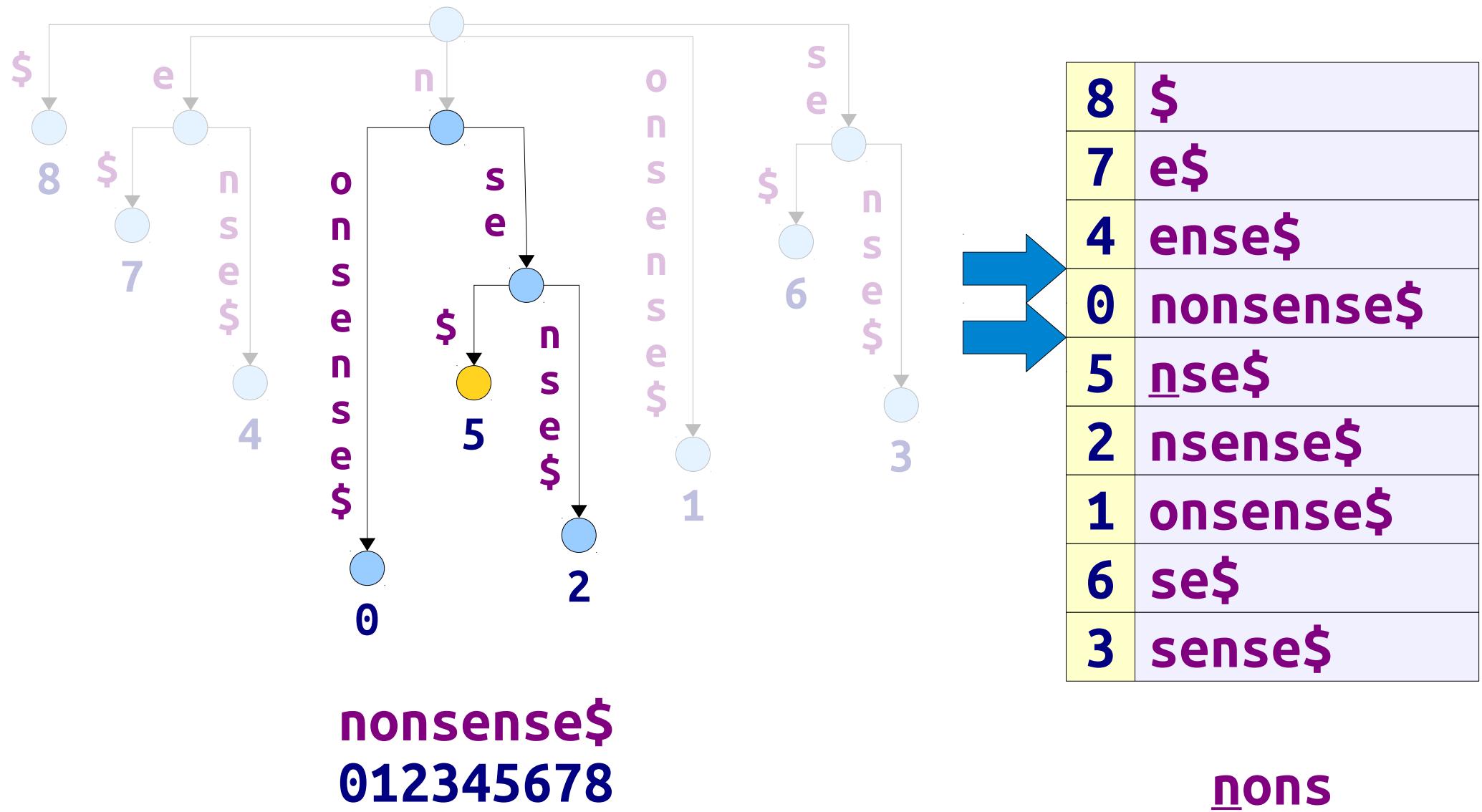
Searching a Suffix Array



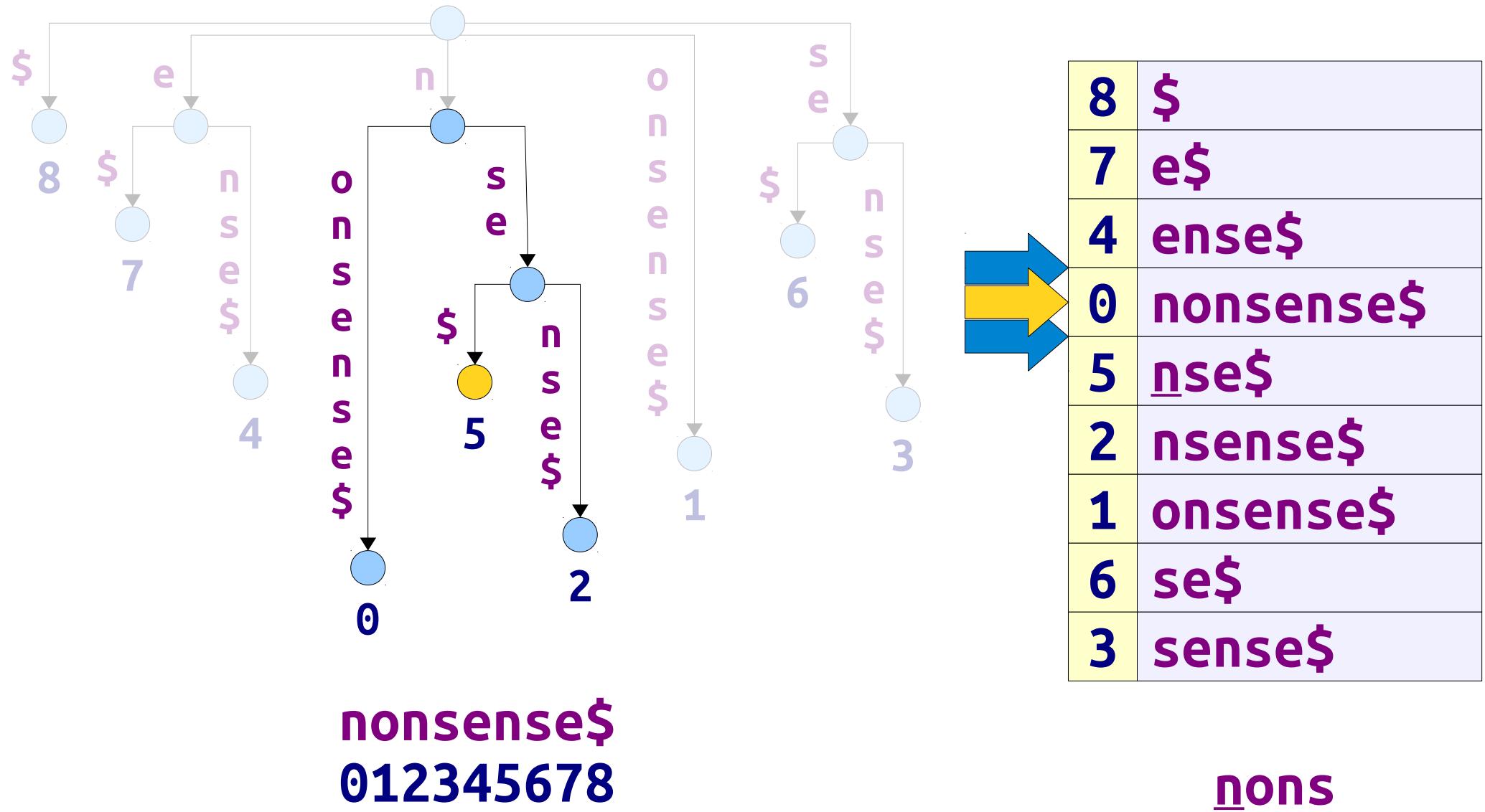
Searching a Suffix Array



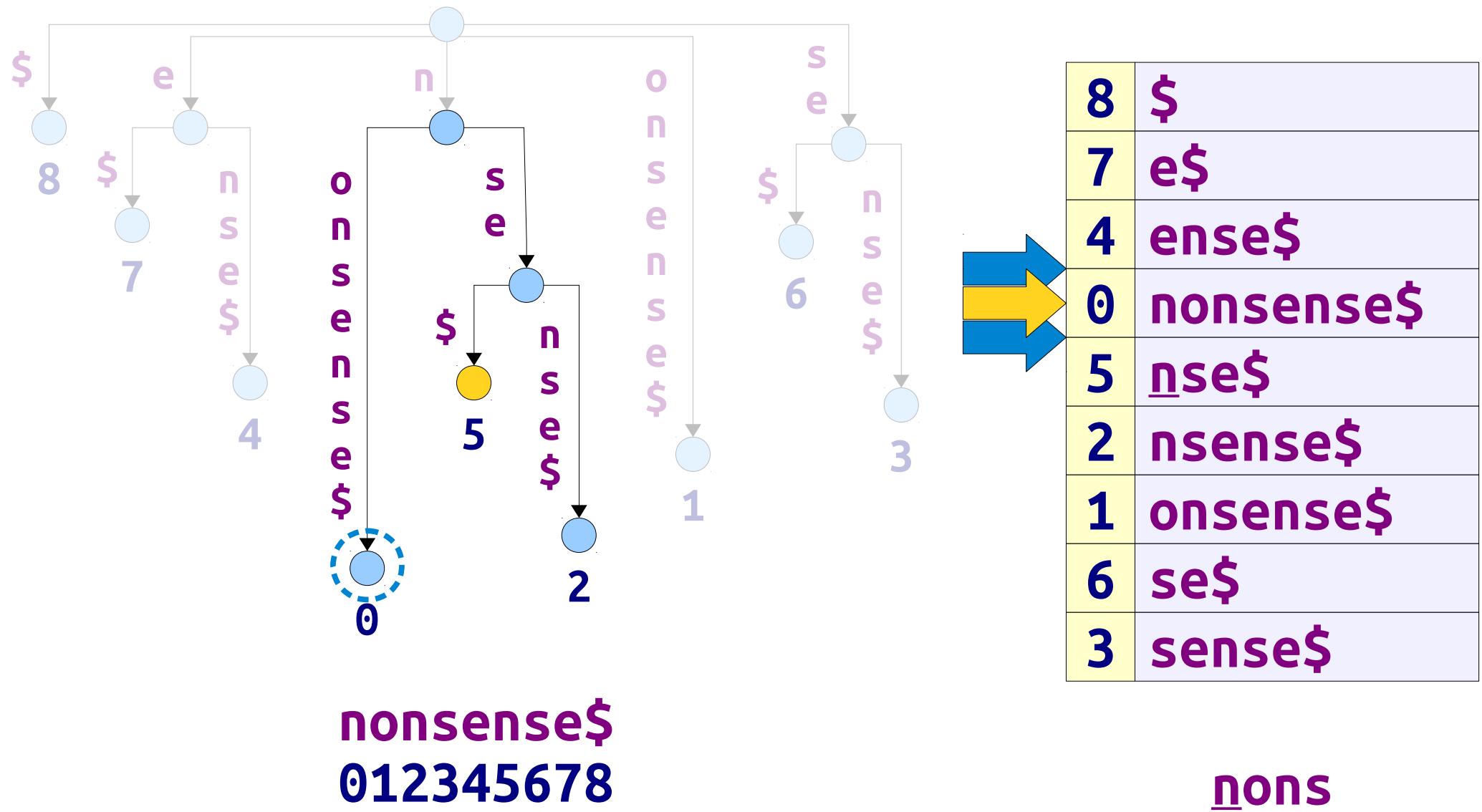
Searching a Suffix Array



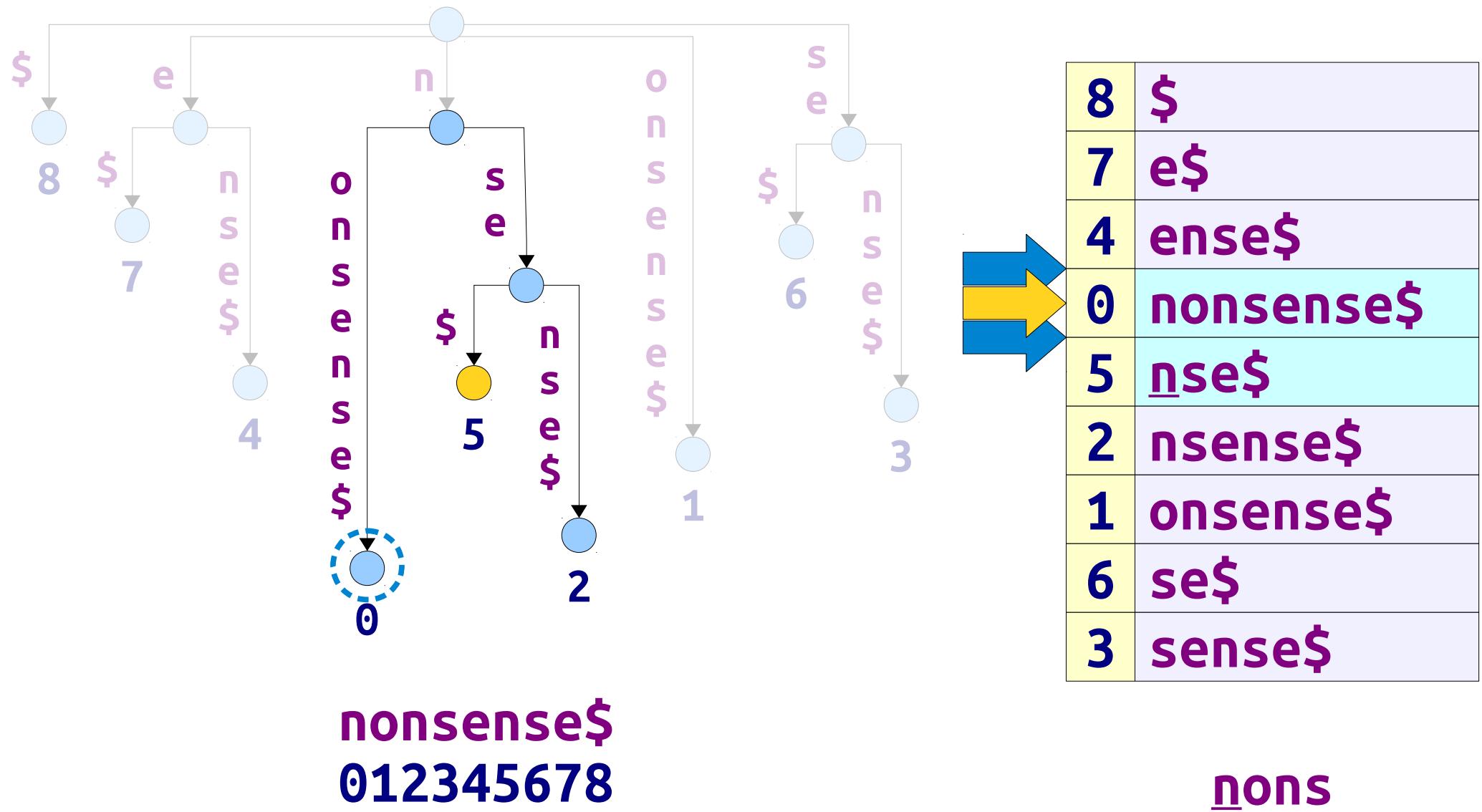
Searching a Suffix Array



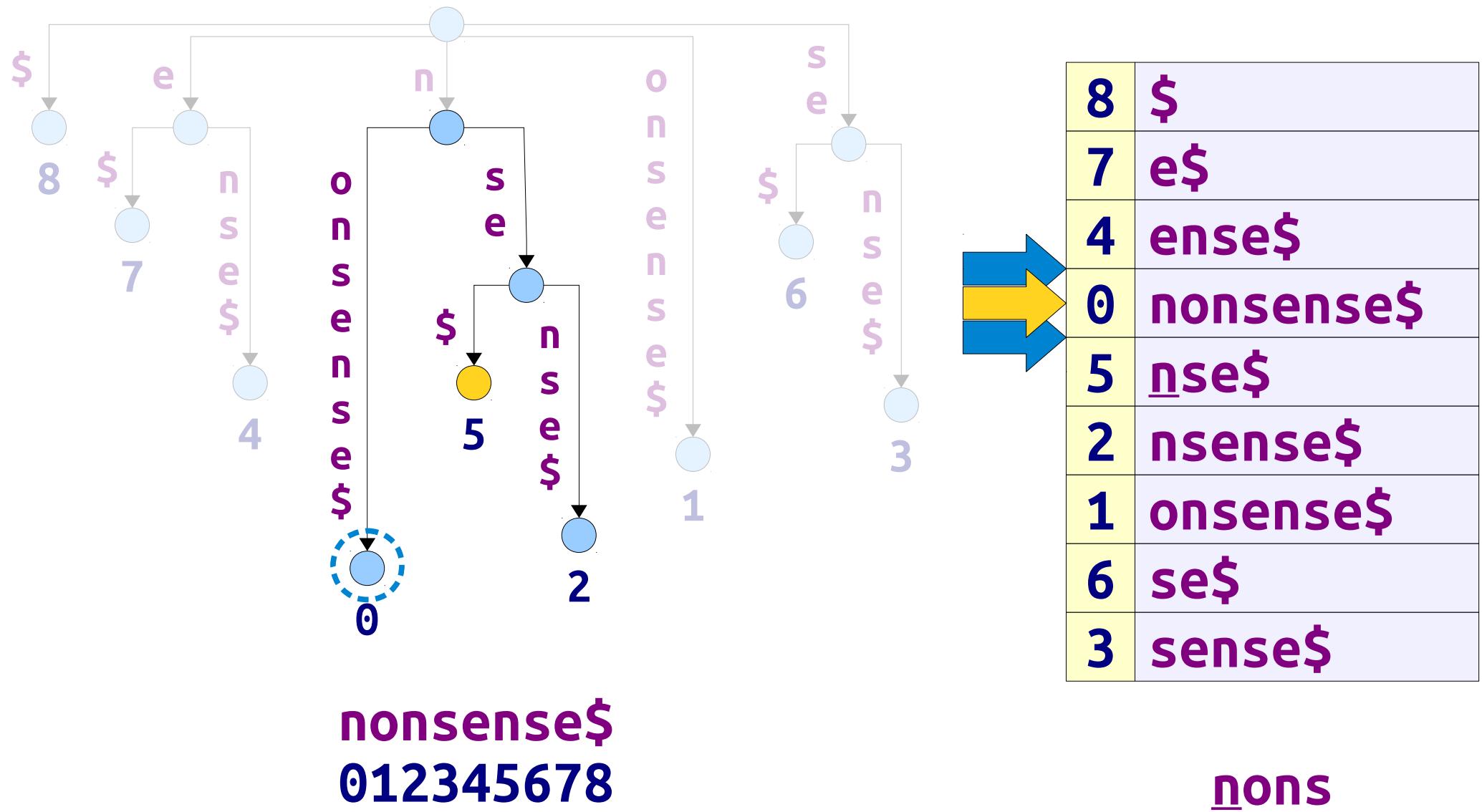
Searching a Suffix Array



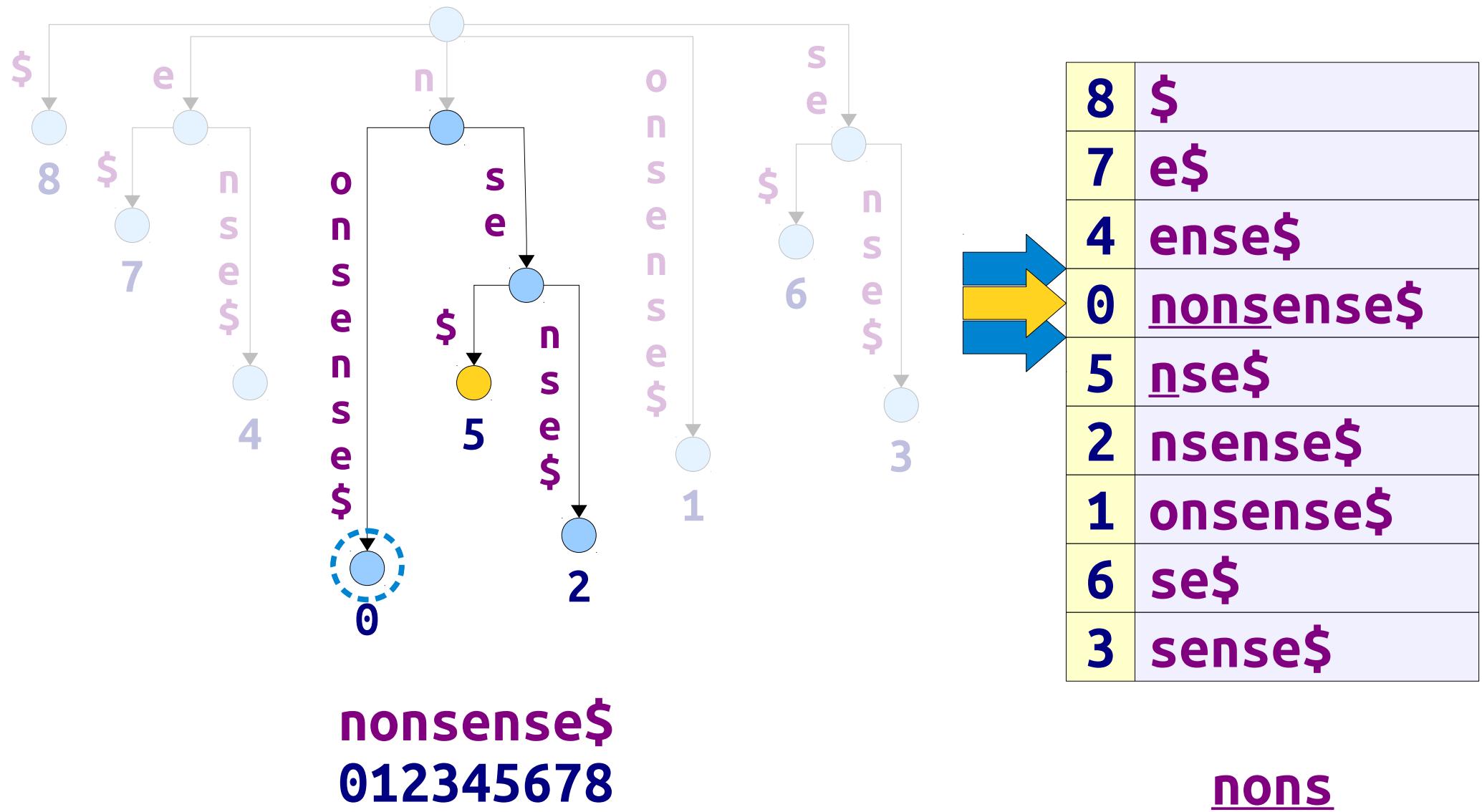
Searching a Suffix Array



Searching a Suffix Array



Searching a Suffix Array



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 $\text{LCE}_{T_1, T_2}(i, j)$, 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 $\langle O(m), O(1) \rangle$ -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	$nonsense\$_1$
1	1	5	$nse\$_1$
3	2	2	$nse\$_2$
3	1	2	$nsense\$_1$
0	1	1	$onsense\$_1$
0	1	6	$se\$_1$
2	2	3	$se\$_2$
2	1	3	$sense\$_1$
0	2	0	$tense\$_2$

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

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	$nonsense\$_1$
1	1	5	$nse\$_1$
3	2	2	$nse\$_2$
3	1	2	$nsense\$_1$
0	1	1	$onsense\$_1$
0	1	6	$se\$_1$
2	2	3	$se\$_2$
2	1	3	$sense\$_1$
0	2	0	$tense\$_2$

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

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	$nonsense\$_1$
1	1	5	$nse\$_1$
3	2	2	$nse\$_2$
3	1	2	$nsense\$_1$
0	1	1	$onsense\$_1$
0	1	6	$se\$_1$
2	2	3	$se\$_2$
2	1	3	$sense\$_1$
0	2	0	$tense\$_2$

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

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	$nonsense\$_1$
1	1	5	$nse\$_1$
3	2	2	$nse\$_2$
3	1	2	$nsense\$_1$
0	1	1	$onsense\$_1$
0	1	6	$se\$_1$
2	2	3	$se\$_2$
2	1	3	$sense\$_1$
0	2	0	$tense\$_2$

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

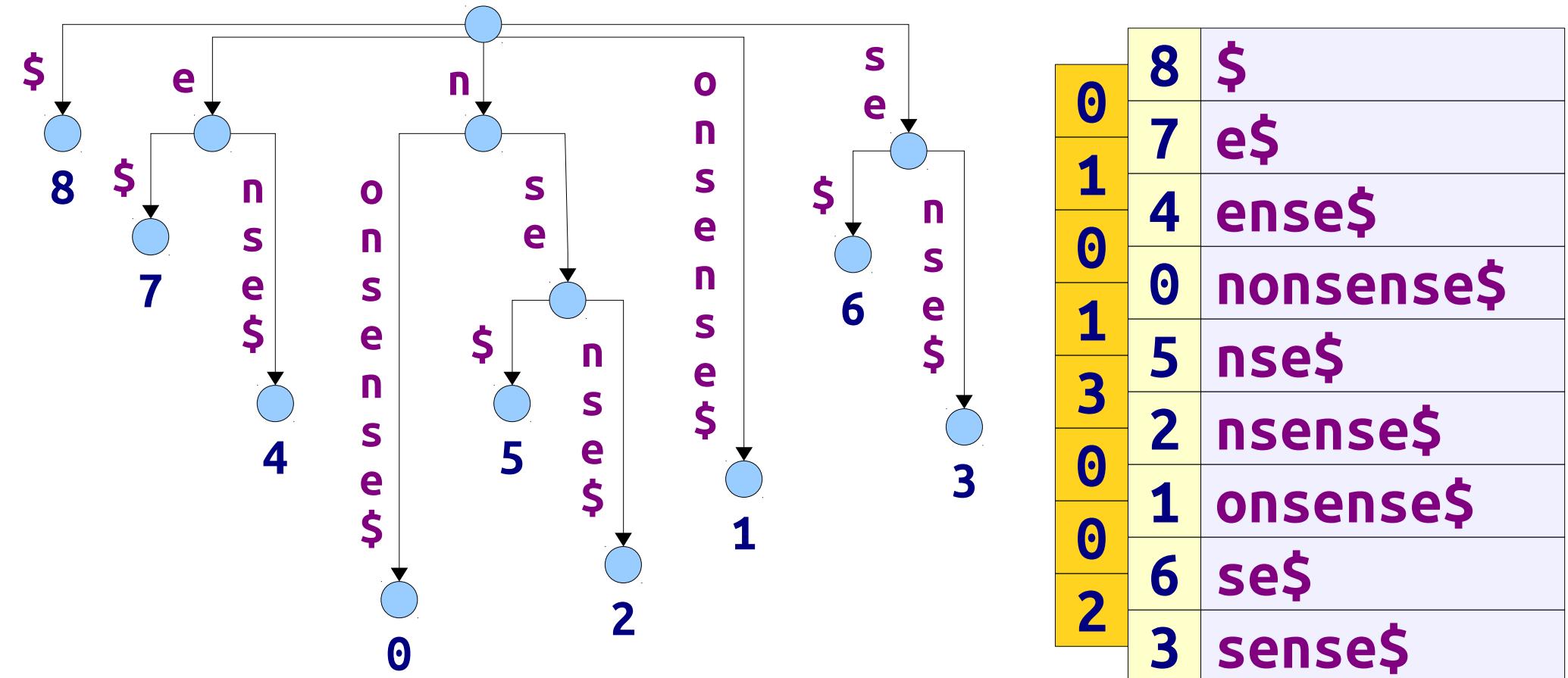
Using LCP: Constructing Suffix Trees

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 $O(m)$.

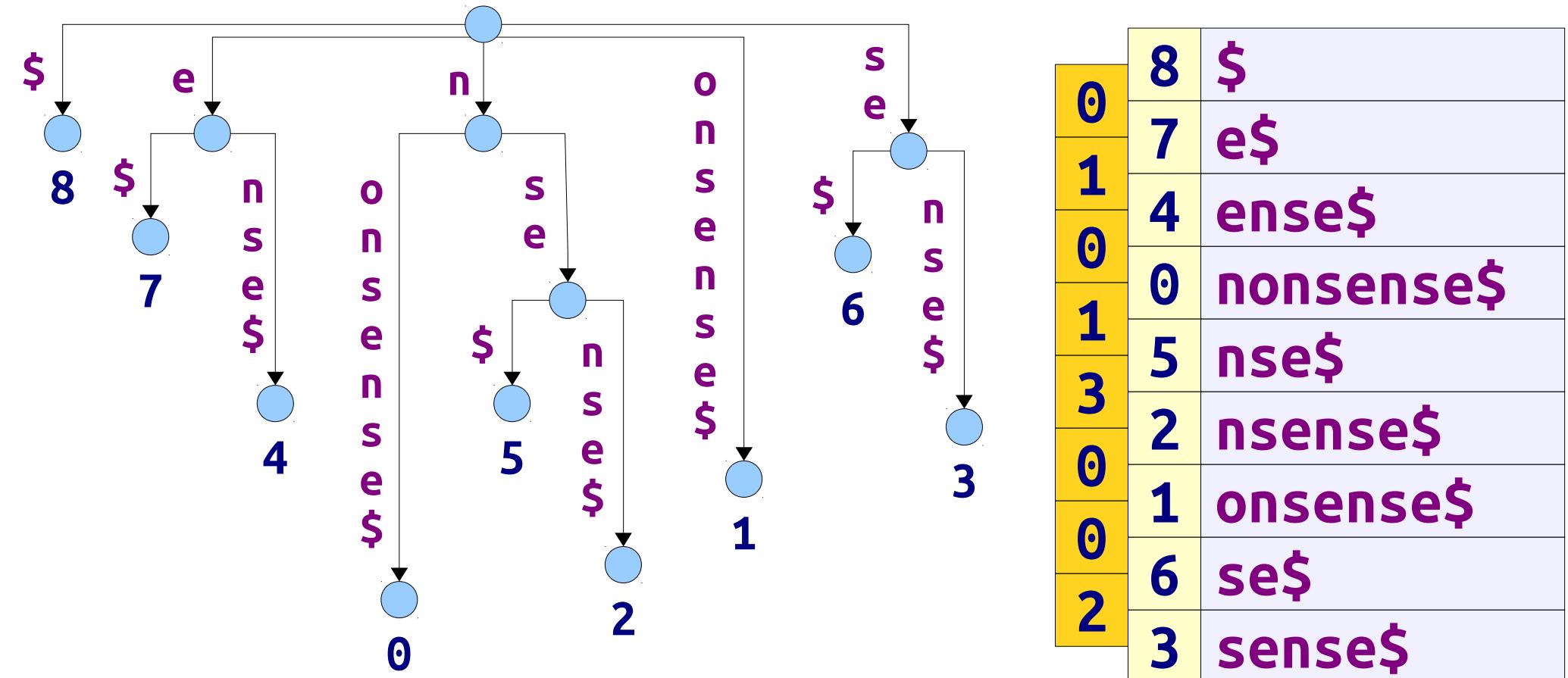
From Suffix Arrays to Suffix Trees

Using LCP



nonsense\$
012345678

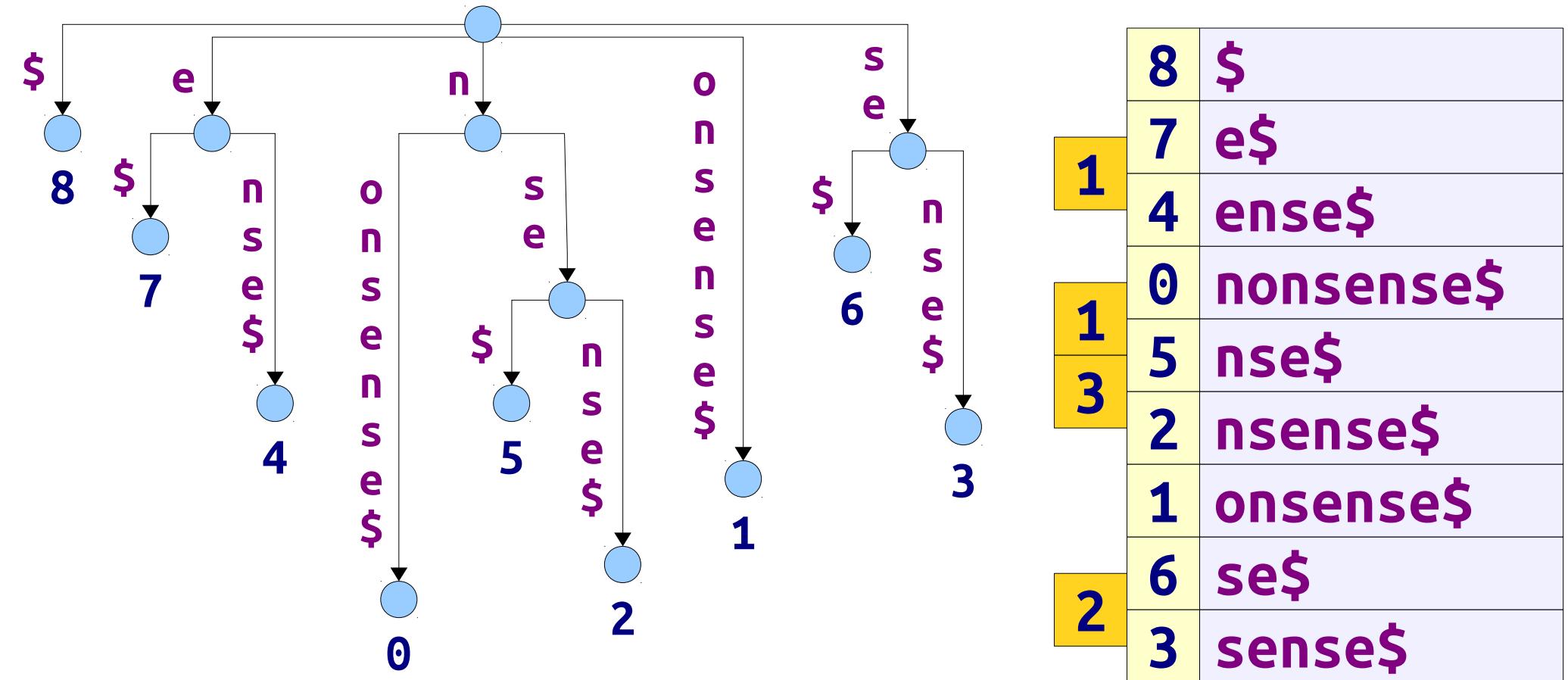
Using LCP



nonsense\$
012345678

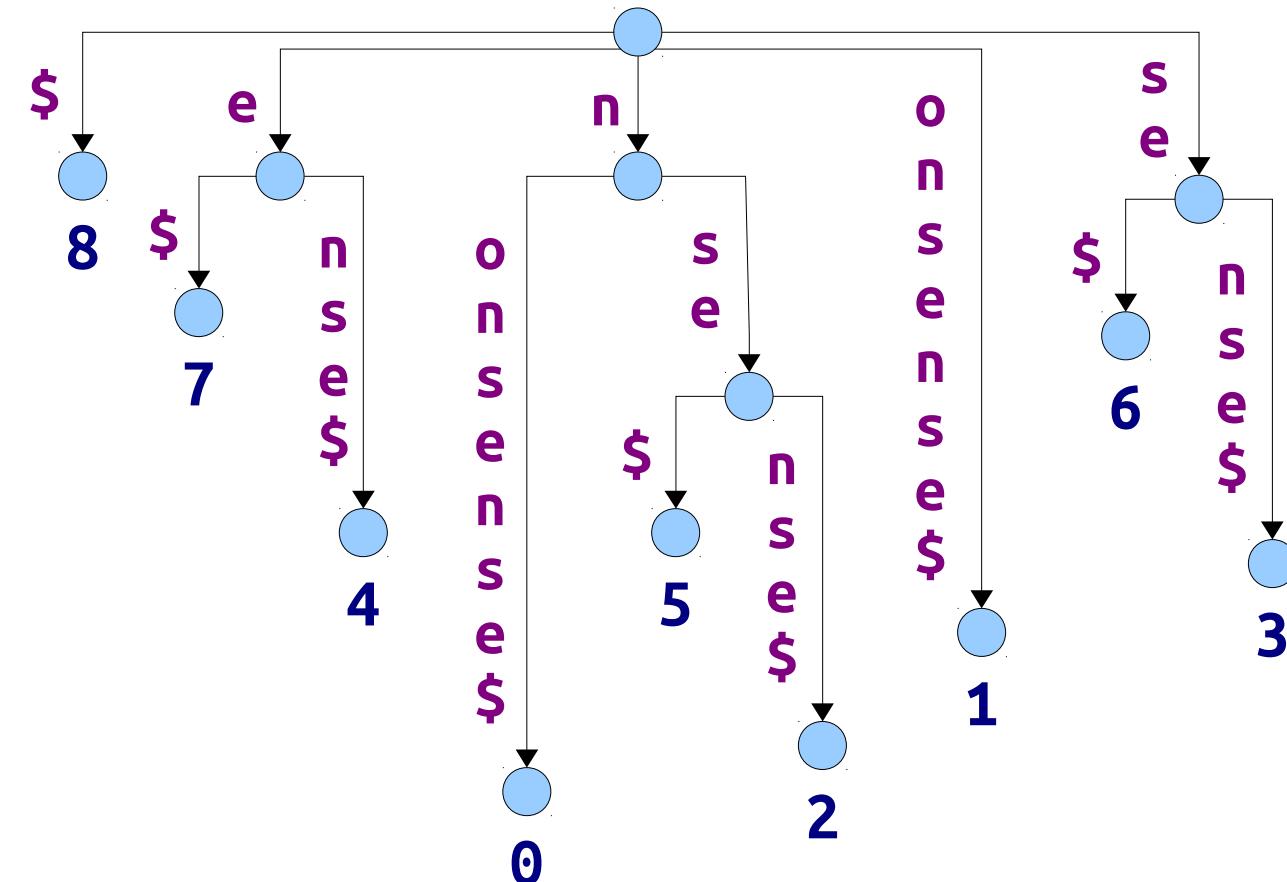
Claim: Any 0's in the suffix array represent demarcation points between subtrees of the root node.

Using LCP



nonsense\$
012345678

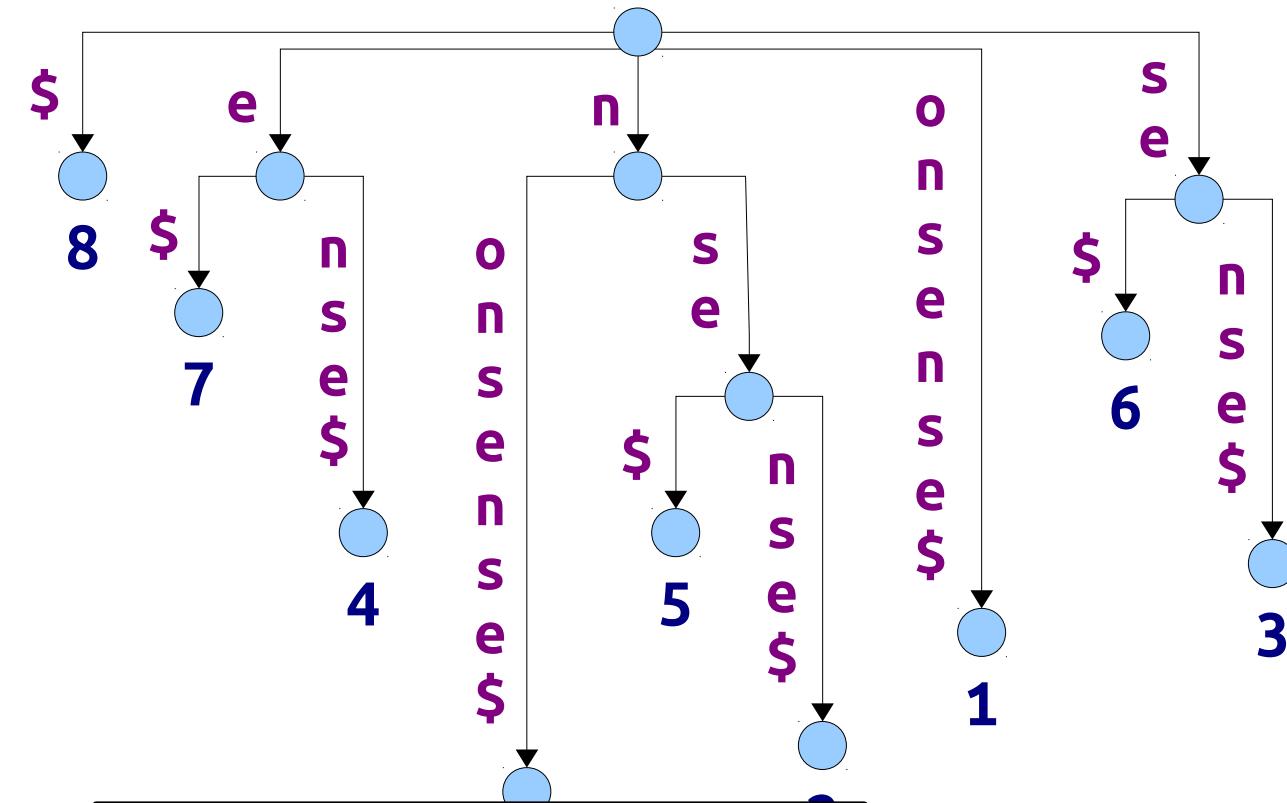
Using LCP



nonsense\$
012345678

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

Using LCP



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

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

0	\$
1	a\$
4	aaaba\$
2	aaabbabaaaaba\$
4	aaba\$
3	aabaaaabbabaaaaba\$
1	aabbabaaaaba\$
3	aba\$
6	abaaaaba\$
2	abaaaabbabaaaaba\$
0	abbabaaaaba\$
2	ba\$
5	baaaba\$
2	baaabbaabaaaaba\$
1	bababaaaaba\$
	bbabaaaaba\$

0

0

\$

a\$

1

aaaba\$

4

aaabbabaaaaba\$

2

aaba\$

4

aabaaaabbabaaaaba\$

3

aabbabaaaaba\$

1

aba\$

3

abaaaaba\$

6

abaaaabbabaaaaba\$

2

abbabaaaaba\$

ba\$

2

baaaba\$

5

baaabbbabaaaaba\$

2

bababaaaaba\$

1

bbabaaaaba\$

0
0

	\$
1	a\$
4	aaaba\$
2	aaabbabaaaaba\$
4	aaba\$
3	aabaaabbabaaaaba\$
1	aabbabaaaaba\$
3	aba\$
6	abaaaba\$
2	abaaabbabaaaaba\$
	abbabaaaaba\$
2	ba\$
5	baaababa\$
2	baaabbbabaaaaba\$
1	bababaaaaba\$
	bbabaaaaba\$

1

0
0

1

\$

a\$

aaaba\$

aaabbabaaaaba\$

aaba\$

aabaaaabbabaaaaba\$

aabbabaaaaba\$

aba\$

abaaaaba\$

abaaaabbabaaaaba\$

abbabaaaaba\$

ba\$

baaabaa\$

baaabbbabaaaaba\$

bababaaaaba\$

bbabaaaaba\$

4

2

4

3

3

6

2

2

5

2

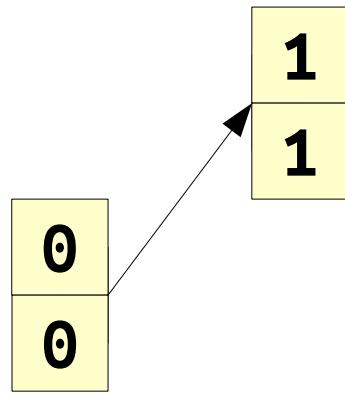
1

0
0

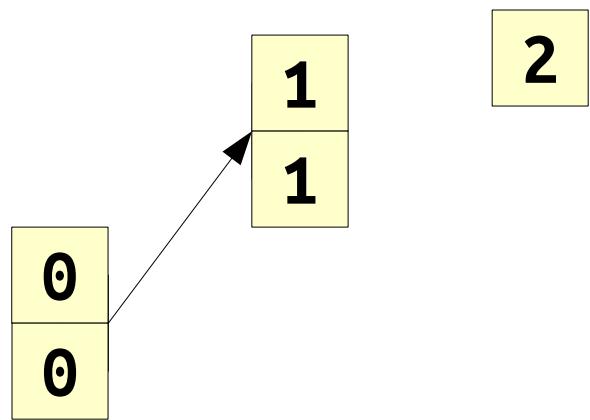
1
1

\$
a\$
aaaba\$
aaabbabaaaaba\$
aababa\$
aabaaaabbabaaaaba\$
aabbabaaaaba\$
aba\$
abaaaaba\$
abaaaabbabaaaaba\$
abbabaaaaba\$
ba\$
baaaba\$
baaabbabaaaaba\$
bababaaaaba\$
bbabaaaaba\$

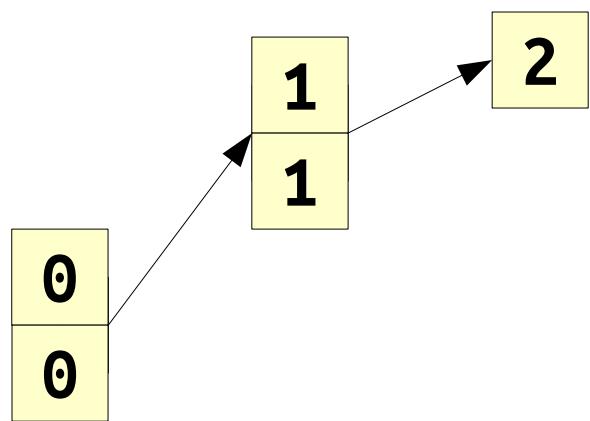
4
2
4
3
3
6
2
2
5
2
1



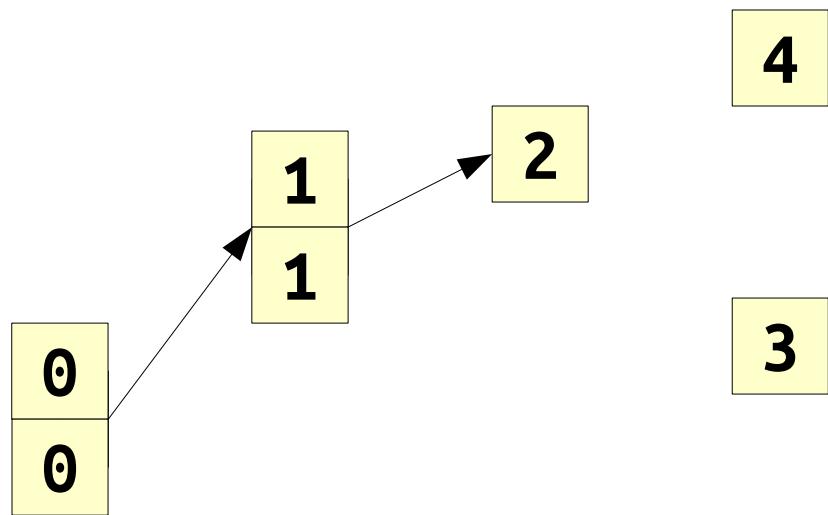
	\$
4	a\$
2	aaaba\$
4	aaabbabaaaaba\$
3	aaba\$
3	aabaaaabbabaaaaba\$
6	aabbabaaaaba\$
2	aba\$
3	abaaaaba\$
6	abaaaabbabaaaaba\$
2	abbabaaaaba\$
2	ba\$
5	baaaba\$
2	baaabbaaaaaba\$
1	bababaaaaba\$
	bbabaaaaba\$



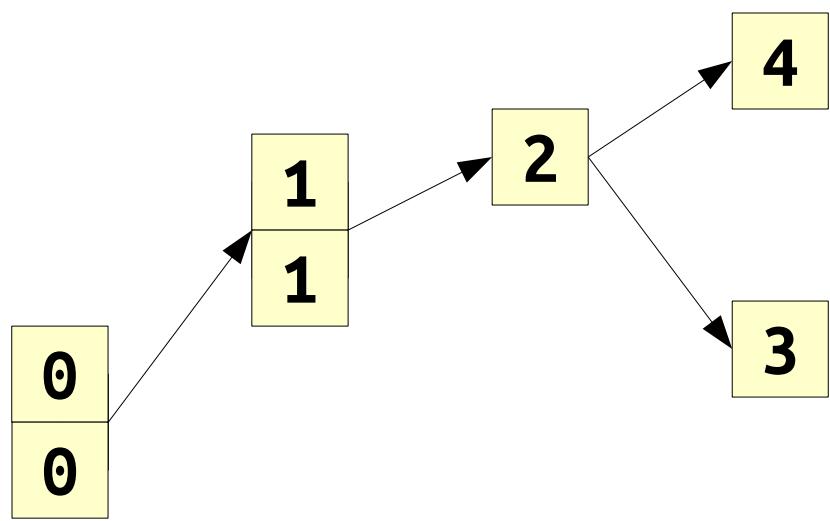
4	\$
3	a\$
4	aaaba\$
2	aaabbabaaaaba\$
3	aaba\$
3	aabaaaabbabaaaaba\$
2	aabbabaaaaba\$
3	aba\$
6	abaaaaba\$
2	abaaaabbabaaaaba\$
2	abbabaaaaba\$
2	ba\$
5	baaaba\$
2	baaabbbabaaaaba\$
1	bababaaaaba\$
	bbabaaaaba\$



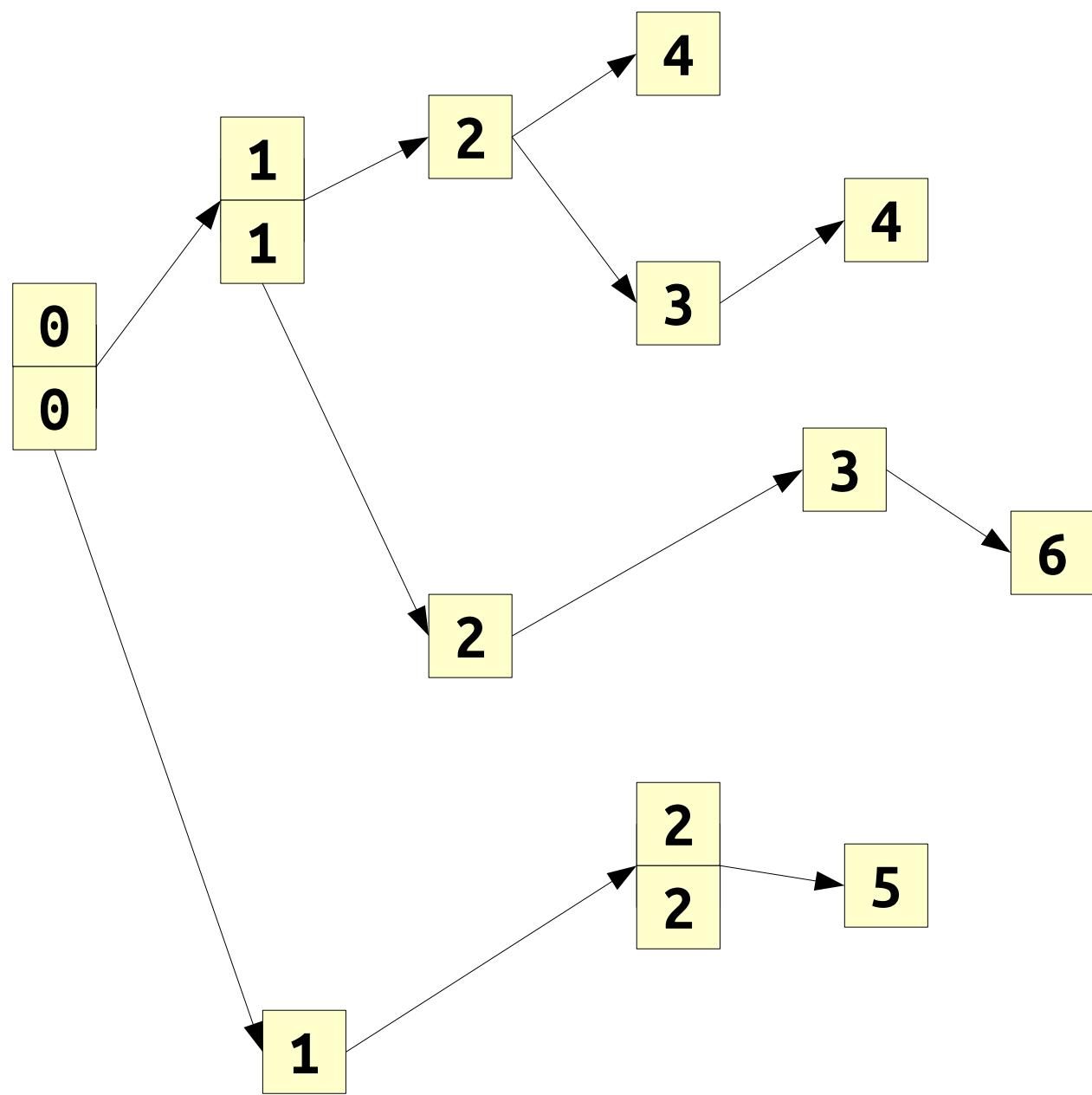
	\$
	a\$
4	aaaba\$
	aaabbabaaaaba\$
4	aaba\$
3	aabaaaabbabaaaaba\$
	aabbabaaaaba\$
3	aba\$
6	abaaaaba\$
2	abaaaabbabaaaaba\$
	abbabaaaaba\$
2	ba\$
5	baaababa\$
2	baaabbabaaaaba\$
1	bababaaaaba\$
	bbabaaaaba\$



	\$
	a\$
	aaaba\$
	aaabbabaaaaba\$
	aaba\$
	aabaaaabbabaaaaba\$
	aabbabaaaaba\$
	aba\$
	abaaaaba\$
3	abaaaabbabaaaaba\$
6	abbabaaaaba\$
2	ba\$
2	baaabaa\$
5	baaababbabaaaaba\$
2	bababaaaaba\$
1	bbbabaaaaba\$

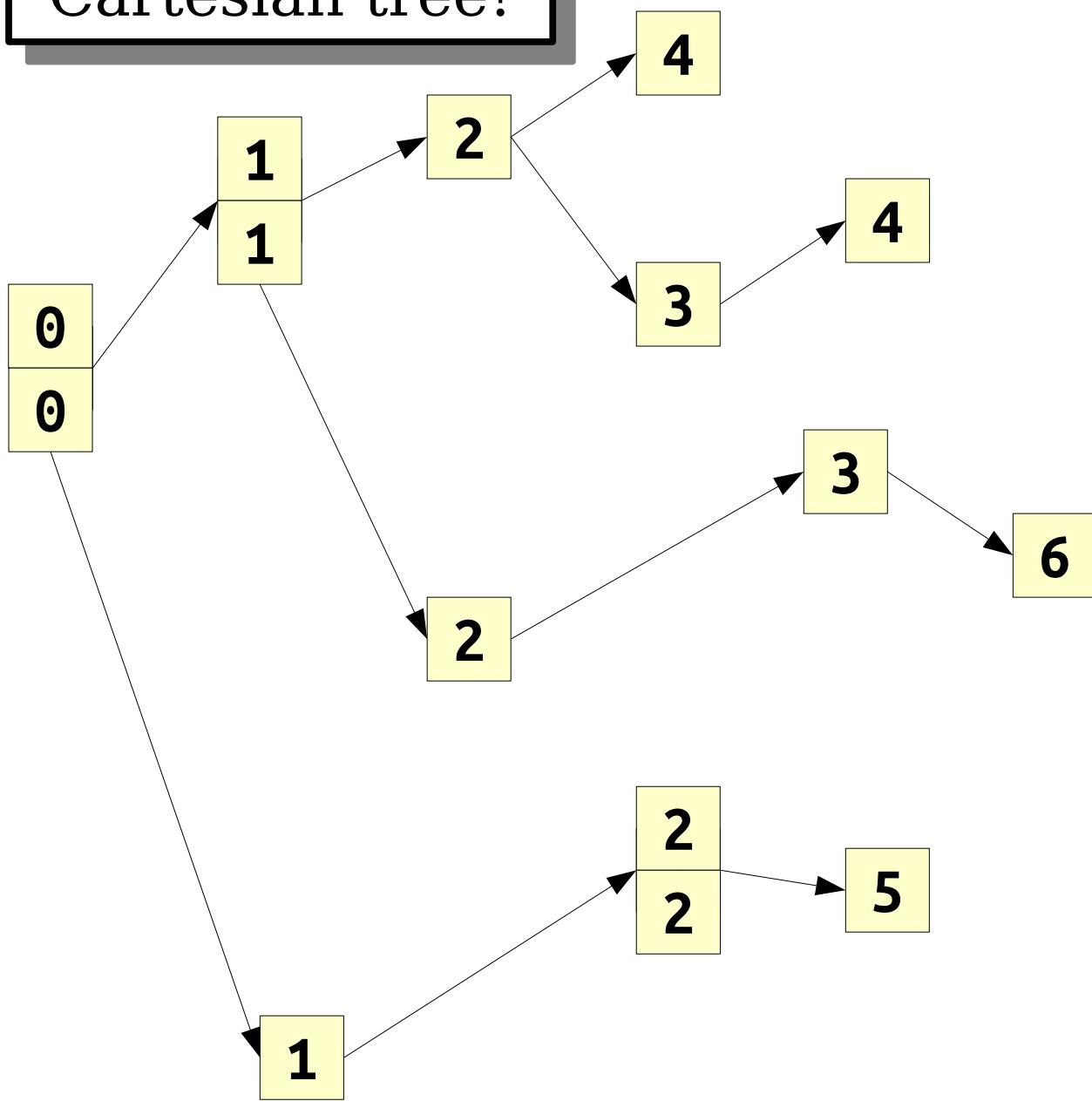


	\$
	a\$
	aaaba\$
	aaabbabaaaaba\$
	aaba\$
4	aabaaaabbabaaaaba\$
	aabbabaaaaba\$
	aba\$
3	abaaaaba\$
6	abaaaabbabaaaaba\$
2	abbabaaaaba\$
	ba\$
2	baaabaa\$
5	baaababbabaaaaba\$
2	bababaaaaba\$
1	bbabaaaaba\$

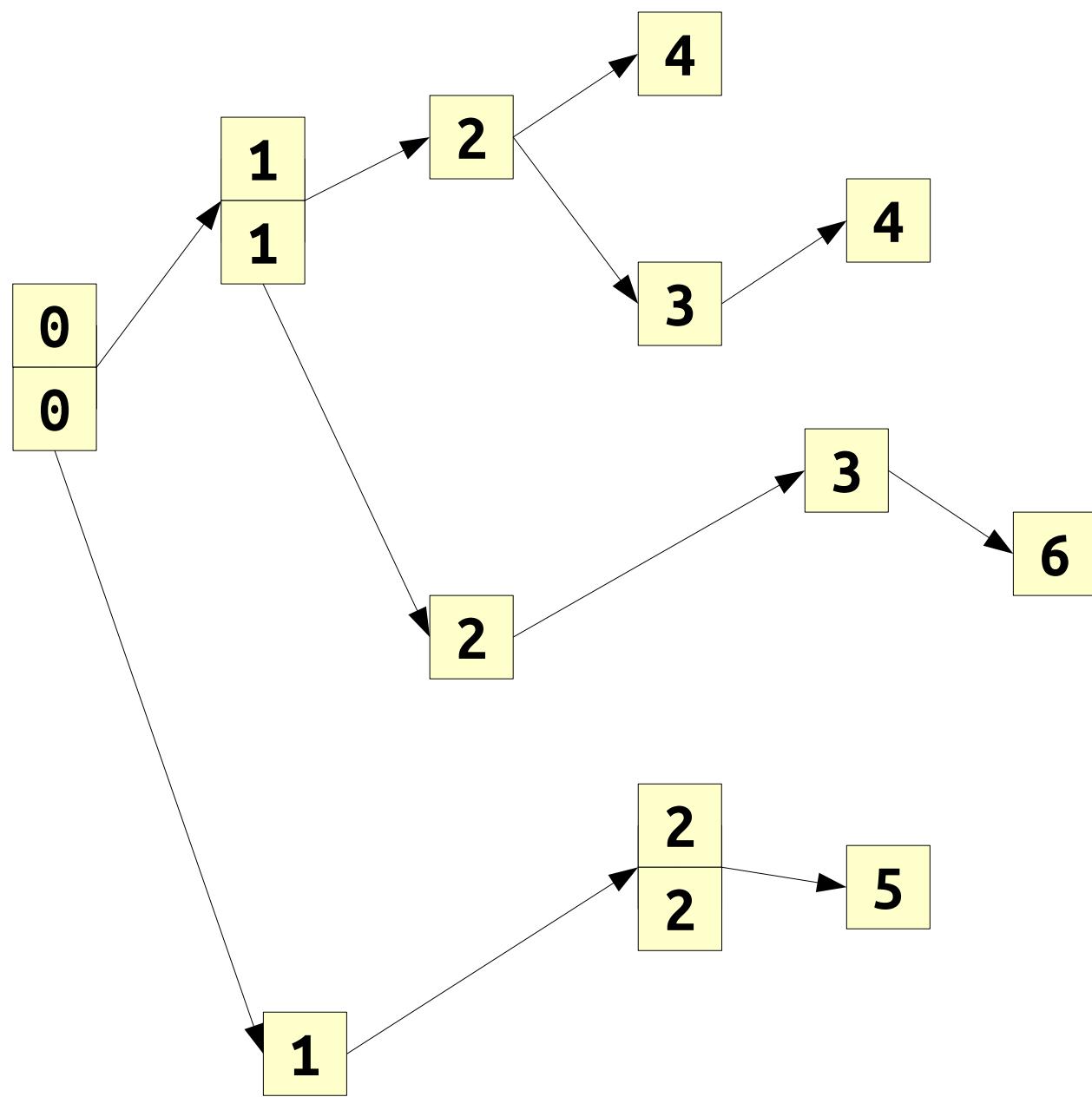


\$
a\$
aaaba\$
aaabbabaaaaba\$
aababa\$
aabaaaabbabaaaaba\$
aabbabaaaaba\$
aba\$
abaaaaba\$
abaaaabbabaaaaba\$
abbabaaaaba\$
ba\$
baaaba\$
baaabbabaaaaba\$
bababaaaaba\$
bbabaaaaba\$

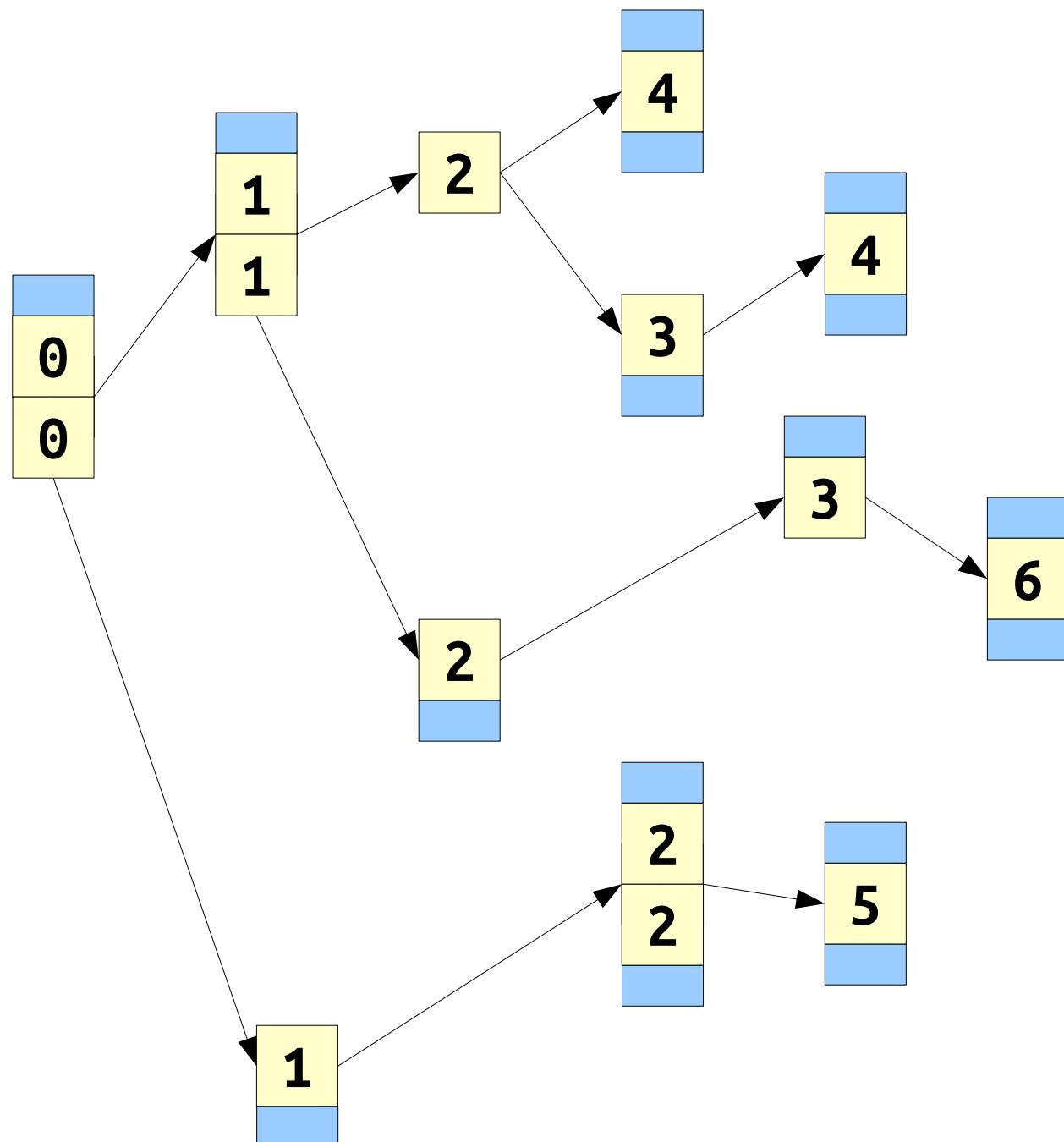
This is a slightly modified Cartesian tree!



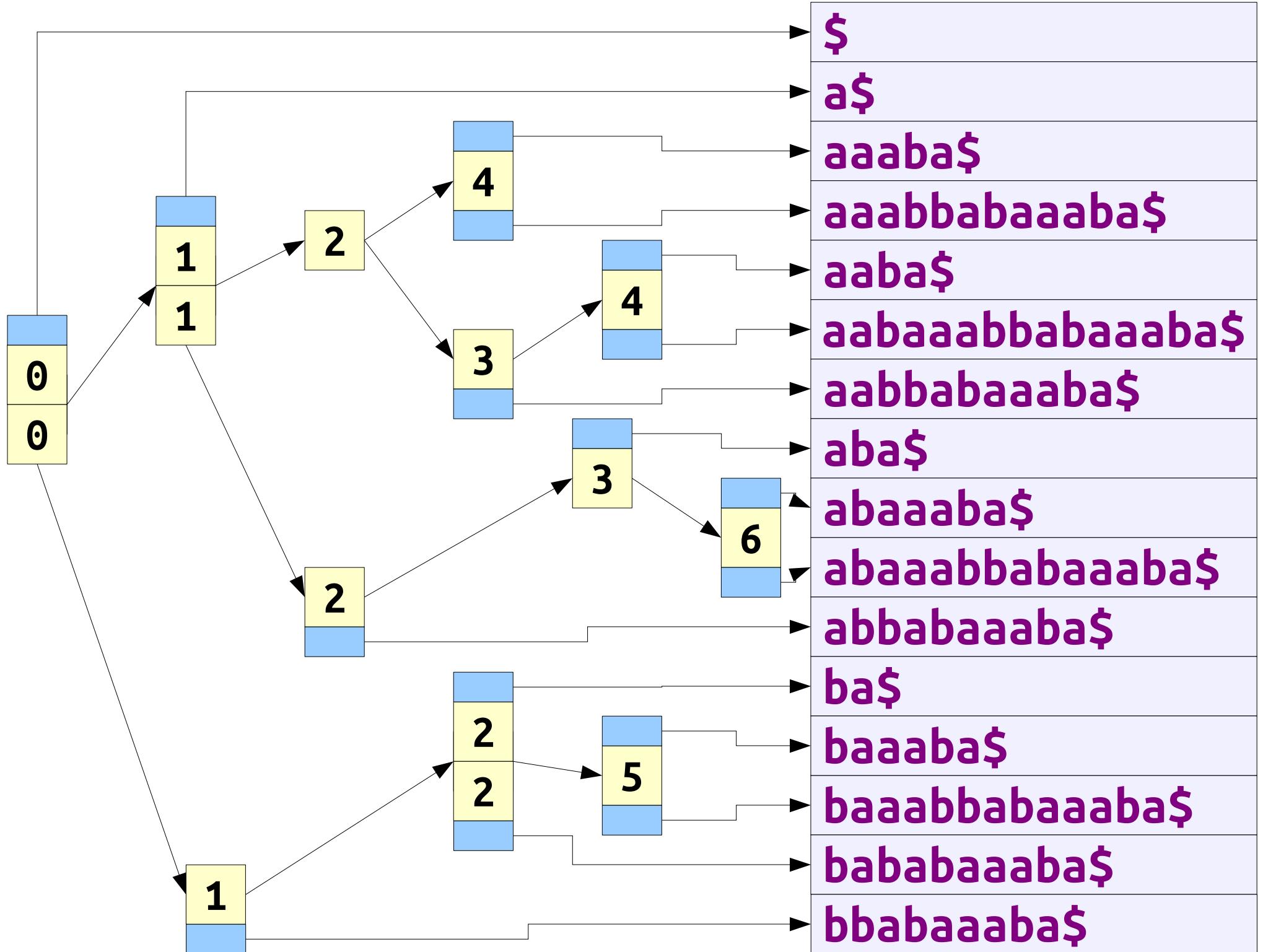
\$
a\$
aaaba\$
aaabbabaaaaba\$
aababa\$
aabaaaabbabaaaaba\$
aabbabaaaaba\$
aba\$
abaaaaba\$
abaaaabbabaaaaba\$
abbabaaaaba\$
ba\$
baaaba\$
baaabbabaaaaba\$
bababaaaaba\$
bbabaaaaba\$



\$
a\$
aaaba\$
aaabbabaaaaba\$
aababa\$
aabaaaabbabaaaaba\$
aabbabaaaaba\$
aba\$
abaaaaba\$
abaaaabbabaaaaba\$
abbabaaaaba\$
ba\$
baaaba\$
baaabbabaaaaba\$
bababaaaaba\$
bbabaaaaba\$



\$
a\$
aaaba\$
aaabbabaaaaba\$
aaba\$
aabaaaabbabaaaaba\$
aabbabaaaaba\$
aba\$
abaaaaba\$
abaaaabbabaaaaba\$
abbabaaaaba\$
ba\$
baaabaa\$
baaabbabaaaaba\$
bababaaaaba\$
bbabaaaaba\$



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)**.

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: **$O(m^2 \log m)$** .
- ***Can we do better?***

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: **$O(m^2 \log m)$** .

Can we do better?

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: **$O(m^2 \log m)$** .

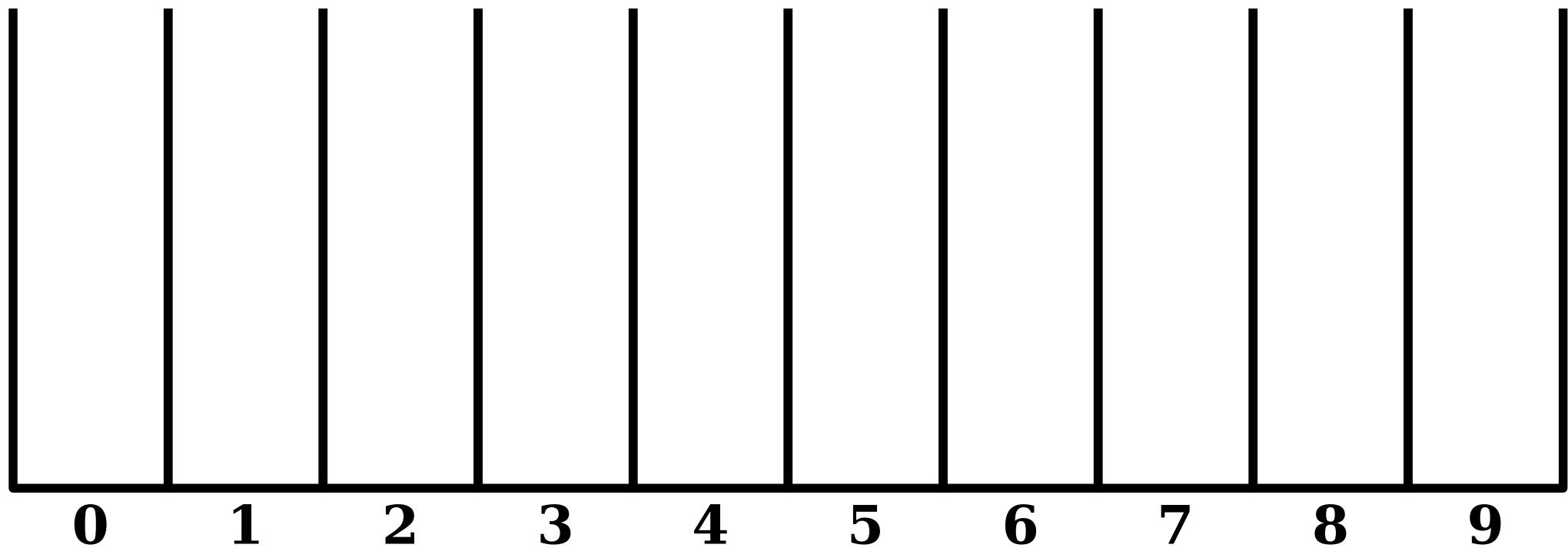
Is there a better sorting algorithm we could use here?

Can we do better?

Radix Sort

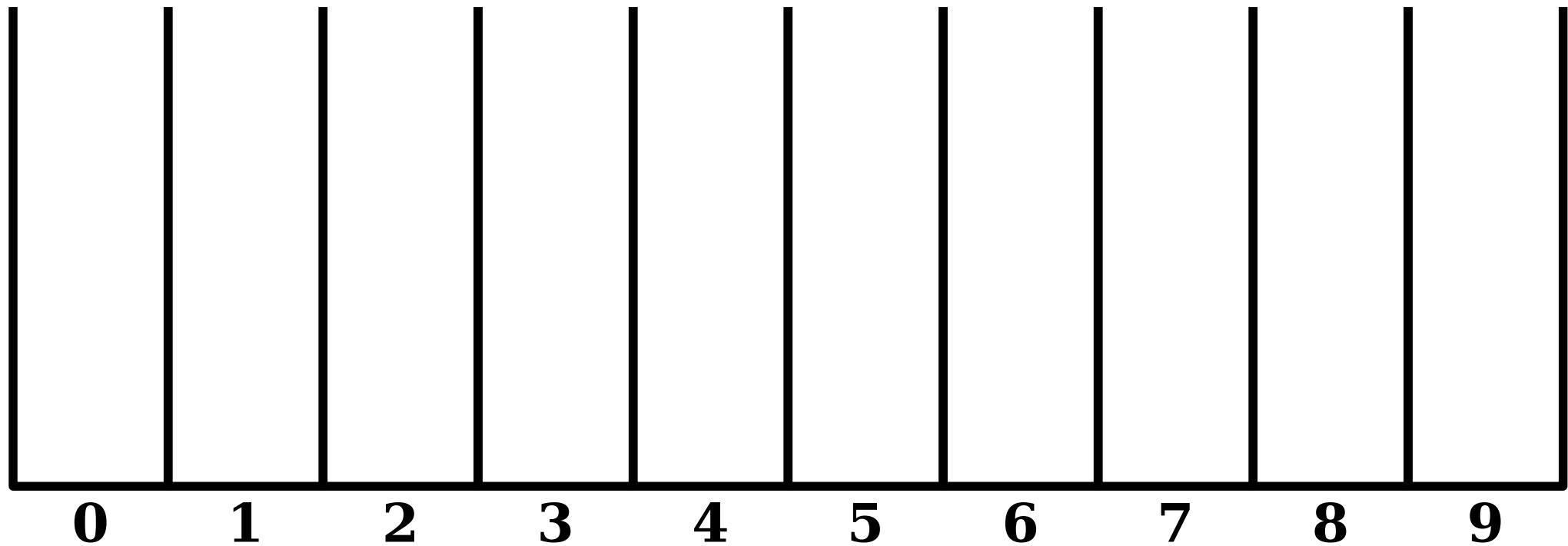
- ***Radix sort*** is a fast sorting algorithm for strings and integers.

Radix Sort



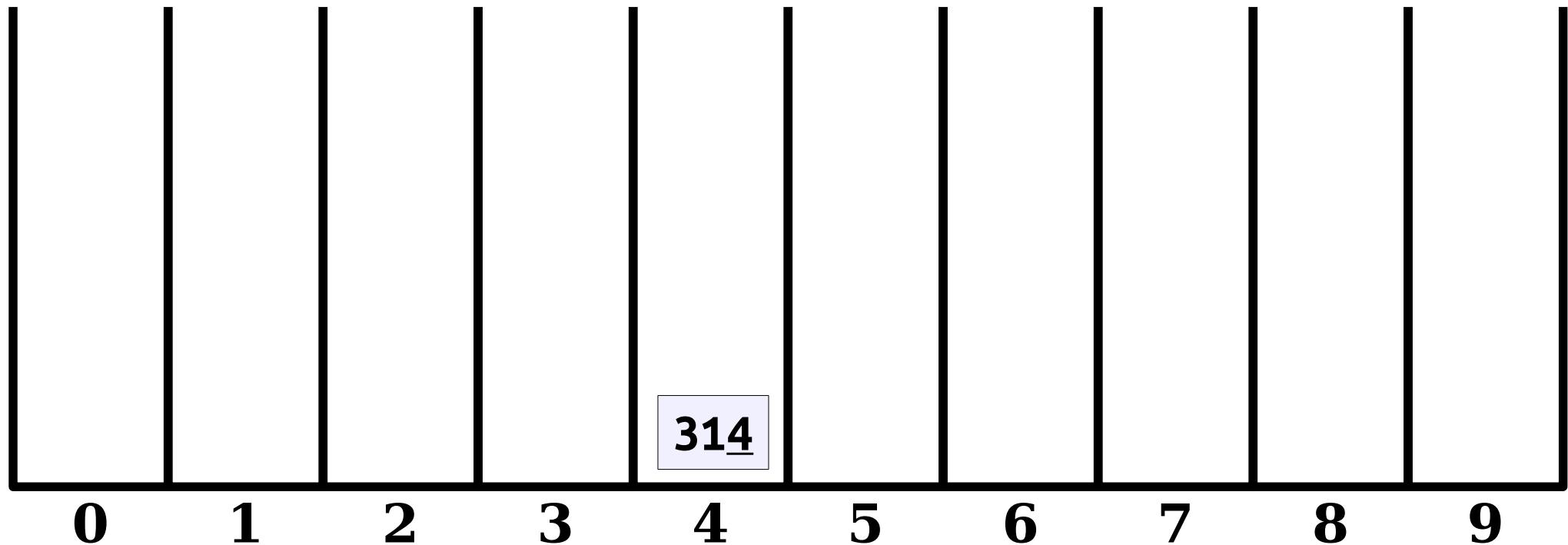
314	159	265	358	979	323	846	264	338
-----	-----	-----	-----	-----	-----	-----	-----	-----

Radix Sort



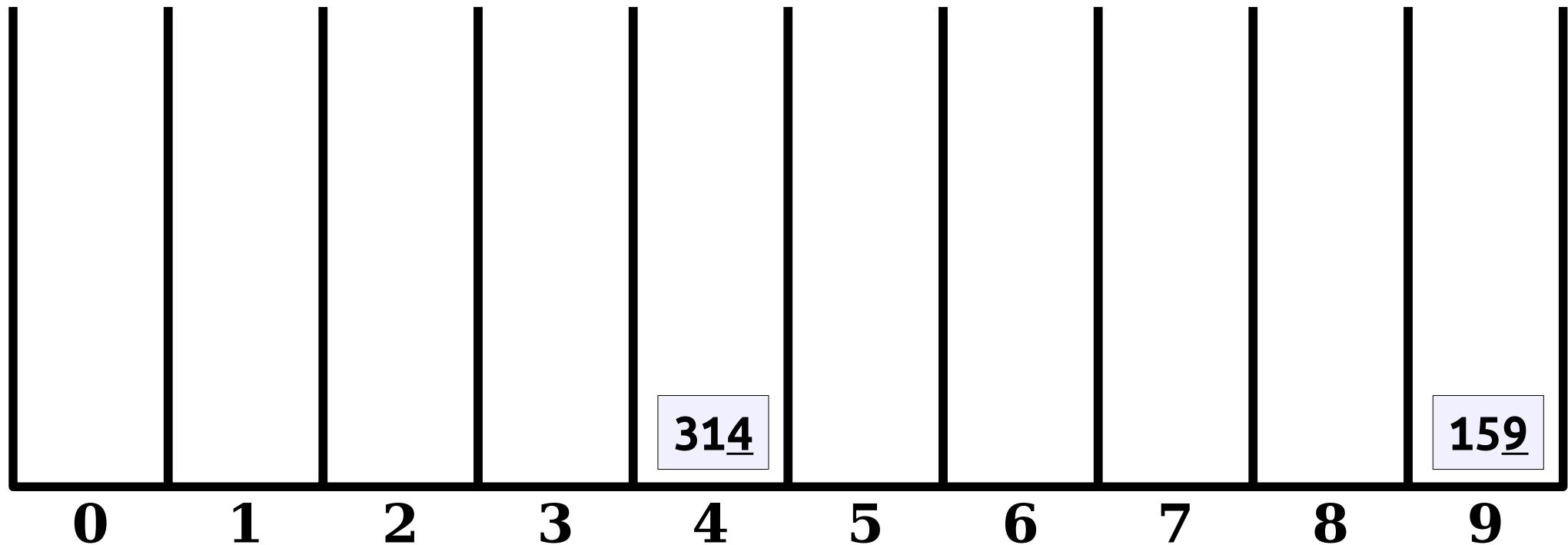
314 159 265 358 979 323 846 264 338

Radix Sort



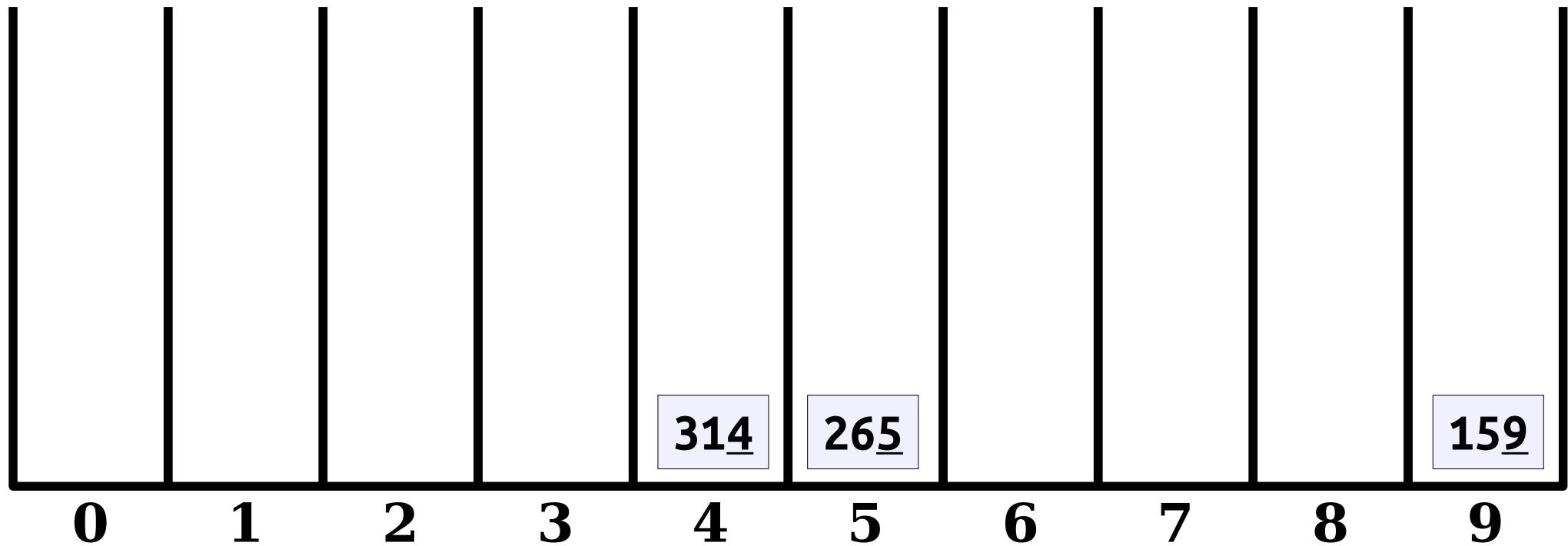
15<u>9</u>	26<u>5</u>	35<u>8</u>	97<u>9</u>	32<u>3</u>	84<u>6</u>	26<u>4</u>	33<u>8</u>
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

Radix Sort



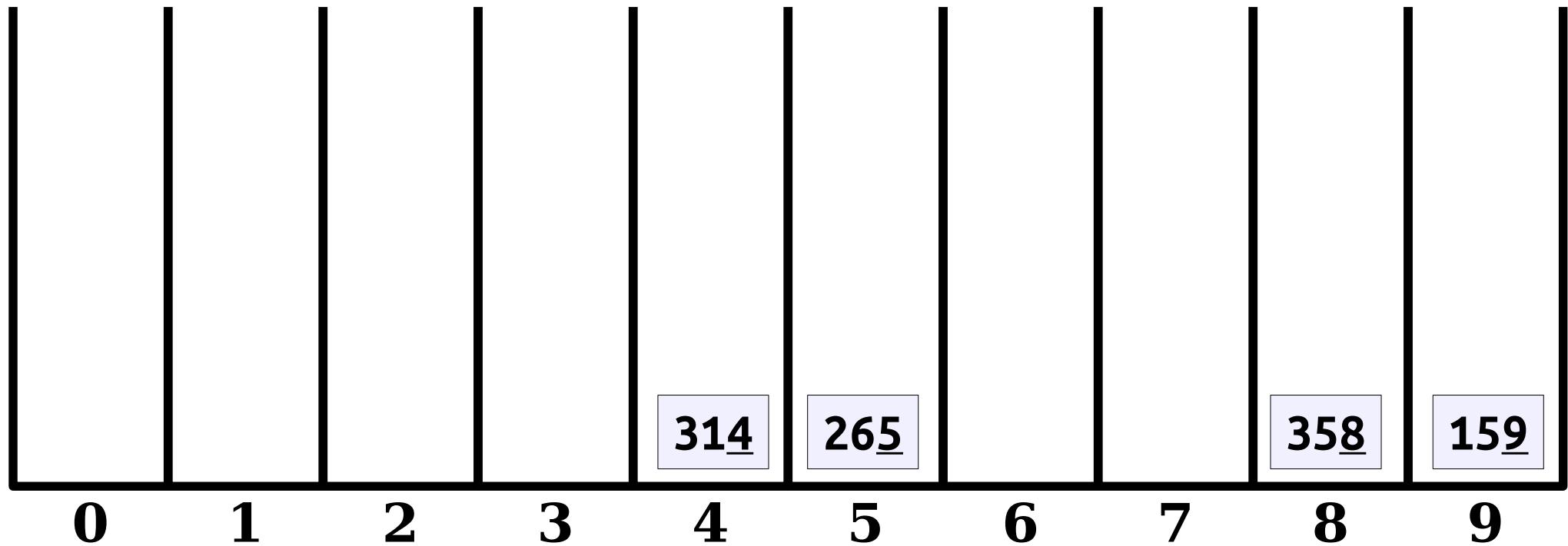
<u>265</u>	<u>358</u>	<u>979</u>	<u>323</u>	<u>846</u>	<u>264</u>	<u>338</u>
------------	------------	------------	------------	------------	------------	------------

Radix Sort



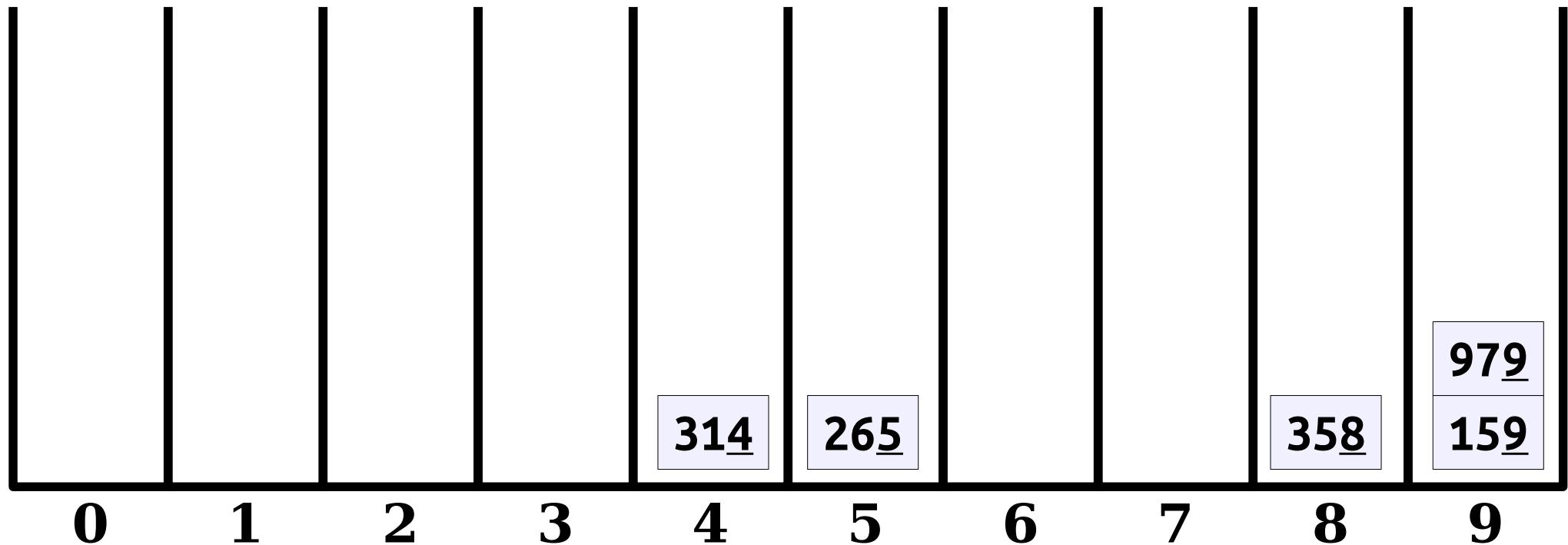
358 979 323 846 264 338

Radix Sort



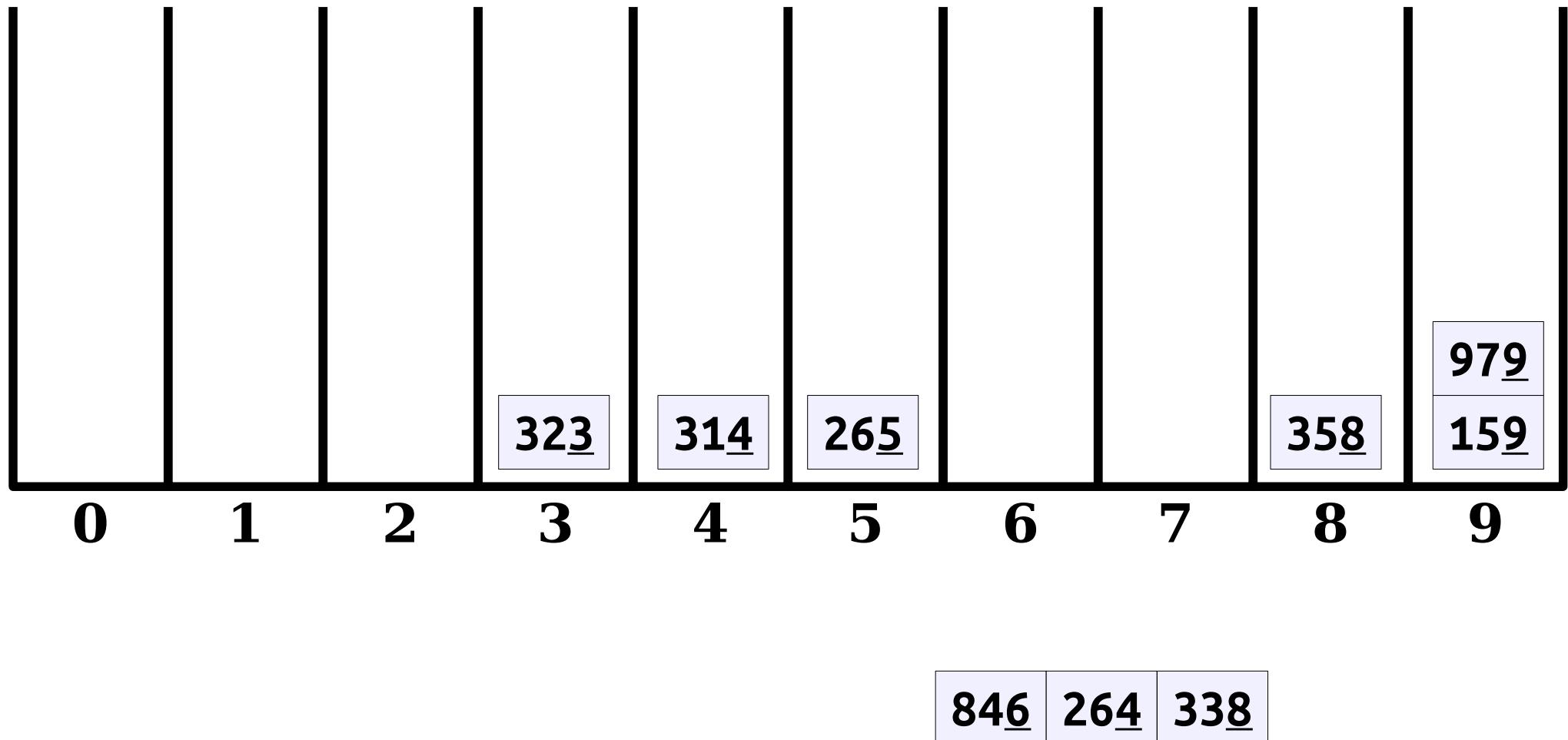
979 323 846 264 338

Radix Sort

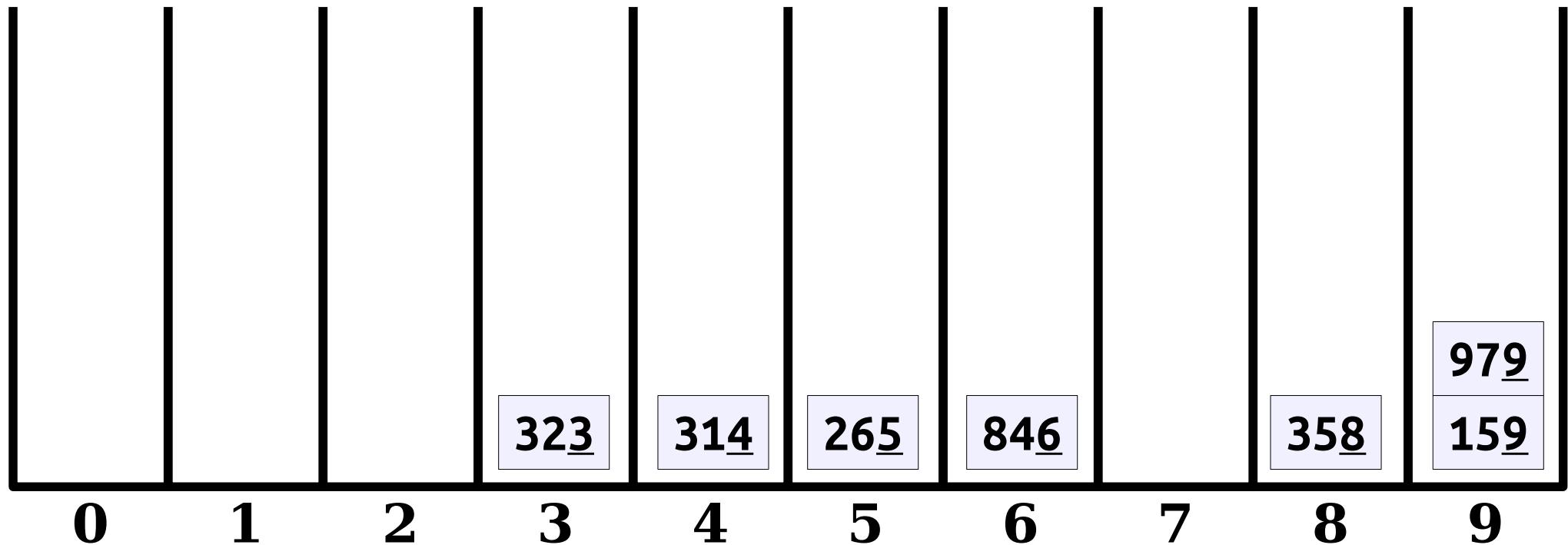


323 846 264 338

Radix Sort

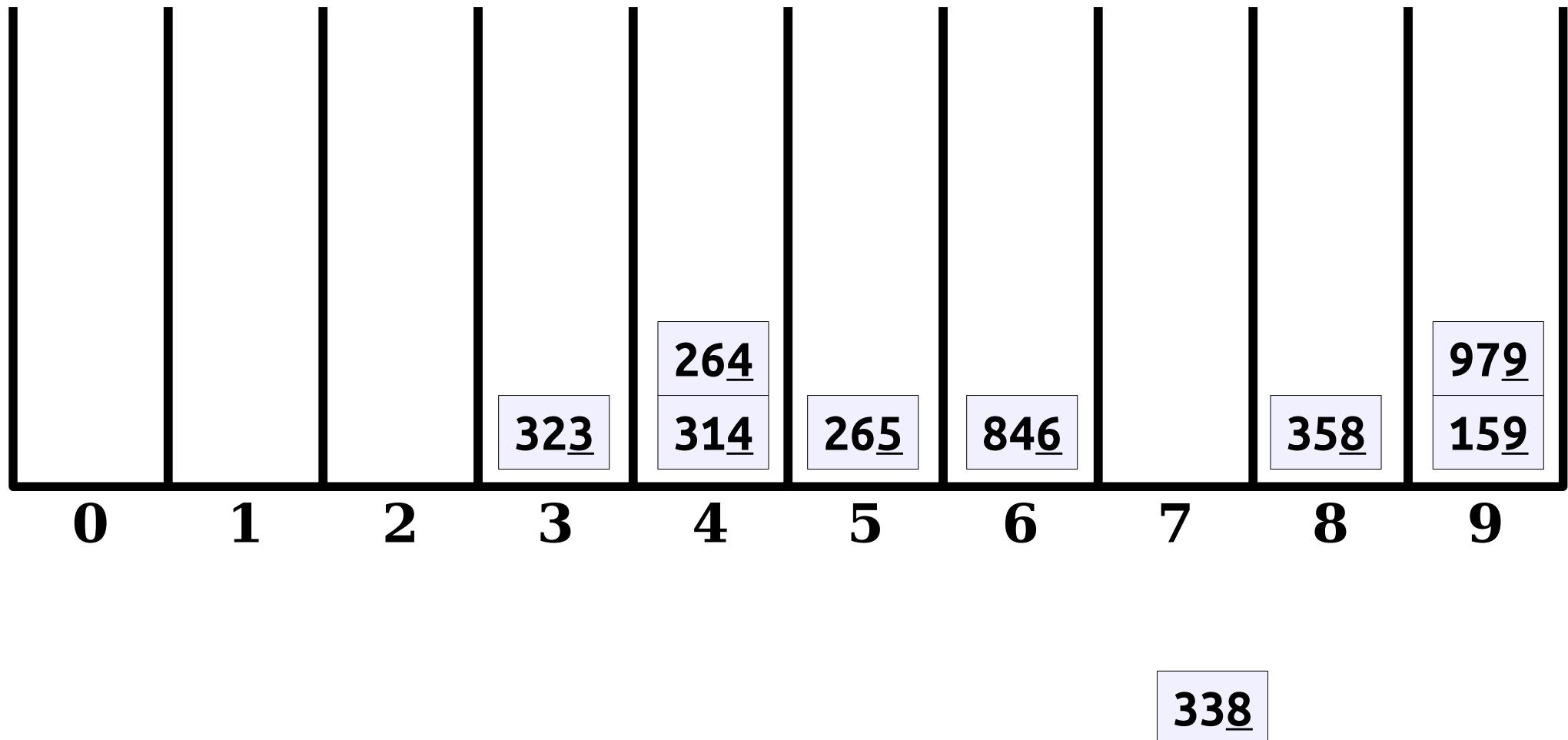


Radix Sort

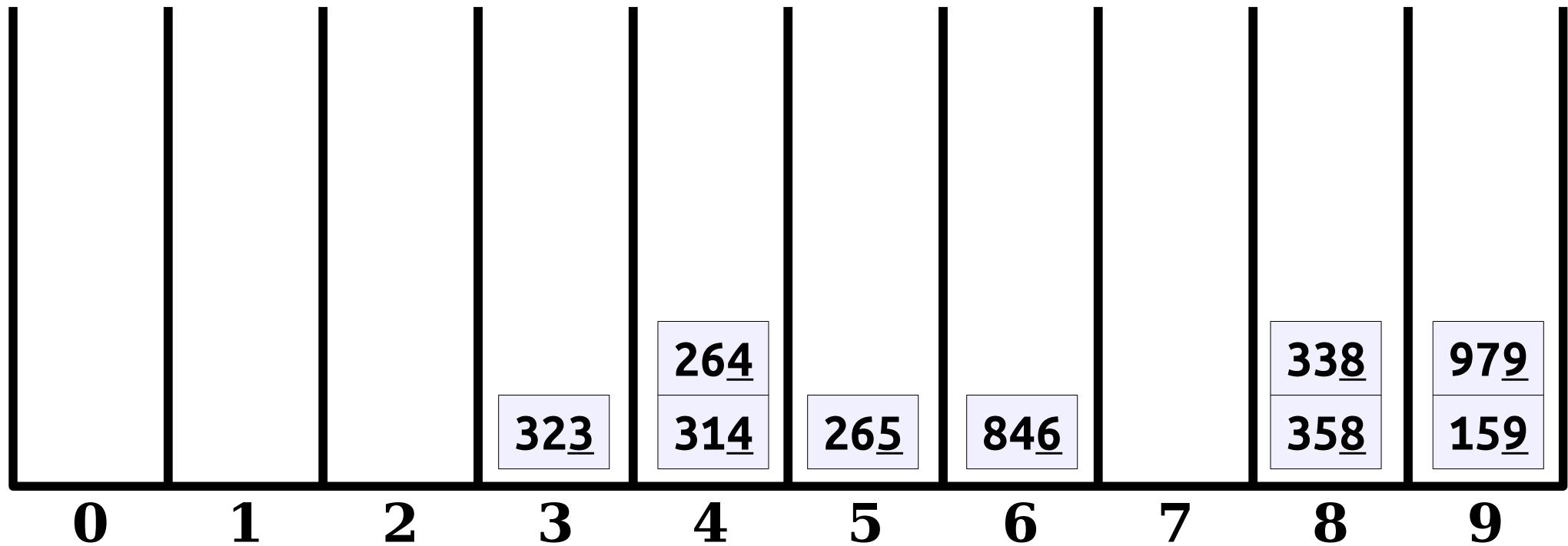


264 338

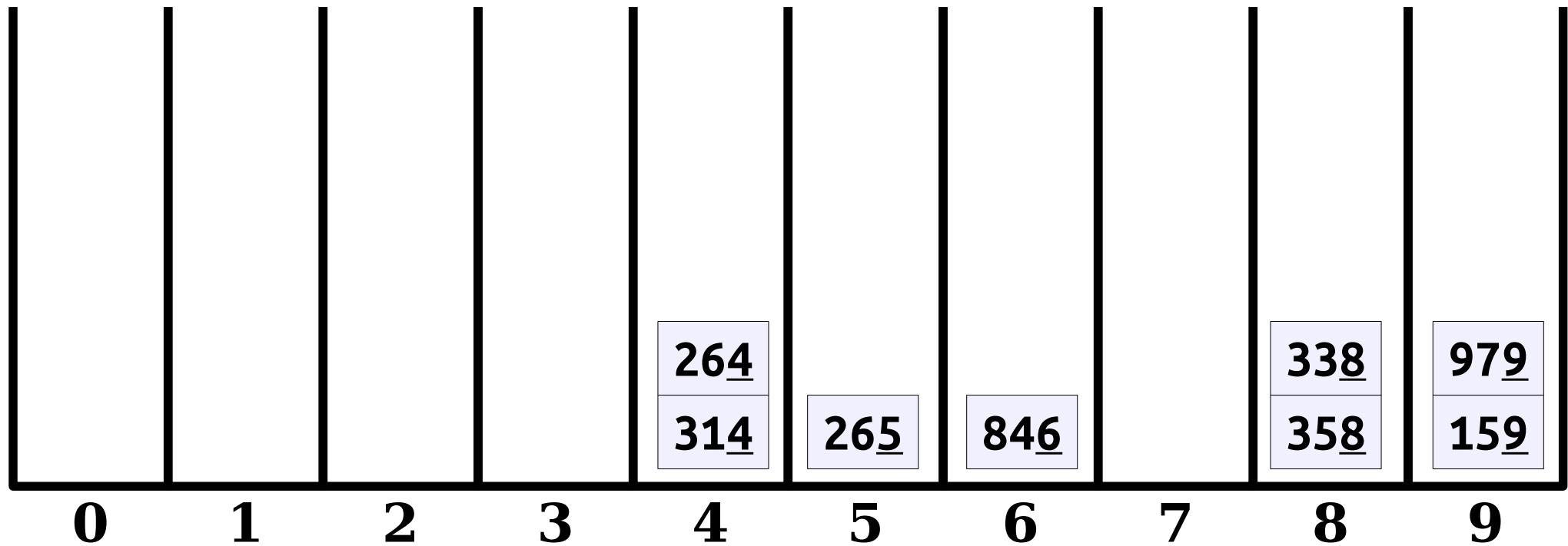
Radix Sort



Radix Sort

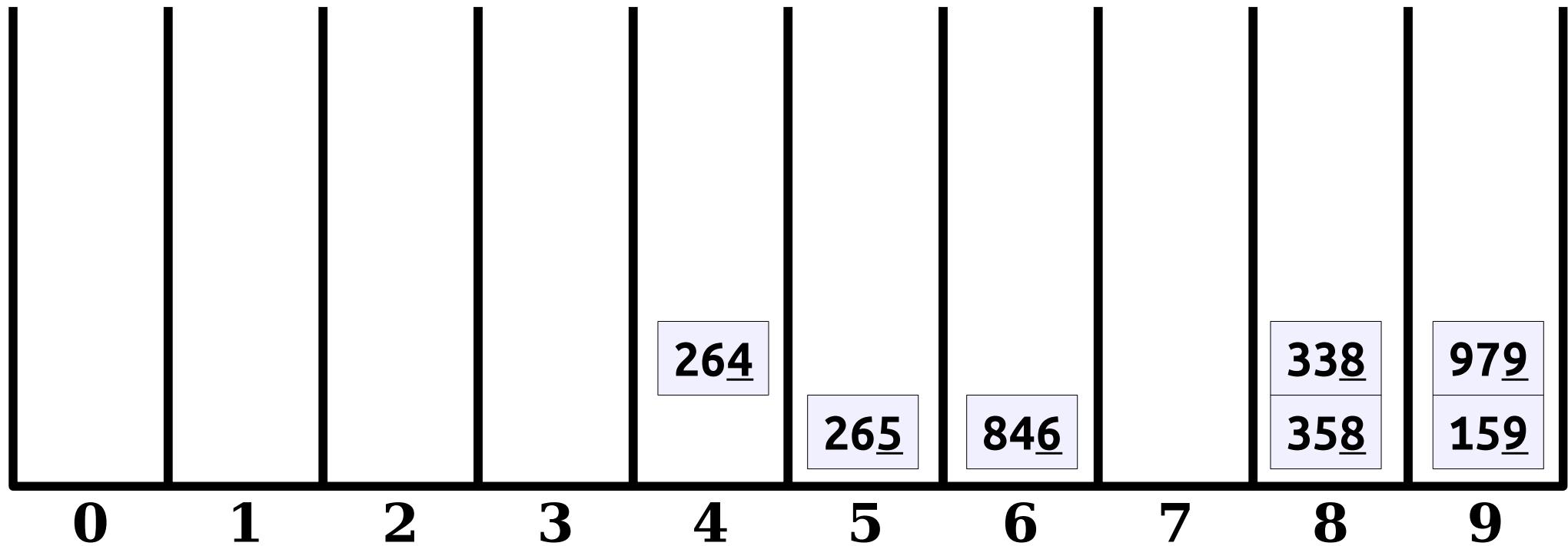


Radix Sort



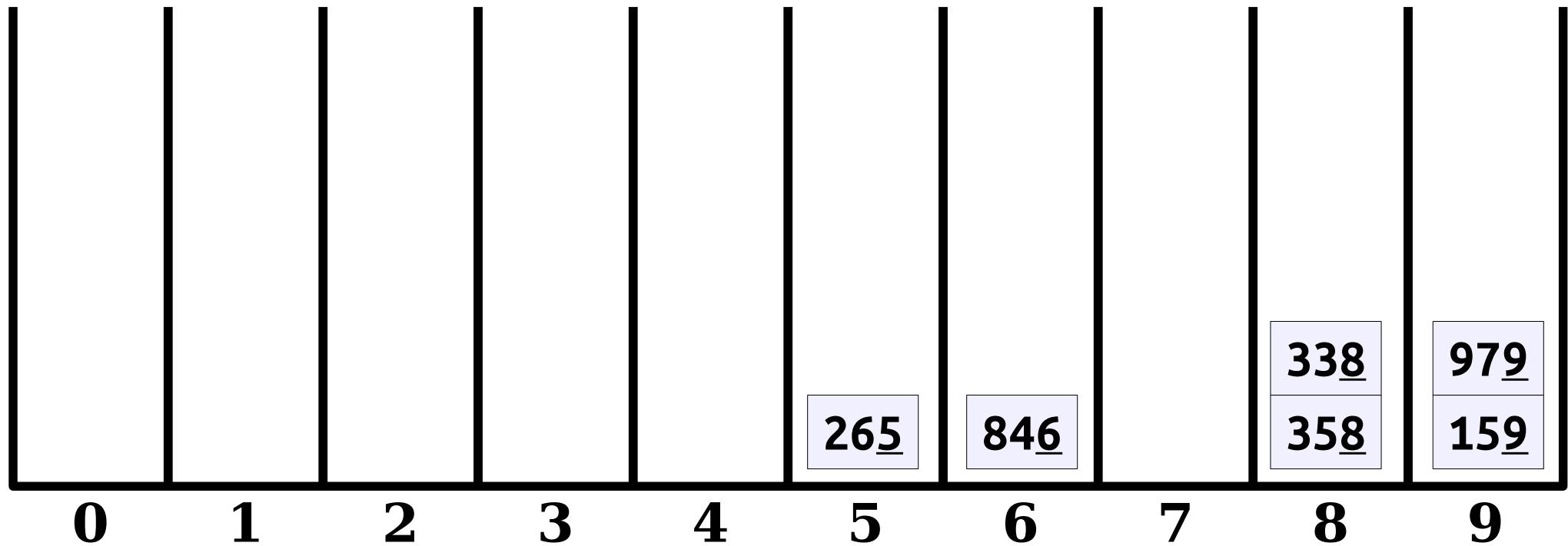
323

Radix Sort



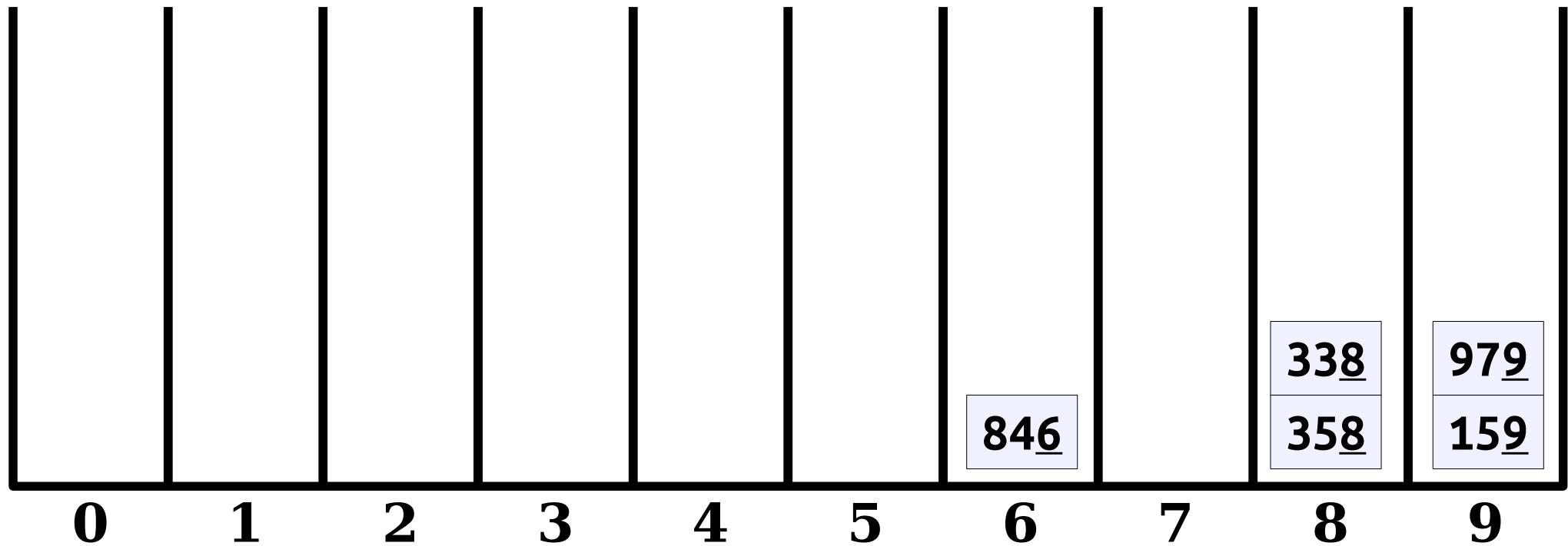
323	314
-----	-----

Radix Sort



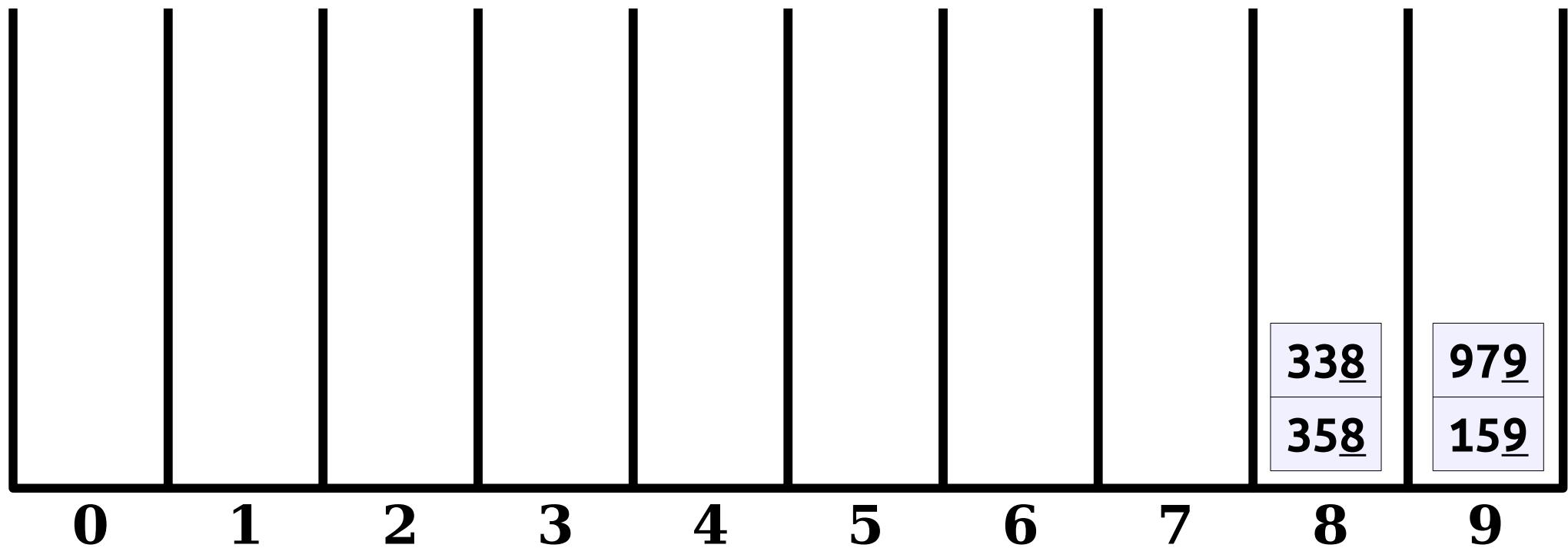
323	314	264
-----	-----	-----

Radix Sort



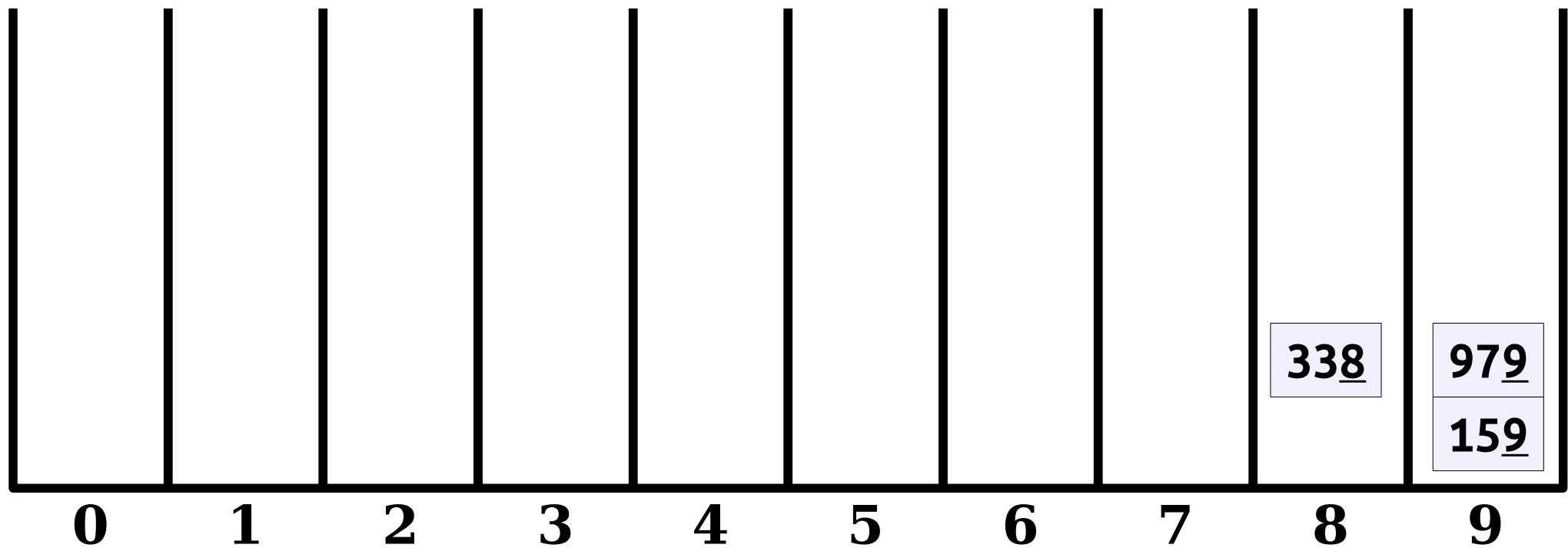
<u>323</u>	<u>314</u>	<u>264</u>	<u>265</u>
------------	------------	------------	------------

Radix Sort



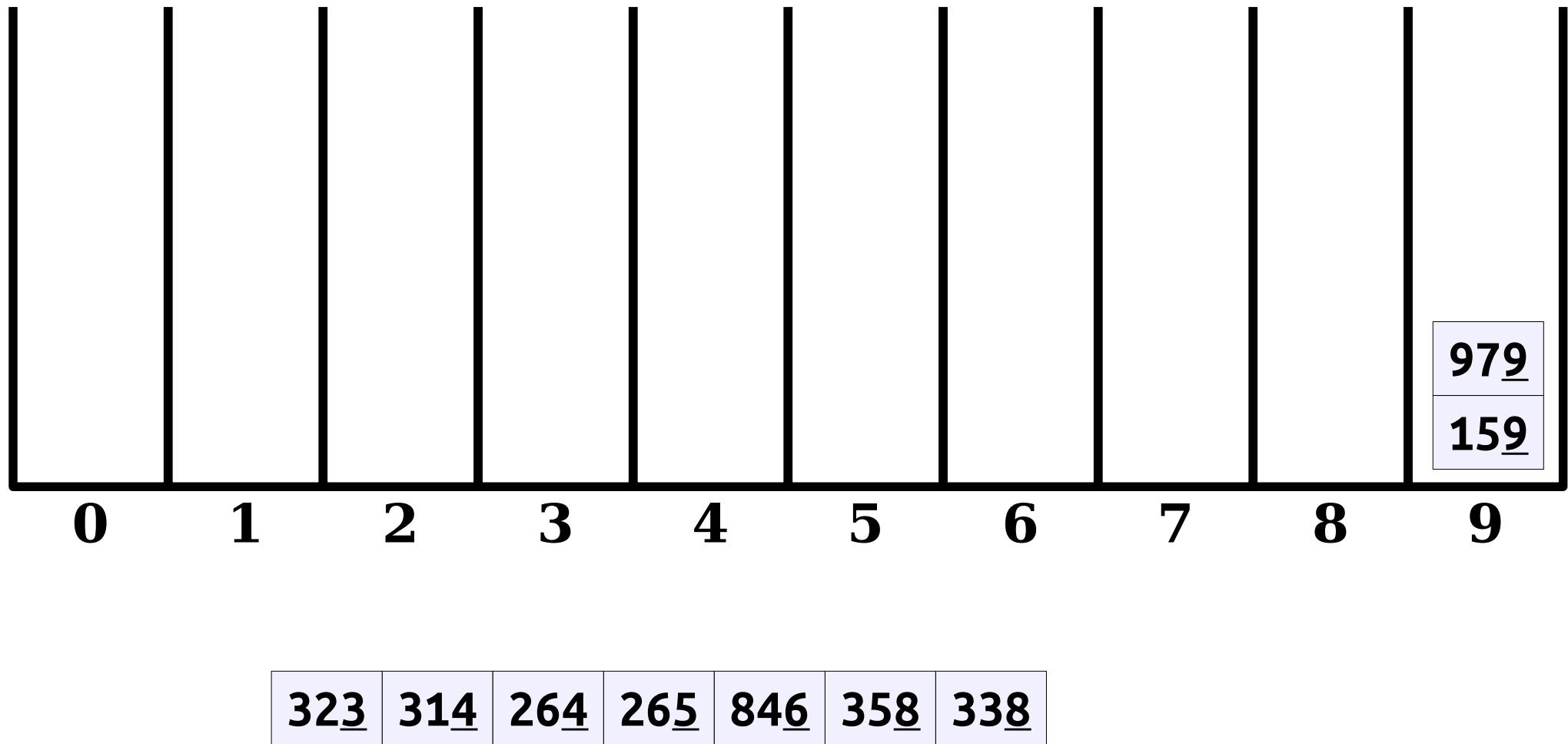
<u>323</u>	<u>314</u>	<u>264</u>	<u>265</u>	<u>846</u>
------------	------------	------------	------------	------------

Radix Sort

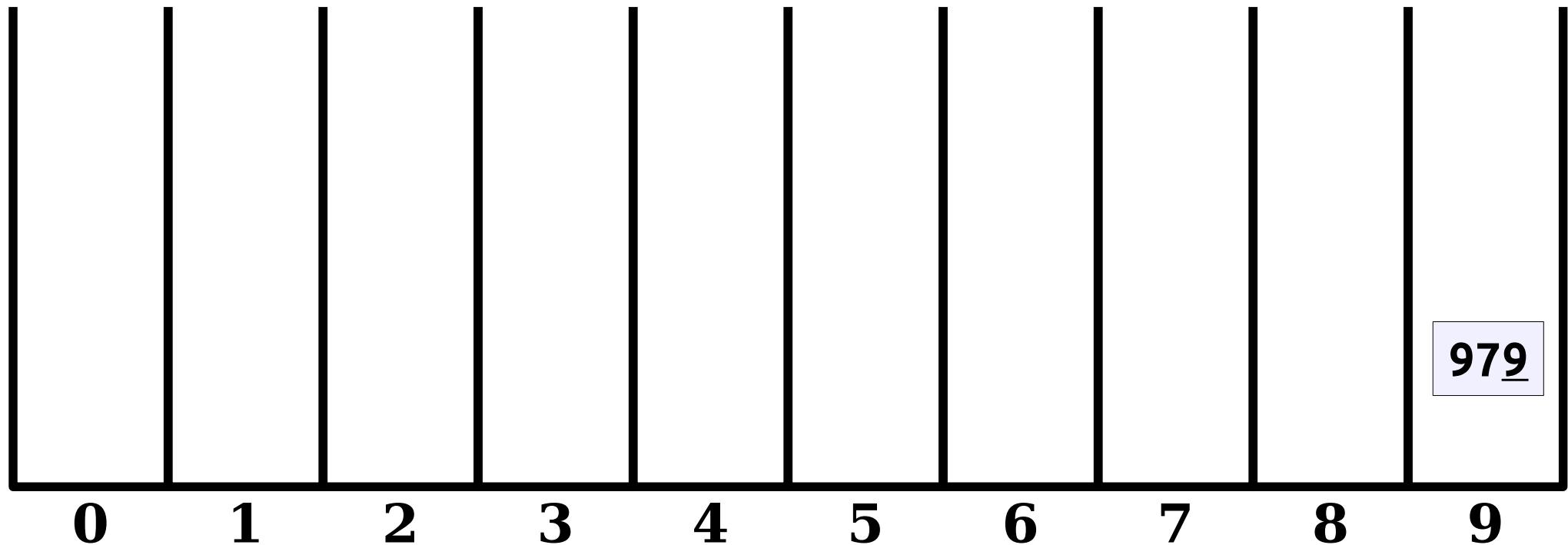


<u>323</u>	<u>314</u>	<u>264</u>	<u>265</u>	<u>846</u>	<u>358</u>
------------	------------	------------	------------	------------	------------

Radix Sort

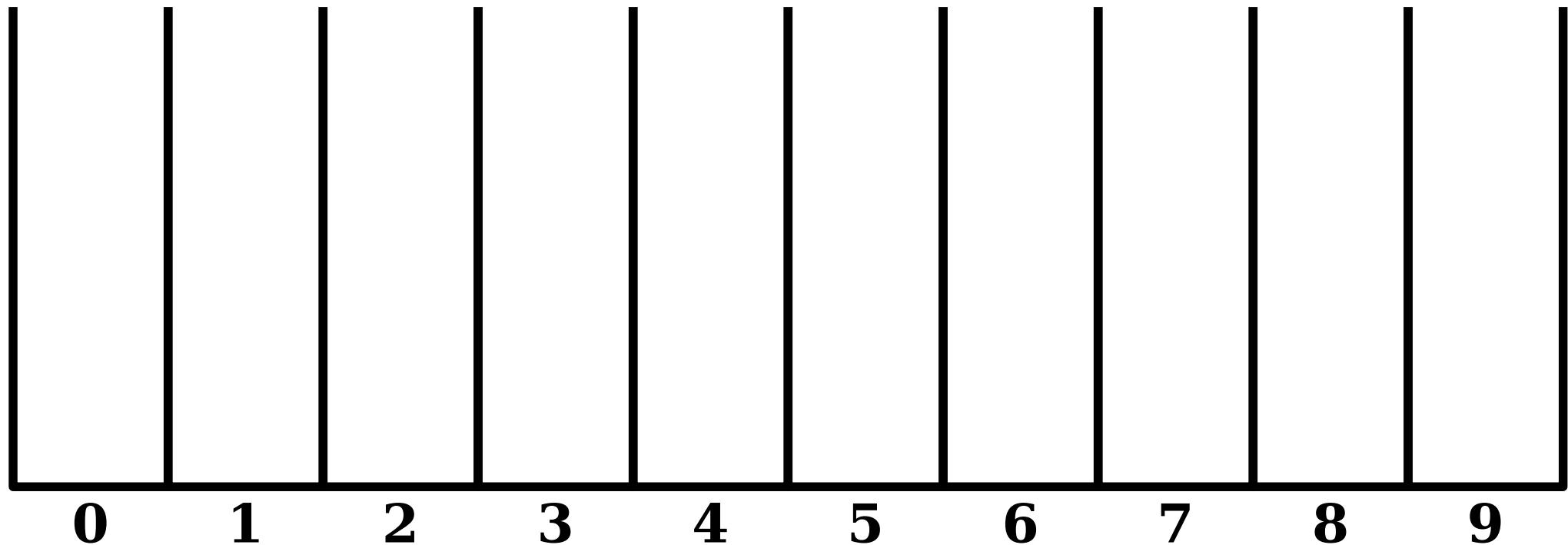


Radix Sort



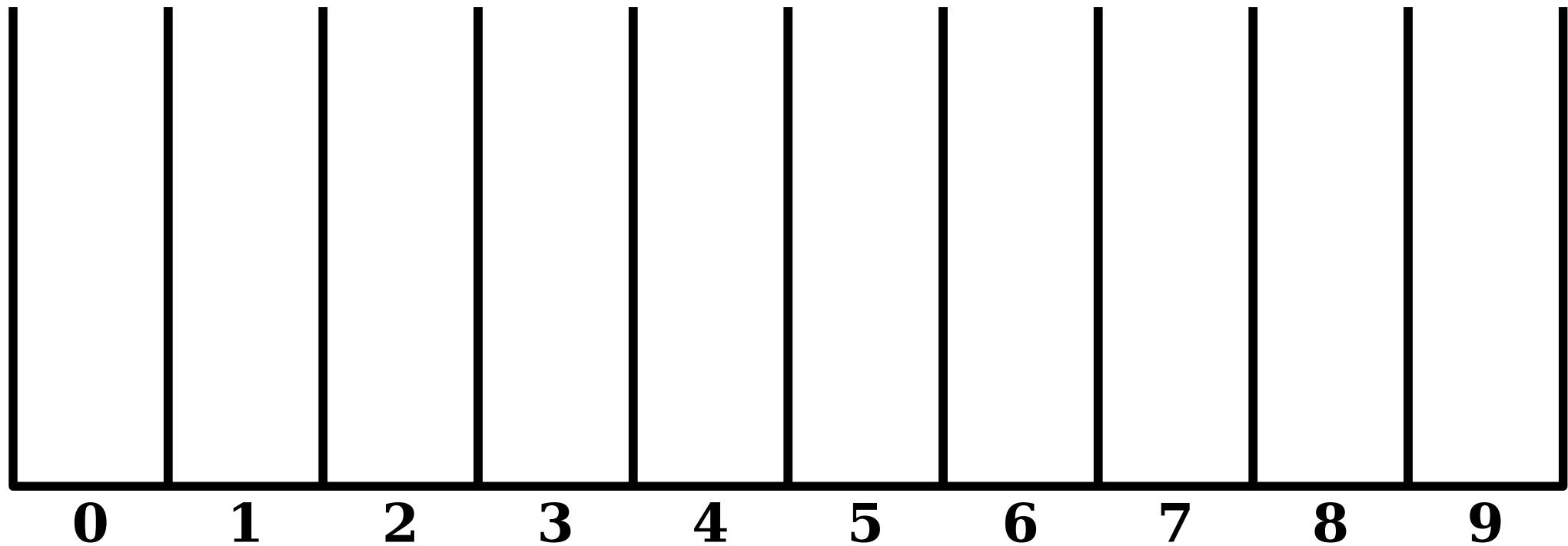
<u>323</u>	<u>314</u>	<u>264</u>	<u>265</u>	<u>846</u>	<u>358</u>	<u>338</u>	<u>159</u>
------------	------------	------------	------------	------------	------------	------------	------------

Radix Sort

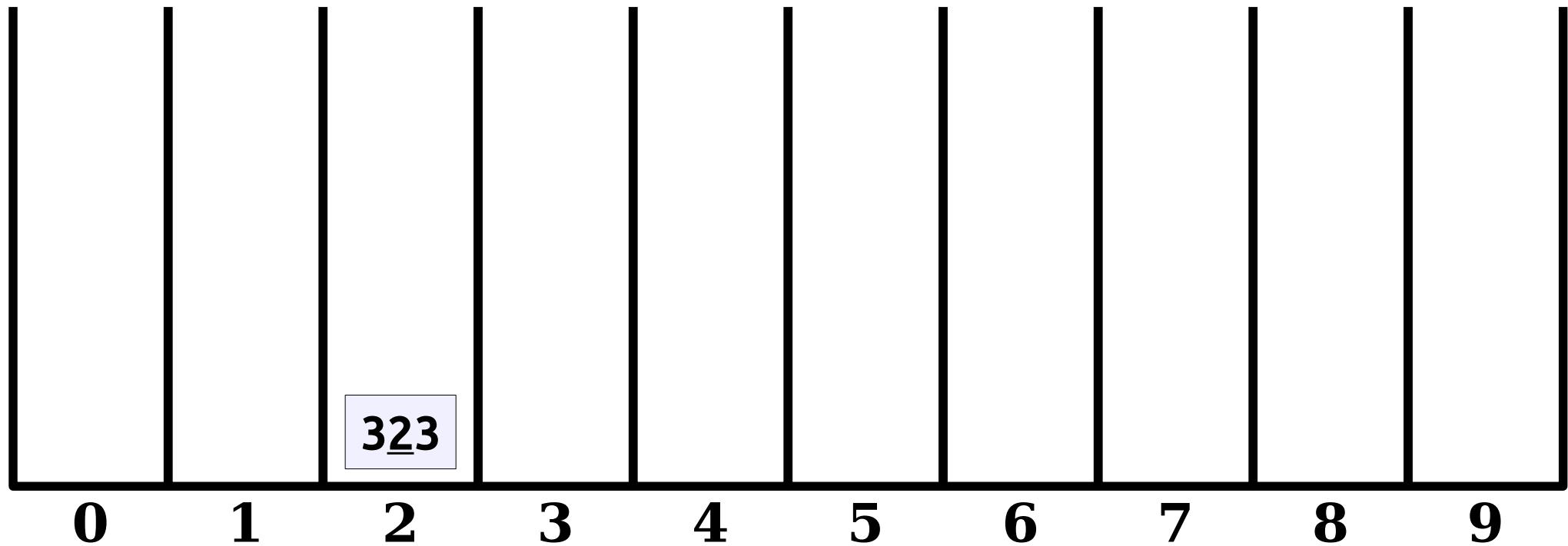


3 <u>2</u> 3	3 <u>1</u> 4	2 <u>6</u> 4	2 <u>6</u> 5	8 <u>4</u> 6	3 <u>5</u> 8	3 <u>3</u> 8	1 <u>5</u> 9	9 <u>7</u> 9
--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------

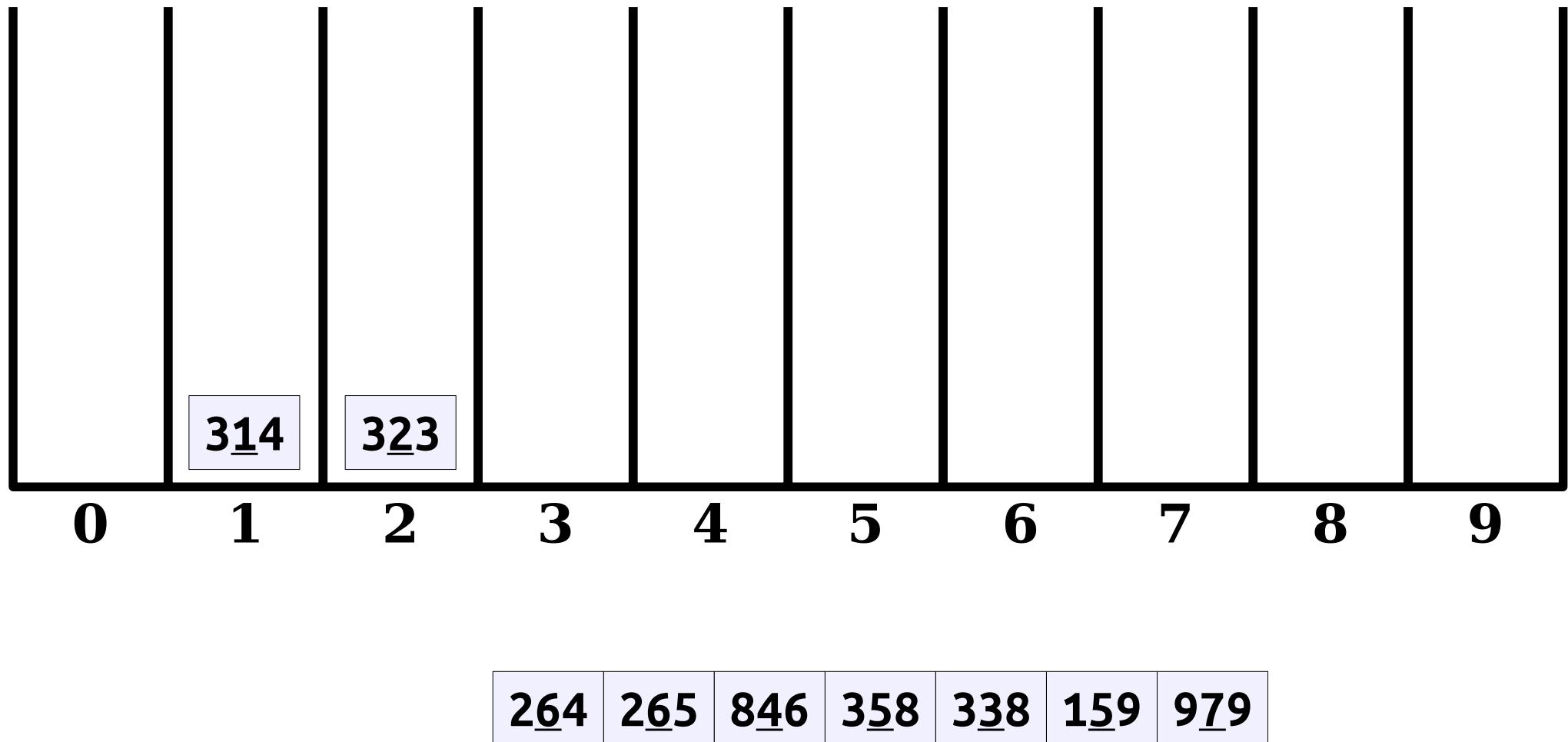
Radix Sort



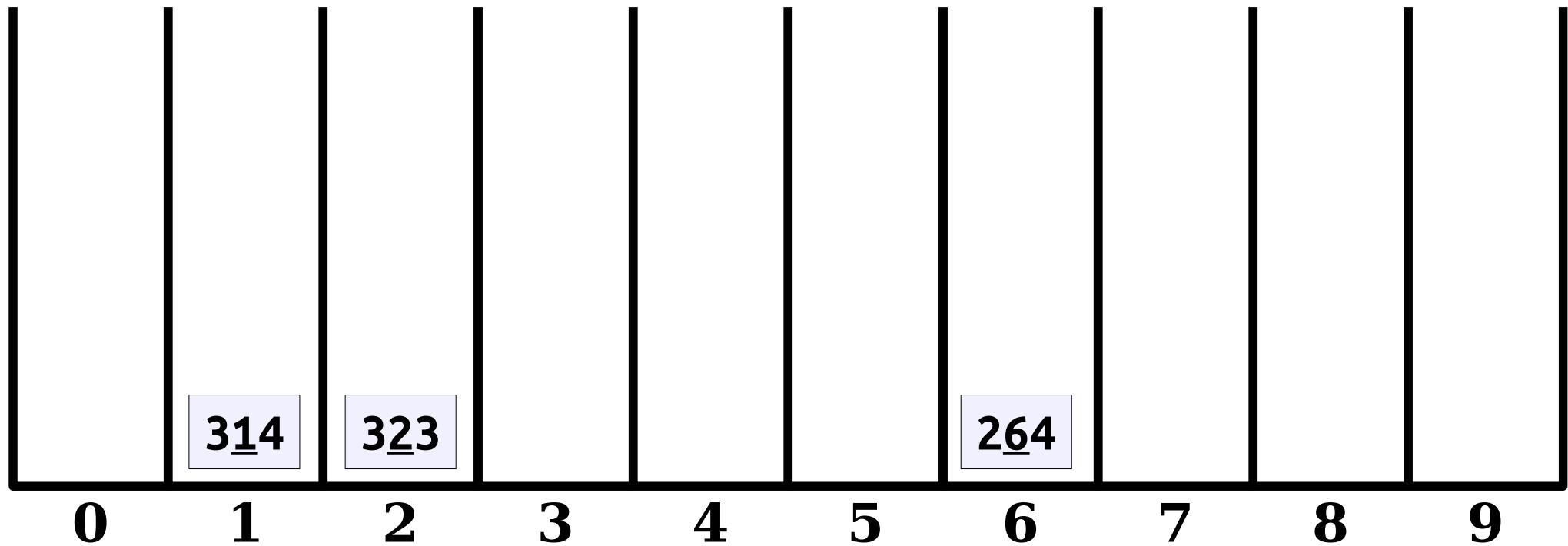
Radix Sort



Radix Sort

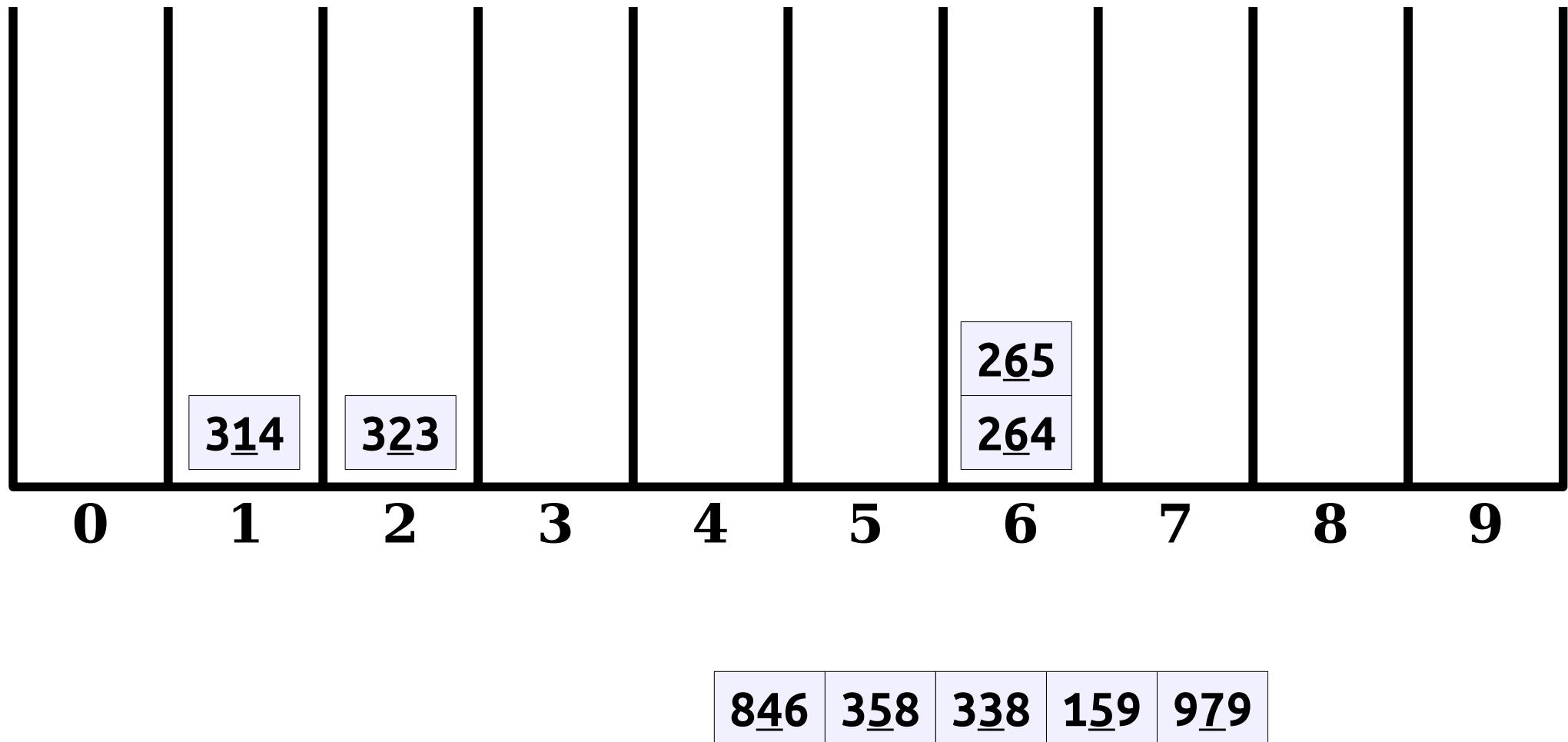


Radix Sort

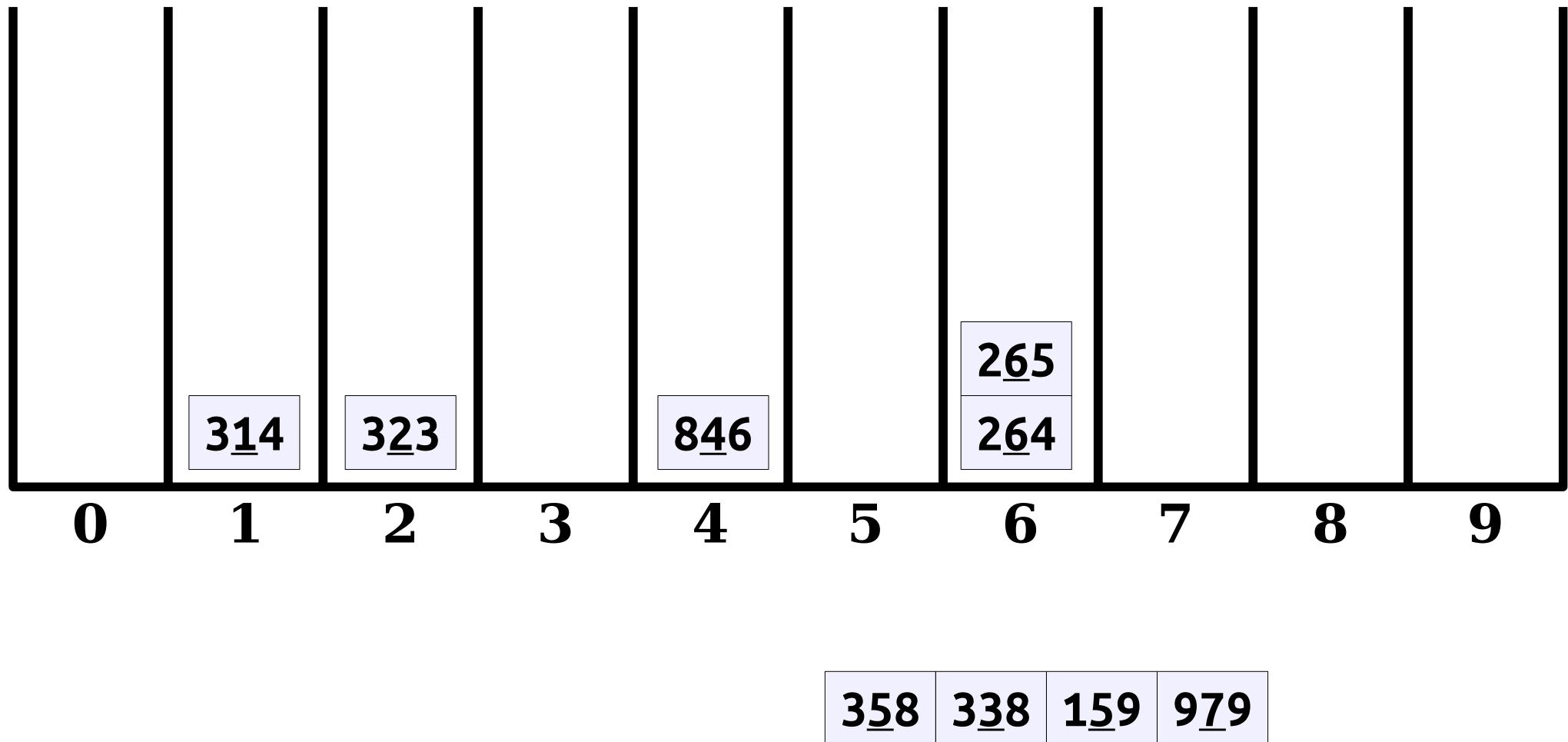


2 <u>6</u> 5	8 <u>4</u> 6	3 <u>5</u> 8	3 <u>3</u> 8	1 <u>5</u> 9	9 <u>7</u> 9
--------------	--------------	--------------	--------------	--------------	--------------

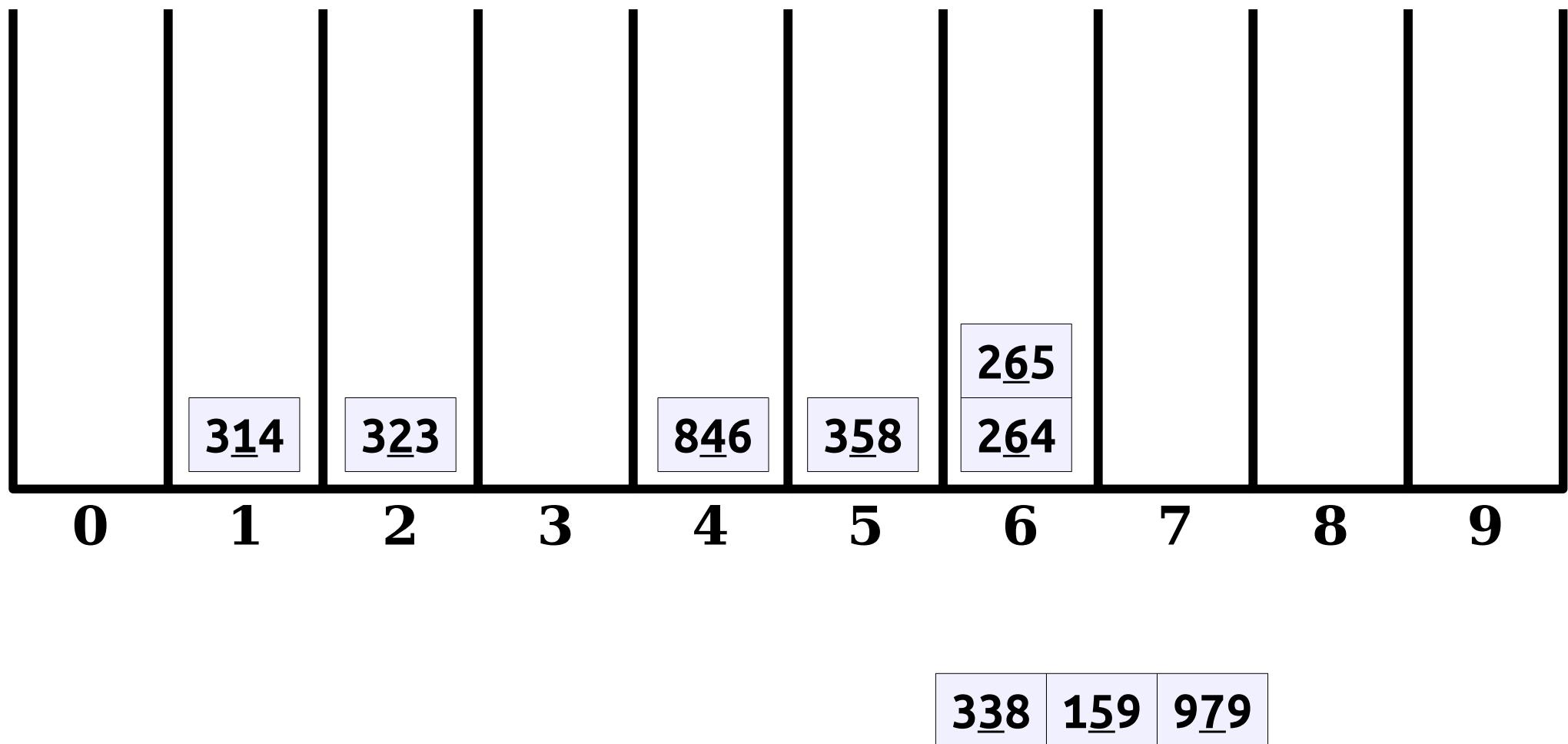
Radix Sort



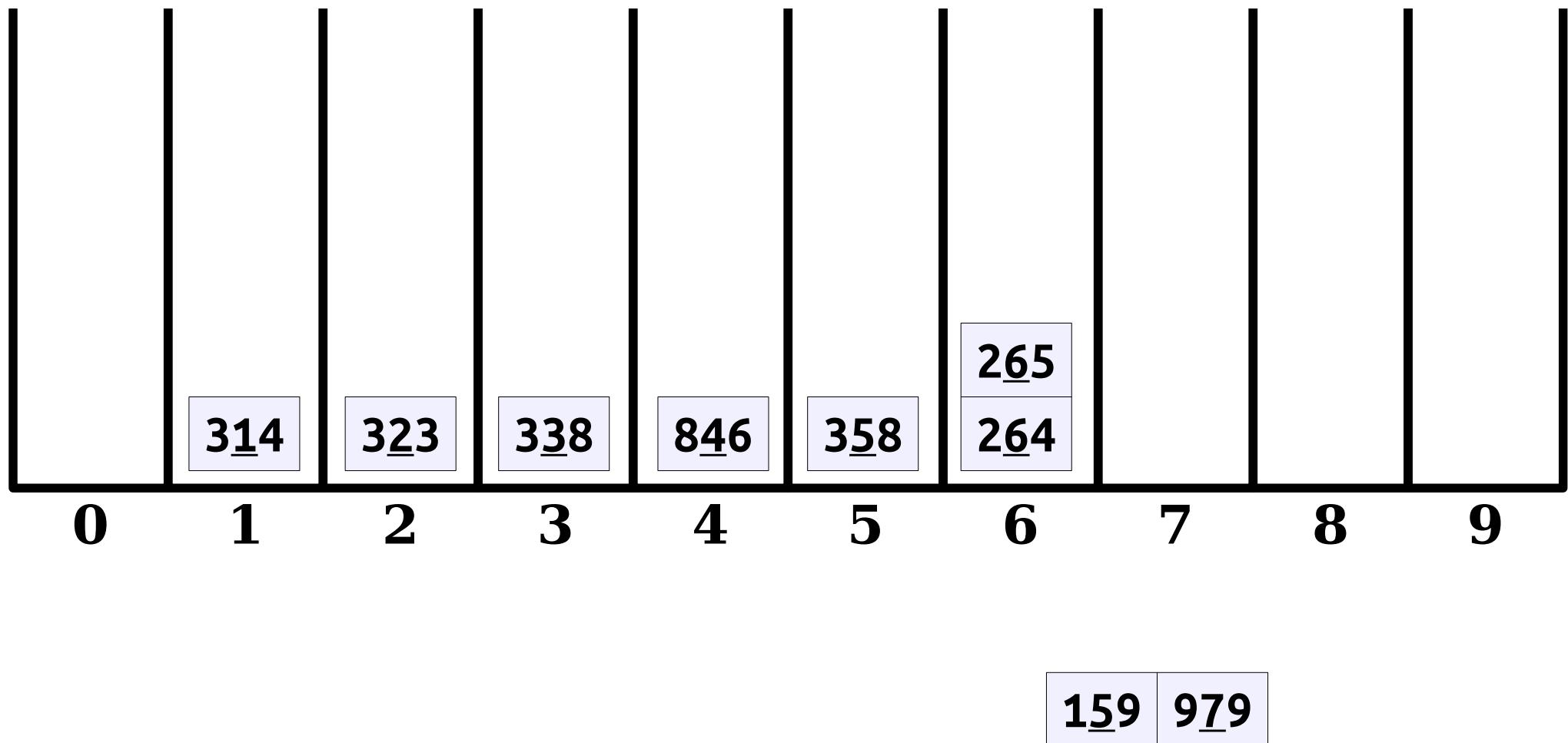
Radix Sort



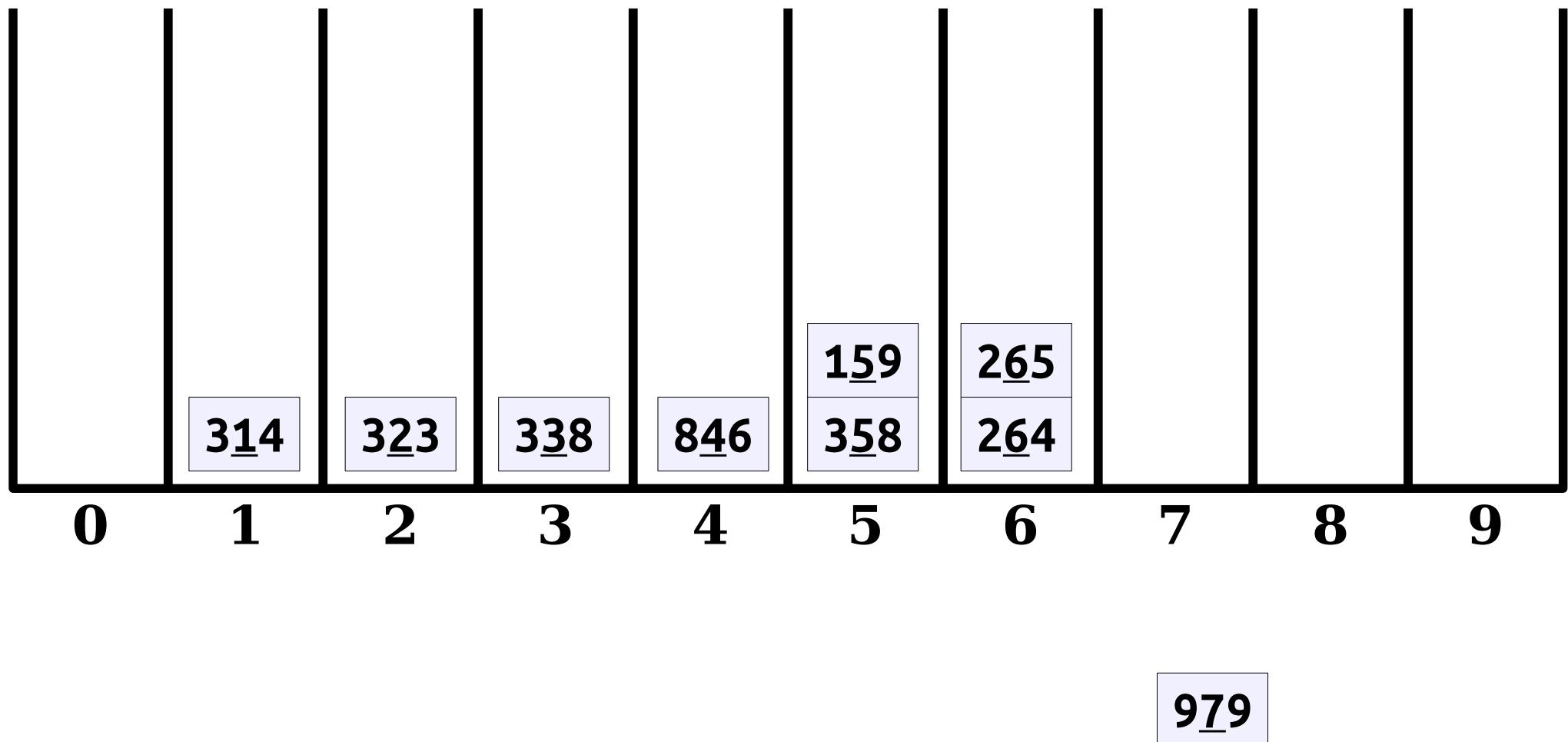
Radix Sort



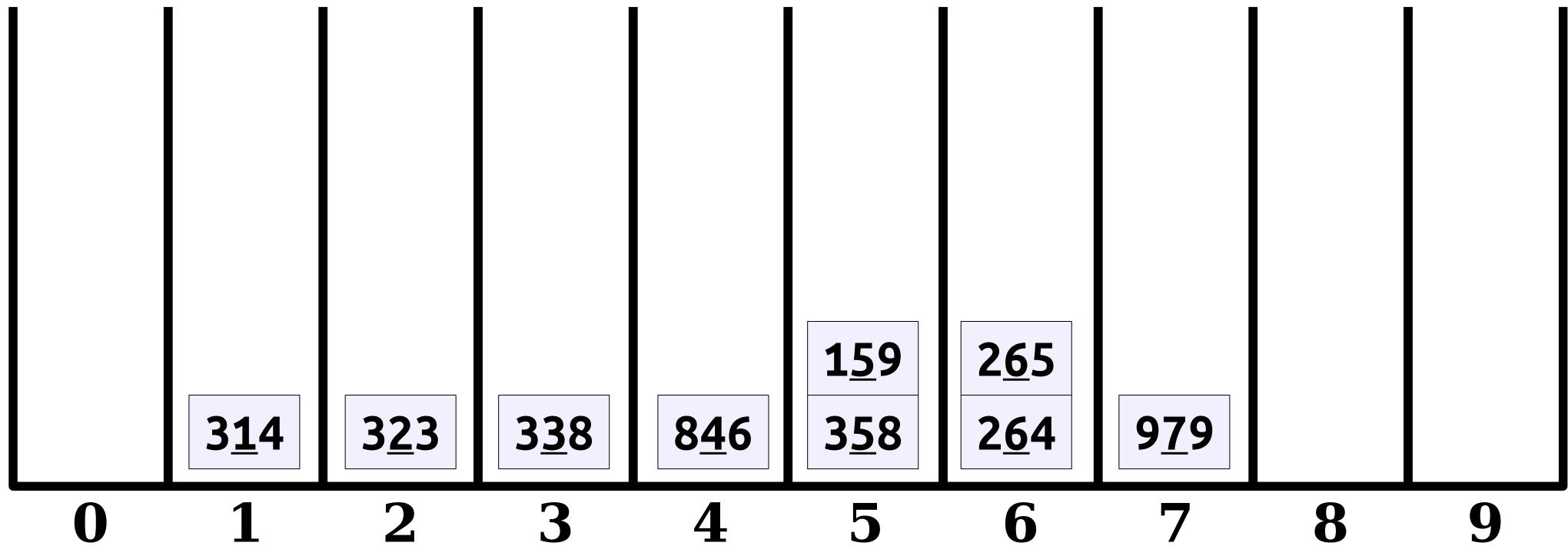
Radix Sort



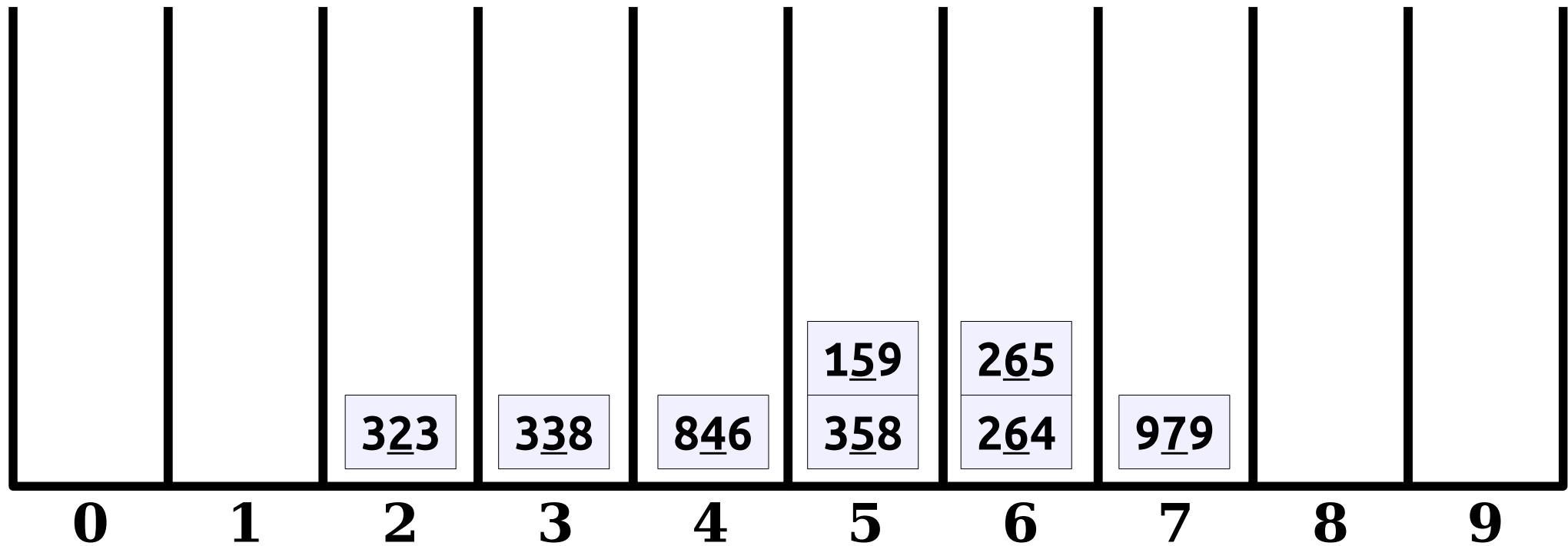
Radix Sort



Radix Sort

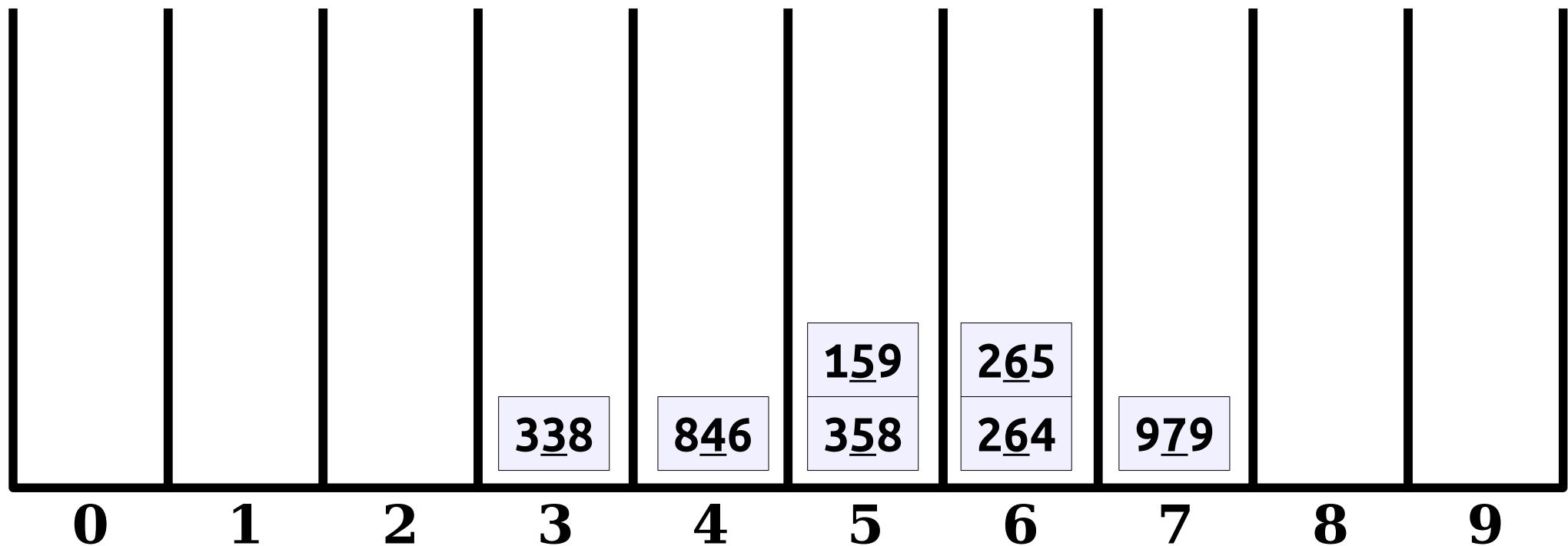


Radix Sort



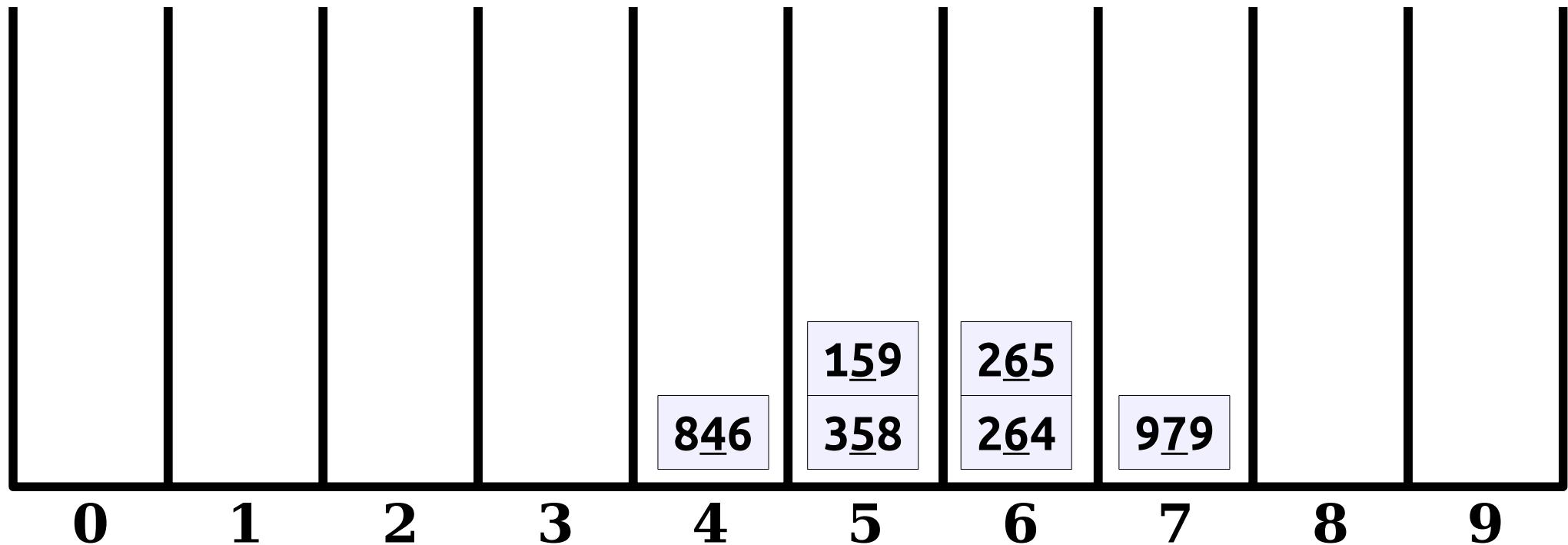
314

Radix Sort



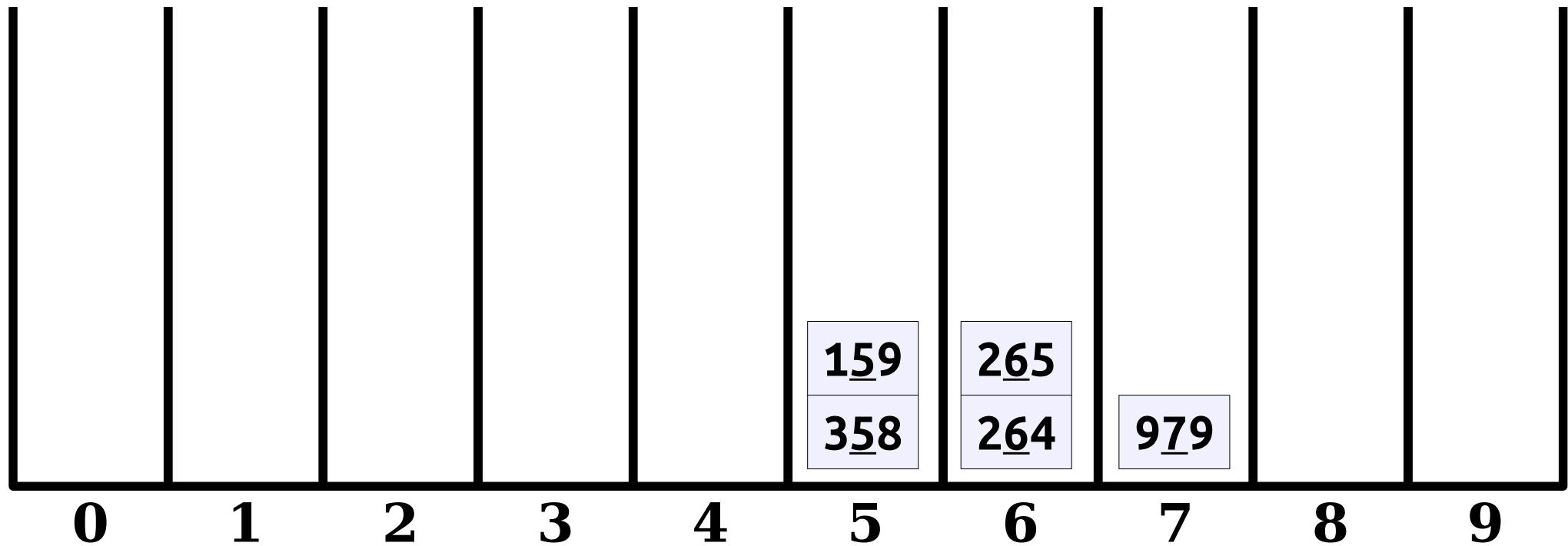
<u>314</u>	<u>323</u>
------------	------------

Radix Sort



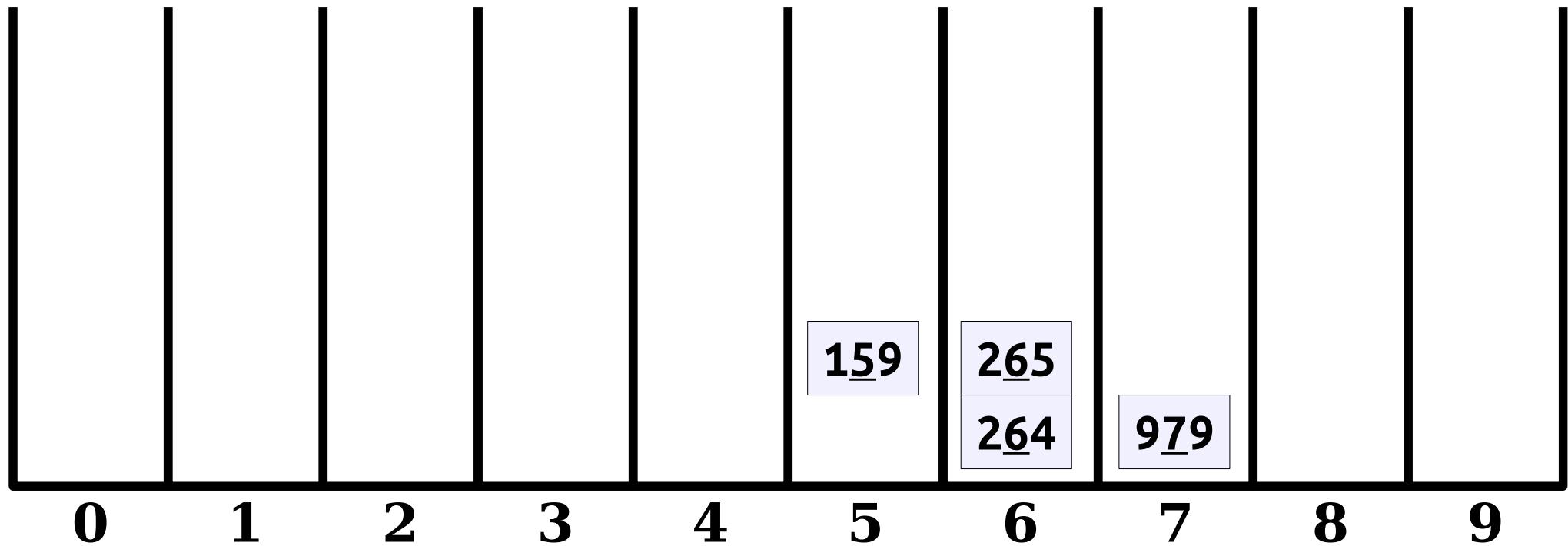
<u>314</u>	<u>323</u>	<u>338</u>
------------	------------	------------

Radix Sort



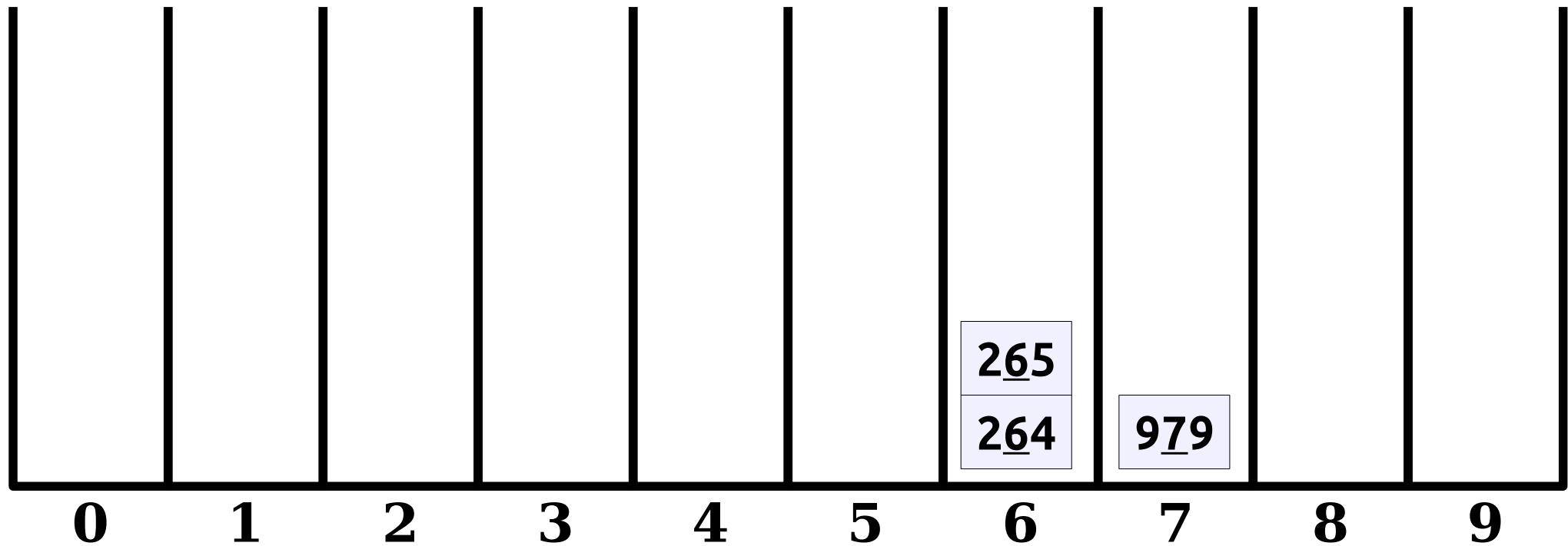
3<u>1</u>4	3<u>2</u>3	3<u>3</u>8	8<u>4</u>6
-------------------	-------------------	-------------------	-------------------

Radix Sort



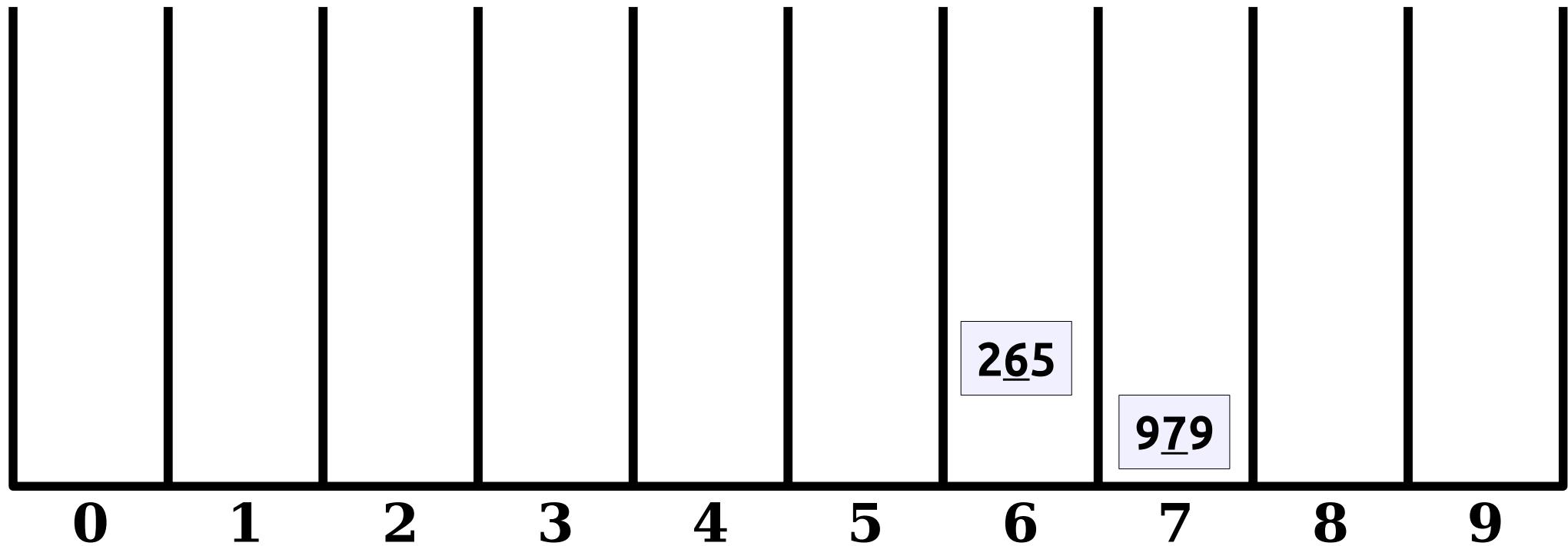
<u>314</u>	<u>323</u>	<u>338</u>	<u>846</u>	<u>358</u>
------------	------------	------------	------------	------------

Radix Sort



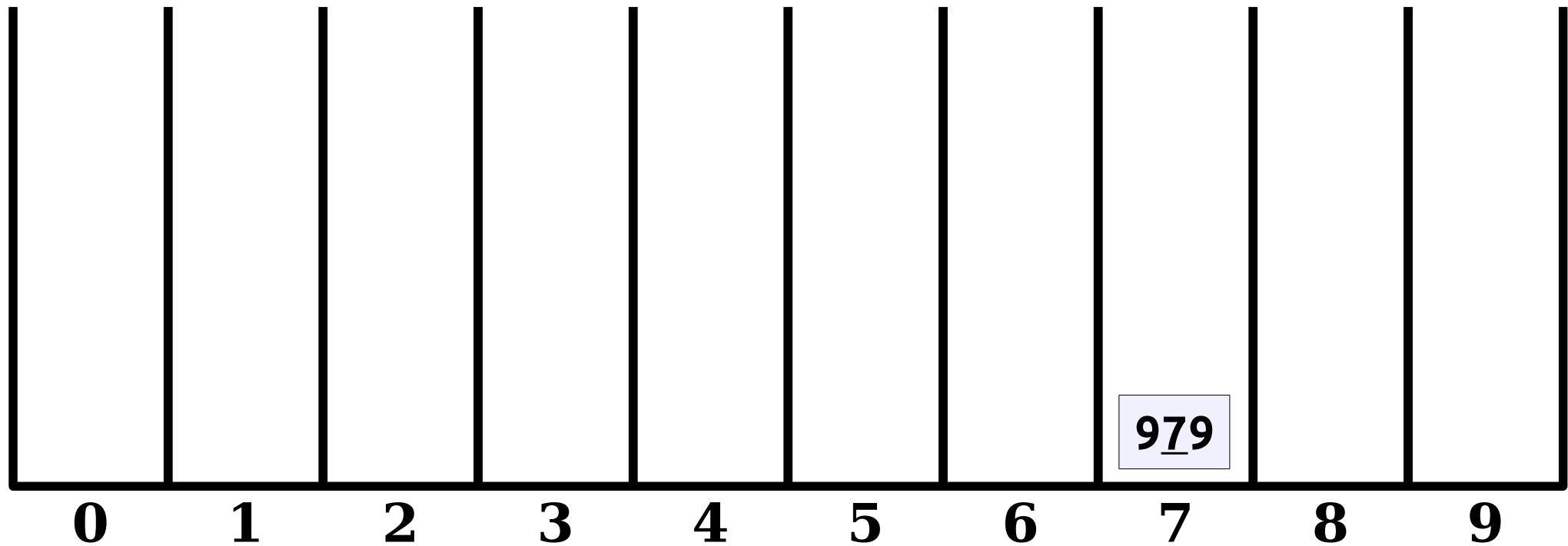
3<u>1</u>4	3<u>2</u>3	3<u>3</u>8	8<u>4</u>6	3<u>5</u>8	1<u>5</u>9
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

Radix Sort



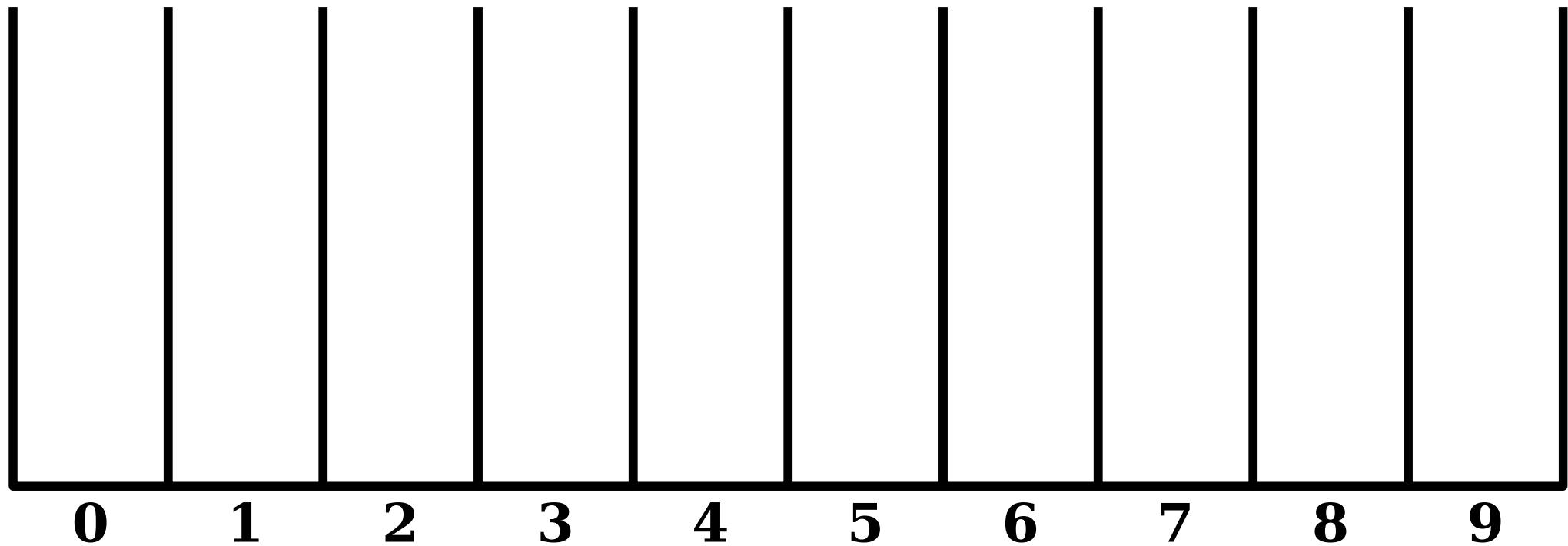
<u>314</u>	<u>323</u>	<u>338</u>	<u>846</u>	<u>358</u>	<u>159</u>	<u>264</u>
------------	------------	------------	------------	------------	------------	------------

Radix Sort



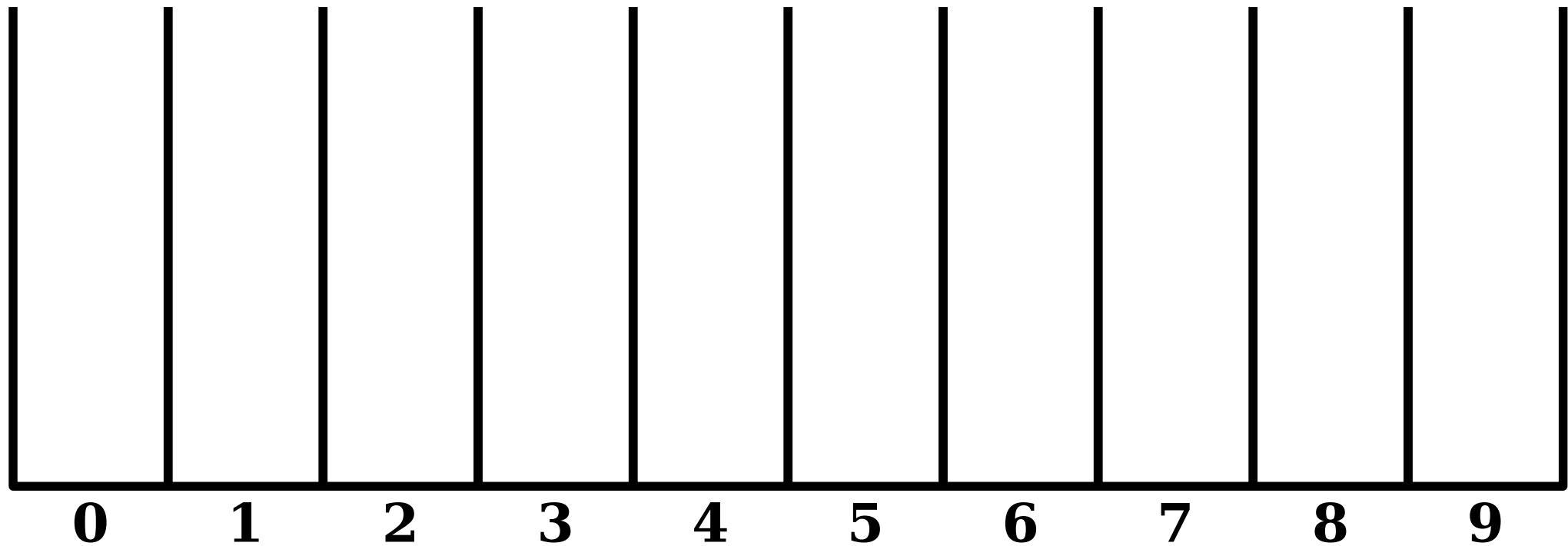
3 <u>1</u> 4	3 <u>2</u> 3	3 <u>3</u> 8	8 <u>4</u> 6	3 <u>5</u> 8	1 <u>5</u> 9	2 <u>6</u> 4	2 <u>6</u> 5
--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------

Radix Sort



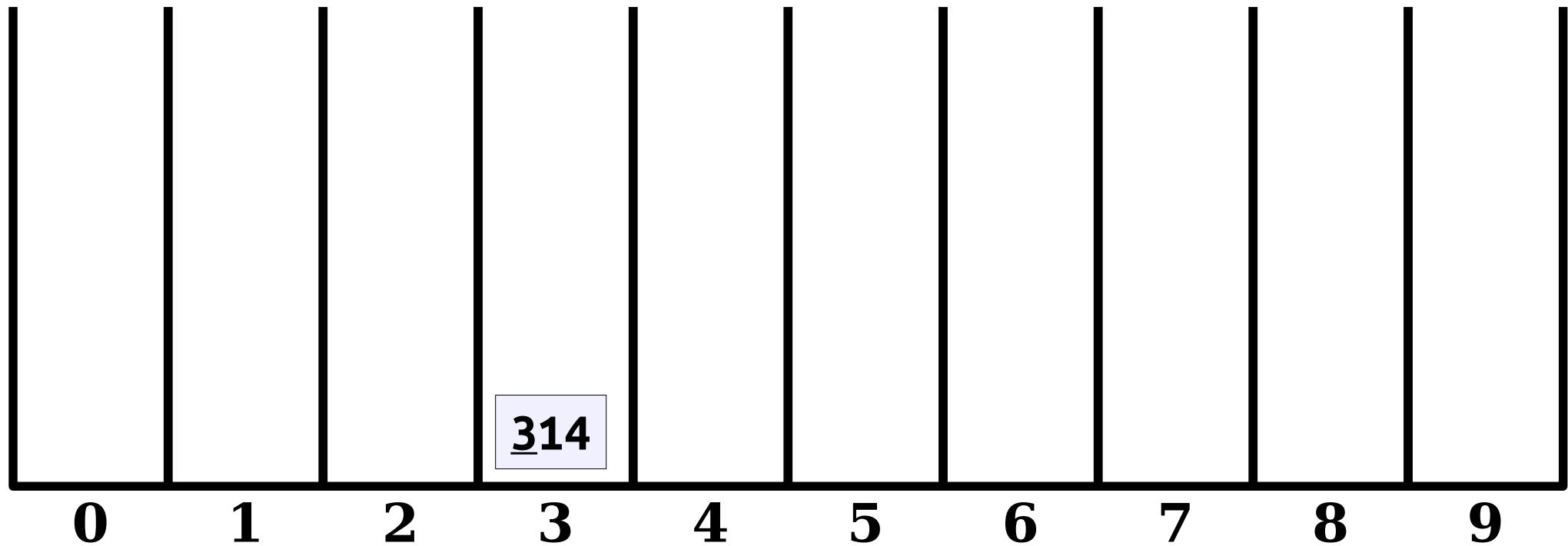
3 <u>1</u> 4	3 <u>2</u> 3	3 <u>3</u> 8	8 <u>4</u> 6	3 <u>5</u> 8	1 <u>5</u> 9	2 <u>6</u> 4	2 <u>6</u> 5	9 <u>7</u> 9
--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------	--------------

Radix Sort



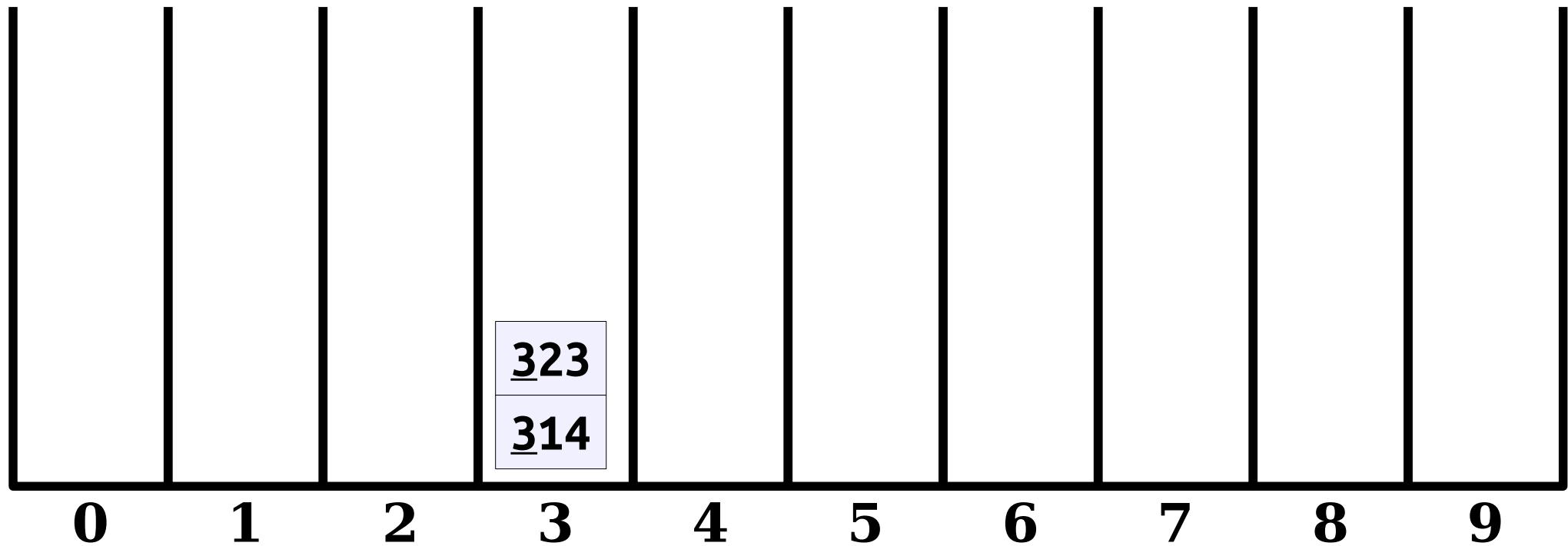
<u>3</u> 14	<u>3</u> 23	<u>3</u> 38	<u>8</u> 46	<u>3</u> 58	<u>1</u> 59	<u>2</u> 64	<u>2</u> 65	<u>9</u> 79
-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

Radix Sort



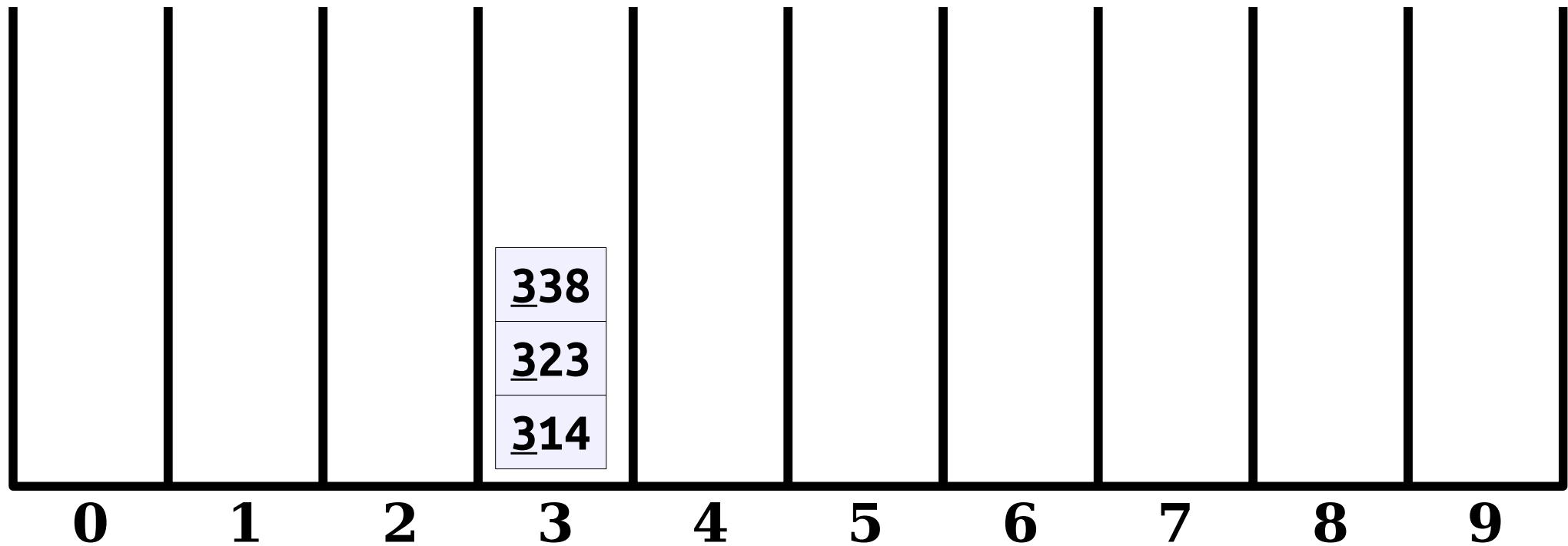
<u>323</u>	<u>338</u>	<u>846</u>	<u>358</u>	<u>159</u>	<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------	------------	------------	------------	------------	------------

Radix Sort



<u>338</u>	<u>846</u>	<u>358</u>	<u>159</u>	<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------	------------	------------	------------	------------

Radix Sort



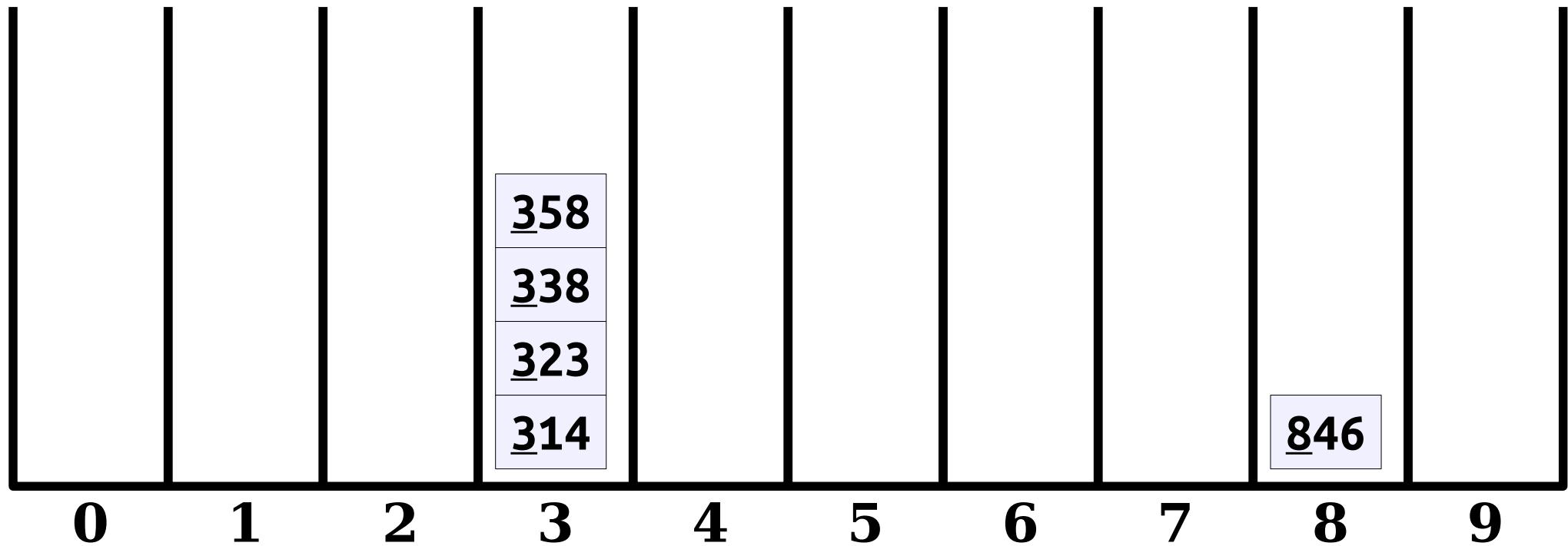
<u>846</u>	<u>358</u>	<u>159</u>	<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------	------------	------------	------------

Radix Sort



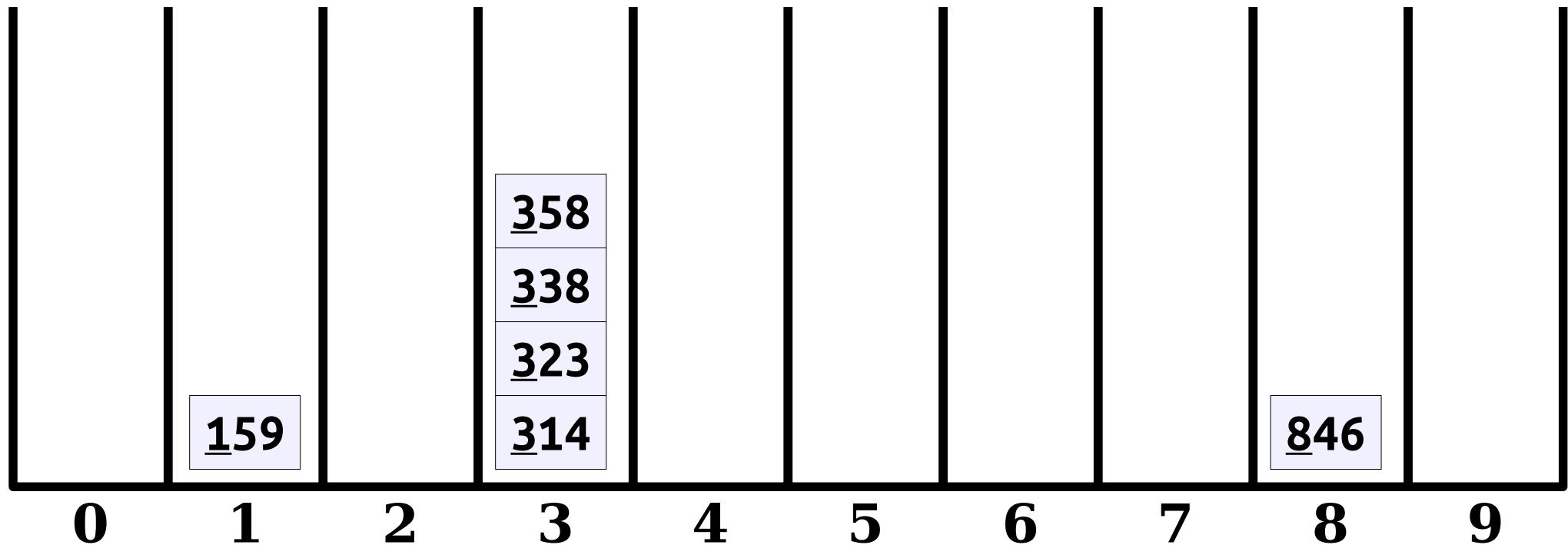
<u>358</u>	<u>159</u>	<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------	------------	------------

Radix Sort



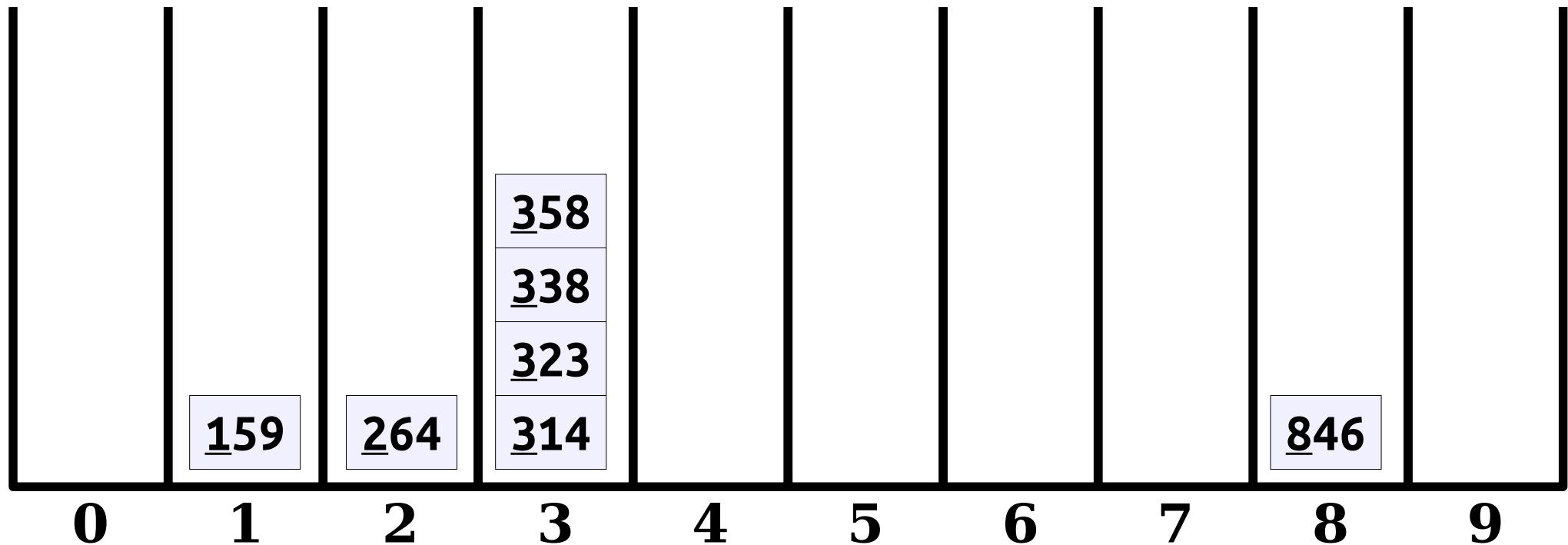
<u>159</u>	<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------	------------

Radix Sort



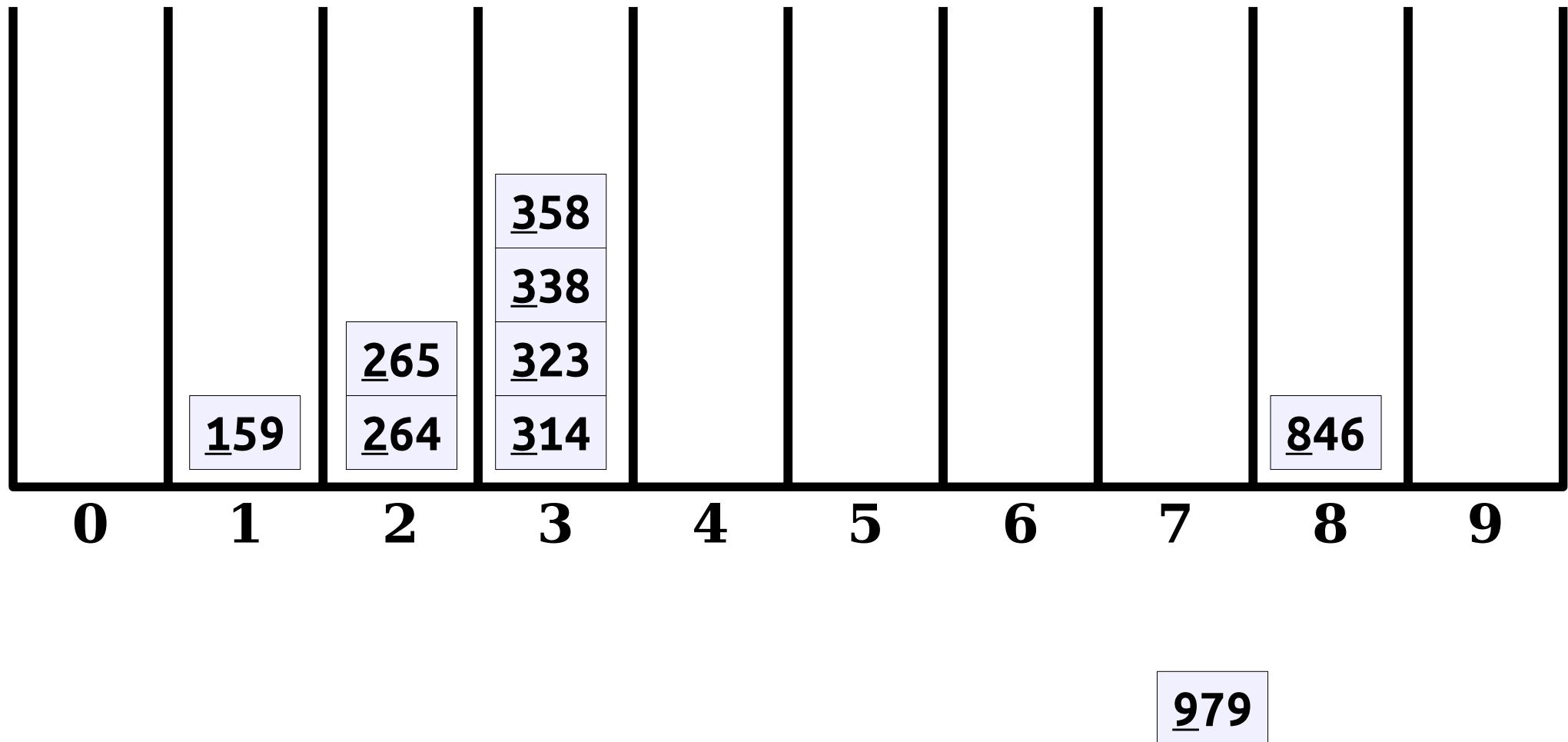
<u>264</u>	<u>265</u>	<u>979</u>
------------	------------	------------

Radix Sort

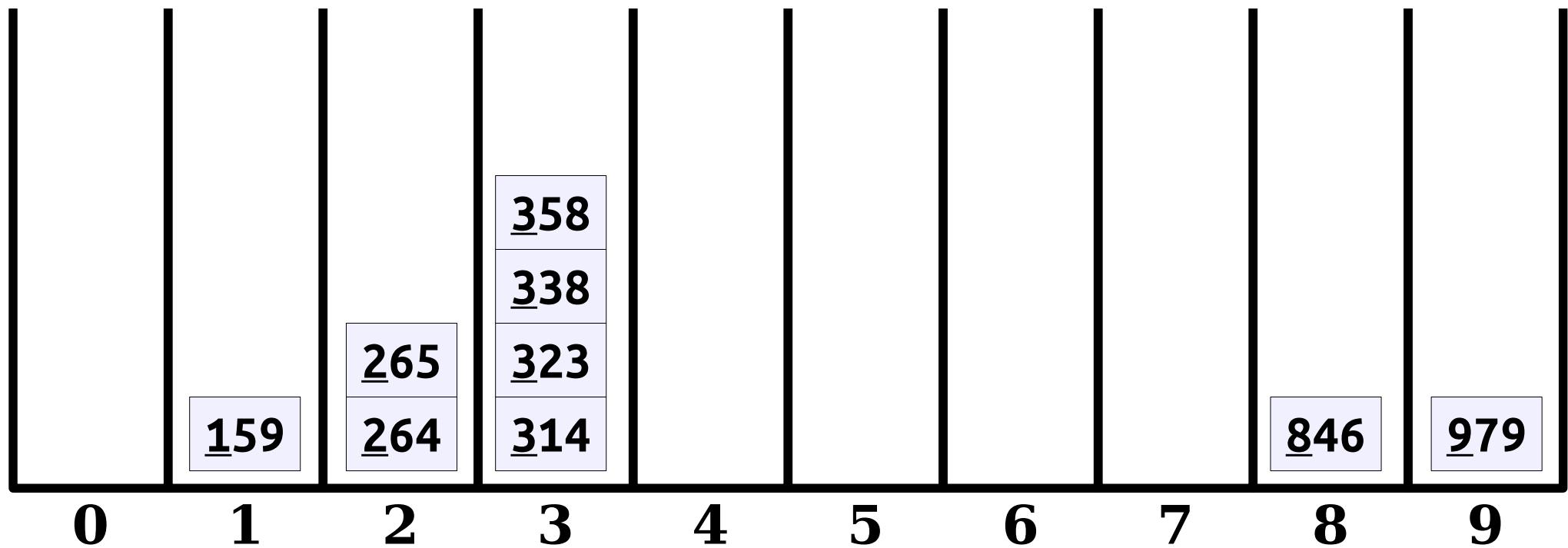


<u>265</u>	<u>979</u>
------------	------------

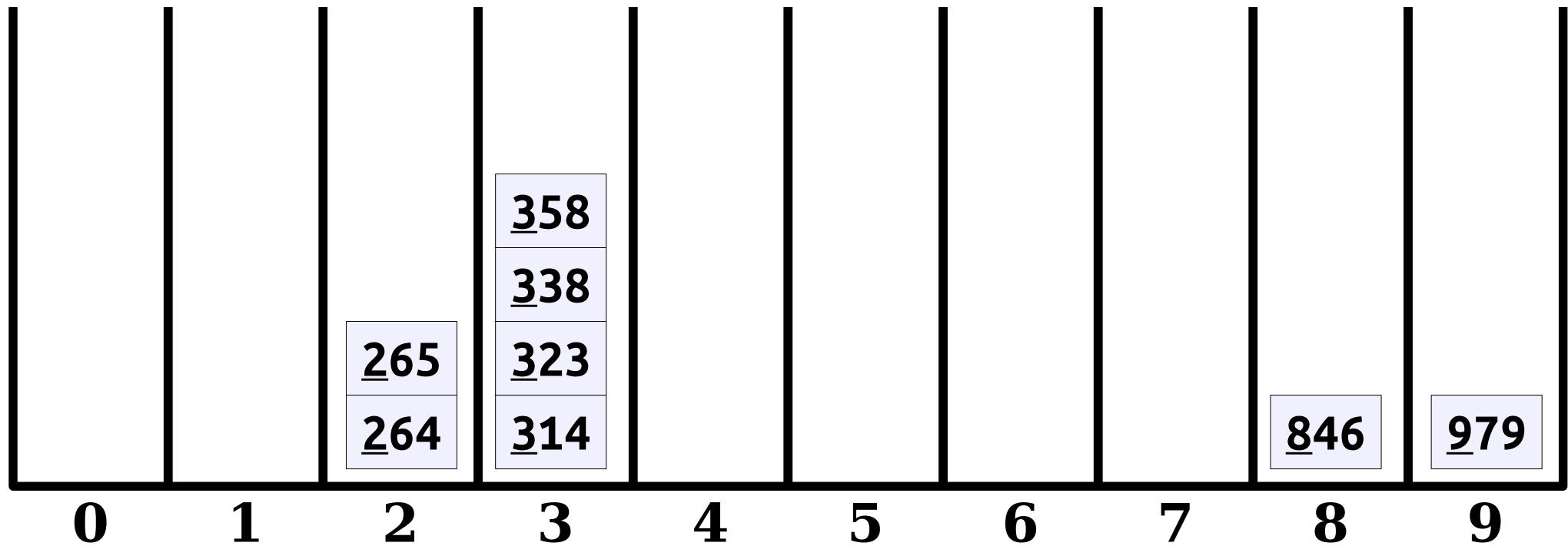
Radix Sort



Radix Sort

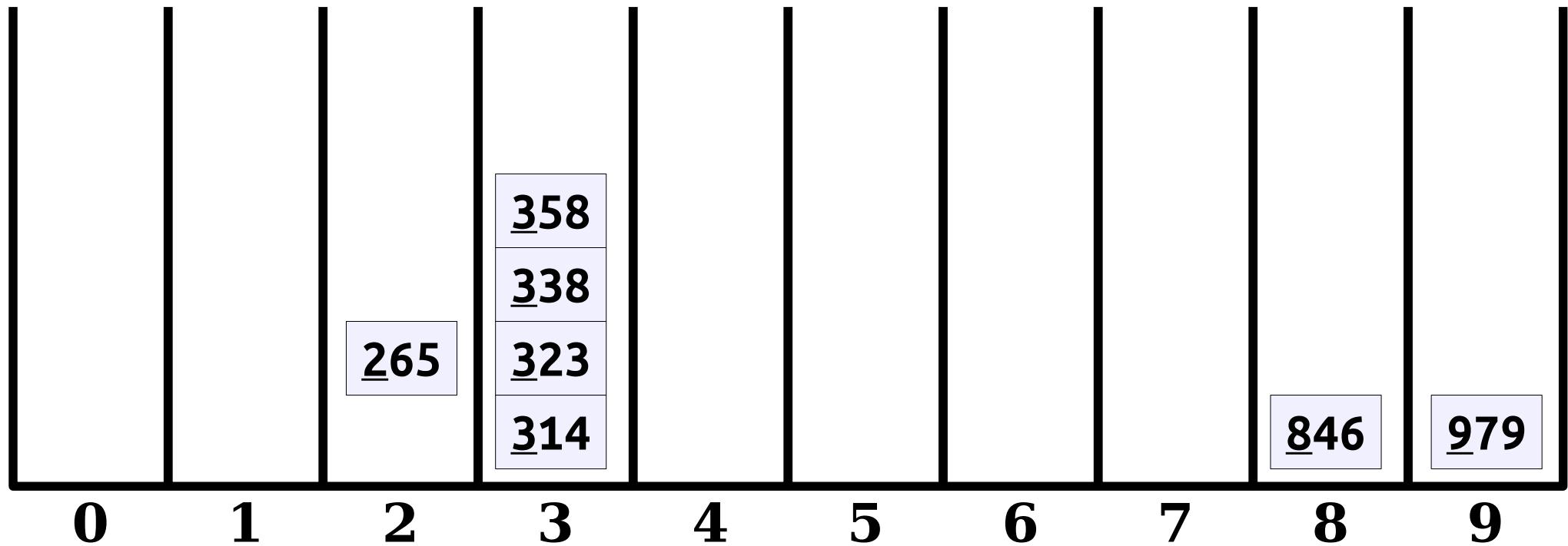


Radix Sort



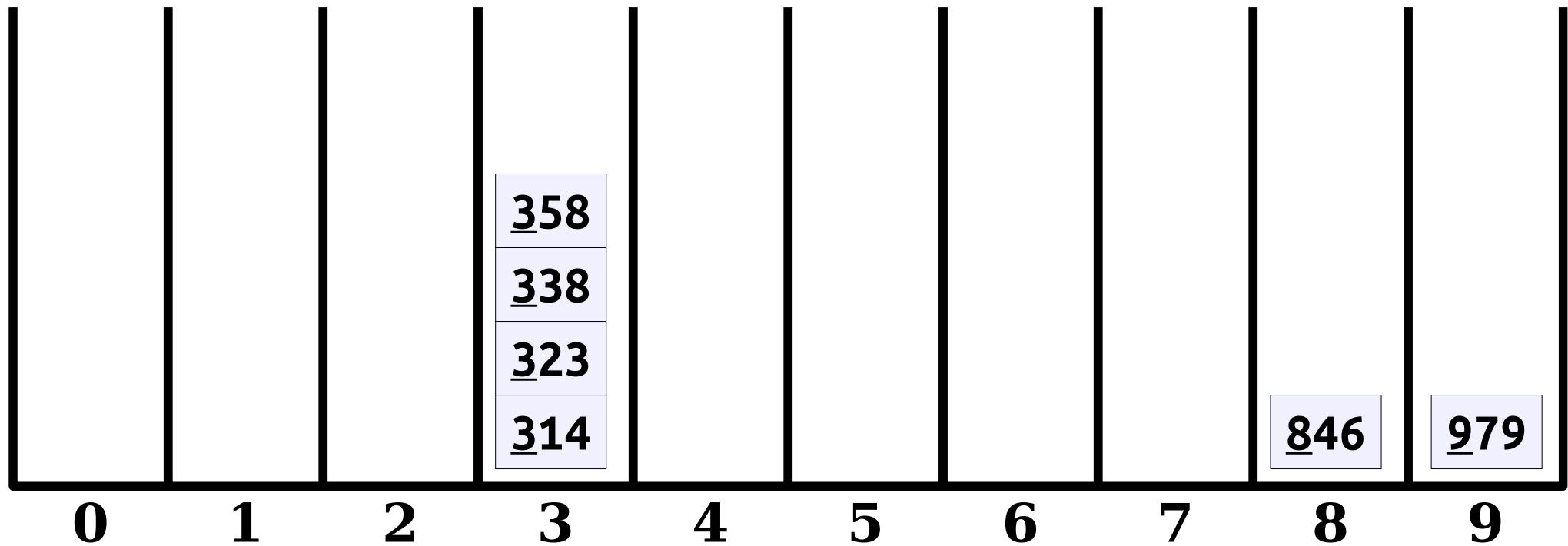
159

Radix Sort



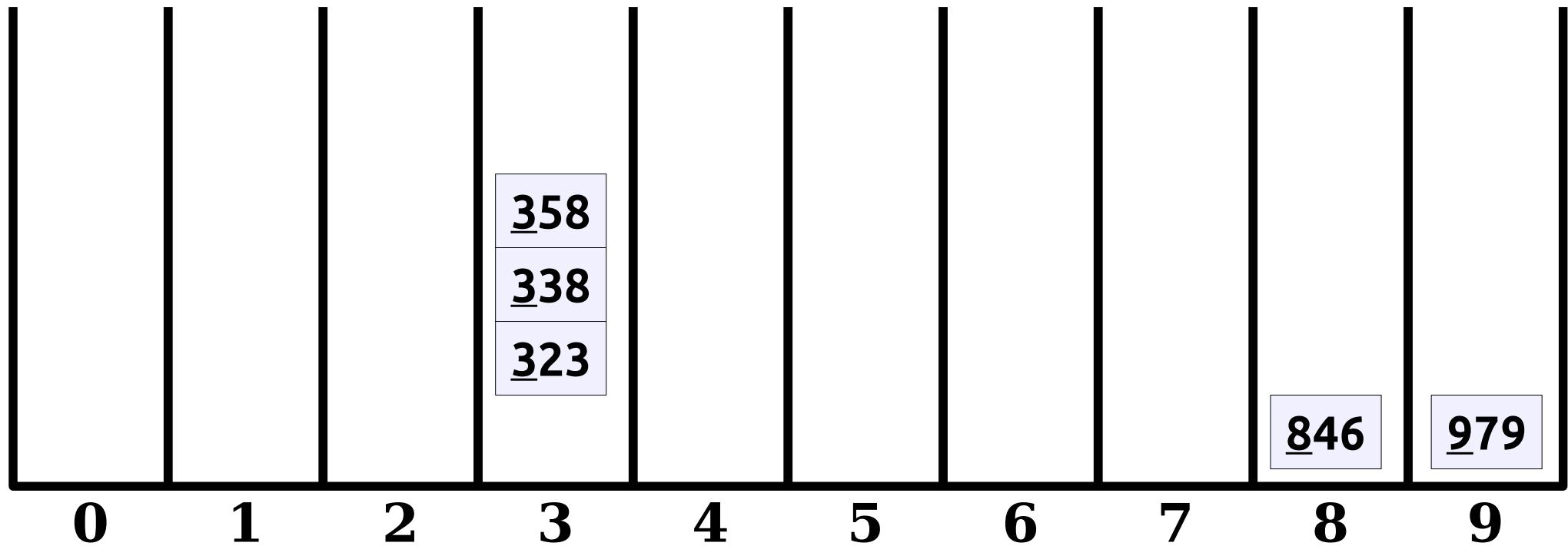
<u>159</u>	<u>264</u>
------------	------------

Radix Sort



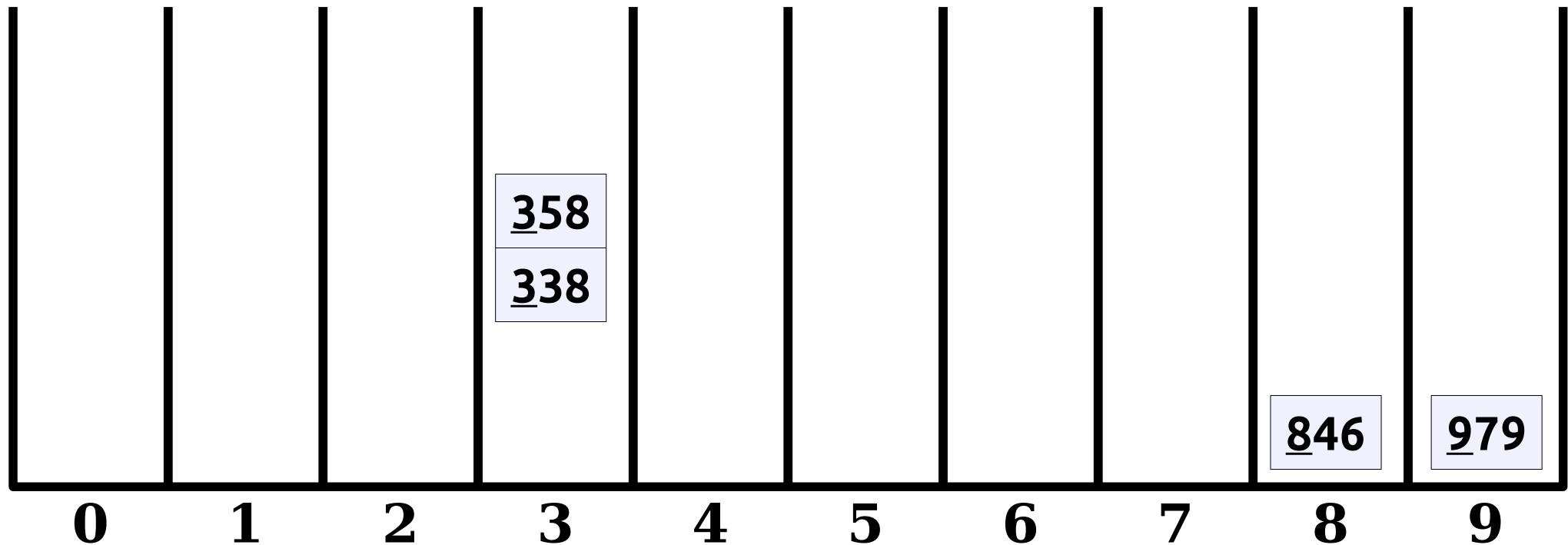
<u>159</u>	<u>264</u>	<u>265</u>
------------	------------	------------

Radix Sort



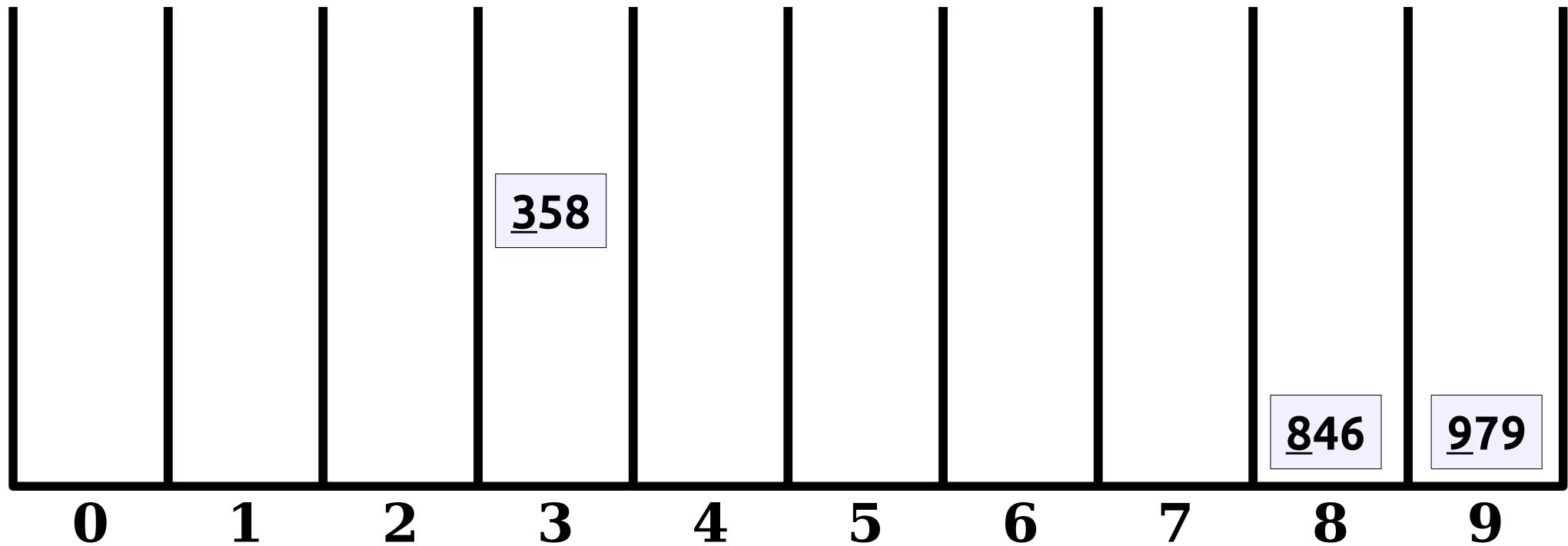
<u>159</u>	<u>264</u>	<u>265</u>	<u>314</u>
------------	------------	------------	------------

Radix Sort



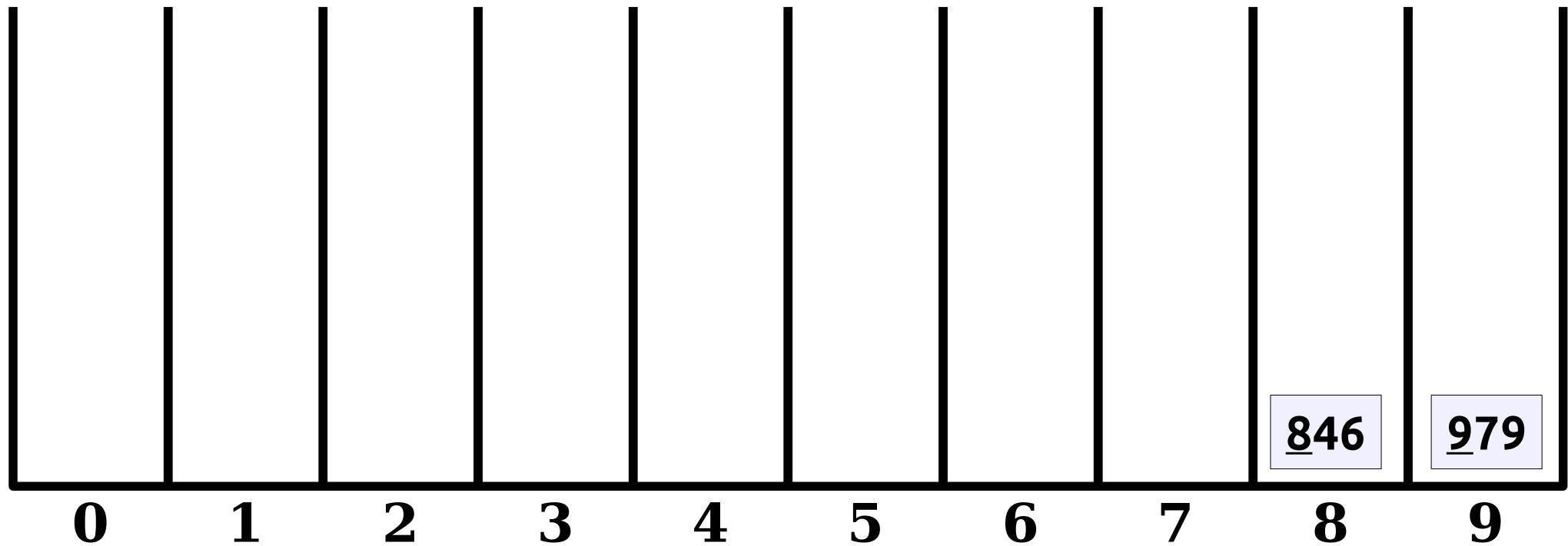
<u>159</u>	<u>264</u>	<u>265</u>	<u>314</u>	<u>323</u>
------------	------------	------------	------------	------------

Radix Sort



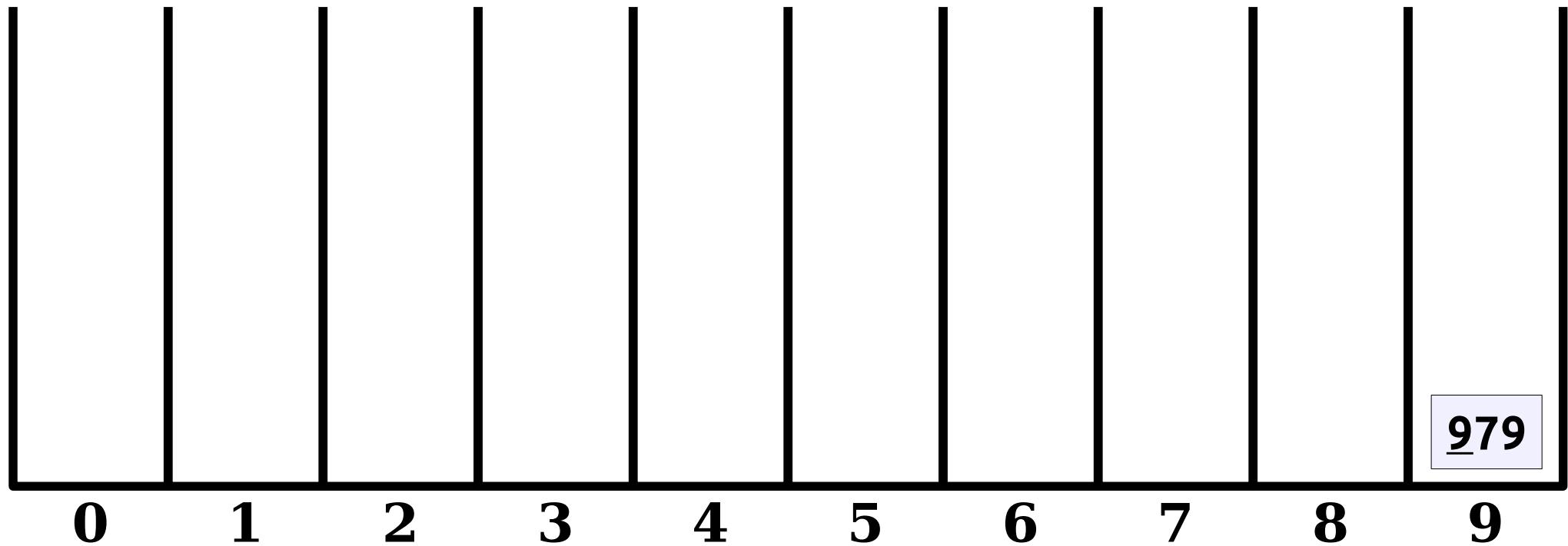
<u>159</u>	<u>264</u>	<u>265</u>	<u>314</u>	<u>323</u>	<u>338</u>
------------	------------	------------	------------	------------	------------

Radix Sort



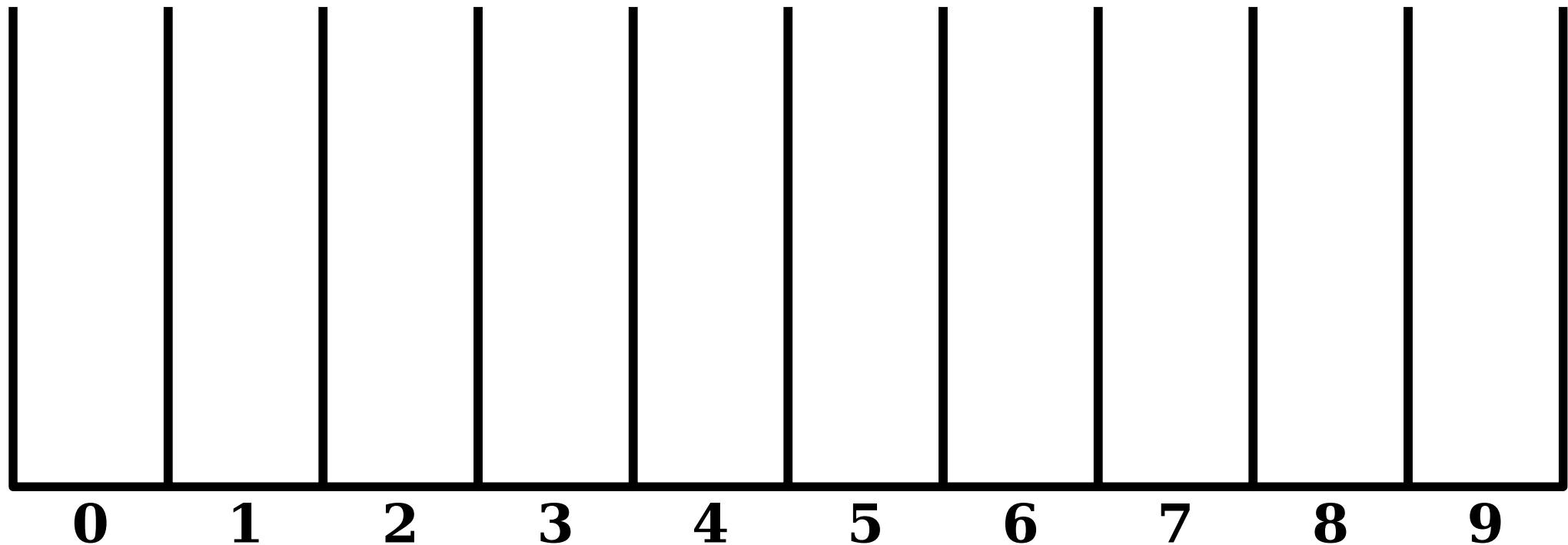
<u>159</u>	<u>264</u>	<u>265</u>	<u>314</u>	<u>323</u>	<u>338</u>	<u>358</u>
------------	------------	------------	------------	------------	------------	------------

Radix Sort



<u>159</u>	<u>264</u>	<u>265</u>	<u>314</u>	<u>323</u>	<u>338</u>	<u>358</u>	<u>846</u>
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

Radix Sort



<u>1</u> 59	<u>2</u> 64	<u>2</u> 65	<u>3</u> 14	<u>3</u> 23	<u>3</u> 38	<u>3</u> 58	<u>8</u> 46	<u>9</u> 79
-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------	-------------

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: **$O(k(t + |\Sigma|))$** .

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 $\Theta(m^2 + m|\Sigma|)$
- 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 $0, 1, 2, \dots, |\Sigma| - 1$.
- If this isn't the case, we can always sort the letters and replace each with its rank.
- Assuming that $|\Sigma| = O(1)$, this doesn't affect the runtime.

Some Terminology

- Define T_k to be the positions in T whose indices are equal to $k \bmod 3$.
 - 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	\$
---	---	---	---	---	---	---	---	---	---	---	---	---	---	----

Some Terminology

- Define T_k to be the positions in T whose indices are equal to $k \bmod 3$.
 - 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 .



monsoon monsoon \$

T_0

Some Terminology

- Define T_k to be the positions in T whose indices are equal to $k \bmod 3$.
 - 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 .

monsoon monsoon \$

T_1

Some Terminology

- Define T_k to be the positions in T whose indices are equal to $k \bmod 3$.
 - 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 \$

T_2

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.

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

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

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

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

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

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

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

A 15x15 grid of letters. The first row contains the letters 'M', 'N', 'O', 'M', 'S', '\$'. All other rows and columns consist entirely of the dollar sign character '\$'.

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.

m	o	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.



m o n s o o n 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.

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.

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.

The image shows a horizontal sequence of colored squares. Each square contains a character from the string 'onsoonnomnoms\$\$nsoonnomnoms\$\$\$\$'. The colors of the squares are as follows: light blue, purple, light blue, purple, light blue, purple, light blue, purple, yellow, light blue, purple, light blue, purple, yellow, light blue, purple, yellow, light blue, purple. This pattern repeats three times, followed by a final group of four squares: light blue, purple, yellow, yellow. The colors are used to visually distinguish between different characters or segments of the string.

onsoonnomnoms\$\$nsoonnomnoms\$\$\$\$

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 .

onsoonnomnoms\$ \$ \$ nsoonnomnoms\$ \$ \$

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	\$	\$	\$	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.



onsoonnonnomnoms\$\$\$ nssoonnonnomnoms\$\$\$\$

So, Um...

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

onsoonnomnom \$\$ onsoonnomnom \$\$

Playing with Blocks

- ***Key Insight:*** Break the resulting string apart into blocks of size three.

onsooonnomnoms\$\$\$\$onsooonnomnoms\$\$\$\$

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.

onsooonnomnoms\$\$nsoonnomnoms\$\$\$\$

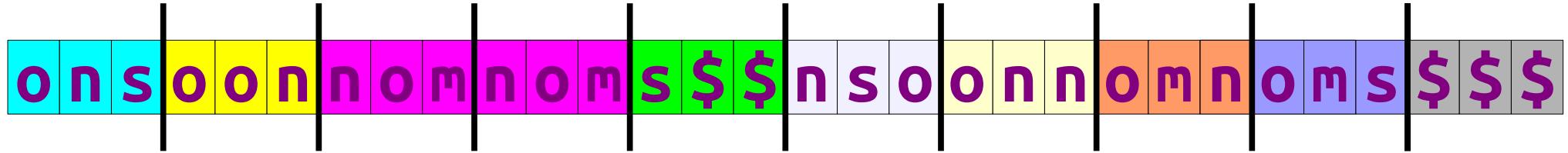
The Recursive Step

- ***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.

A horizontal string of characters: "onsoonnoonoms\$\$\$". Vertical black lines are placed at the start and end of each block of three characters. The string is divided into the following blocks: "on", "soo", "n", "no", "on", "o", "m", "n", "o", "m", "s", "\$", "\$", "\$". The characters are colored purple, and the background of the grid is light purple.

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.

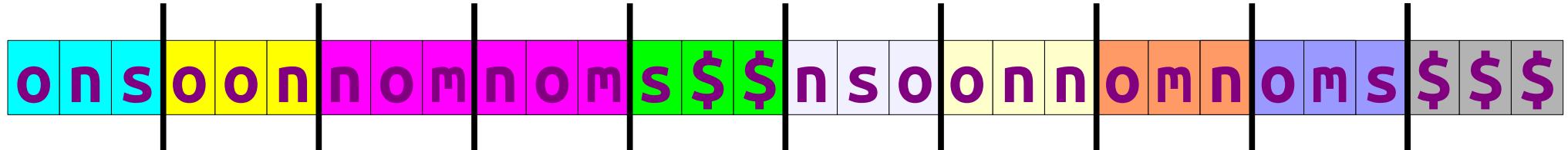


The Recursive Step

- ***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.



onsoonnomnoms\$\$\$



on so oo n nom no m noms \$ \$\$ \$ \$ \$

The Recursive Step

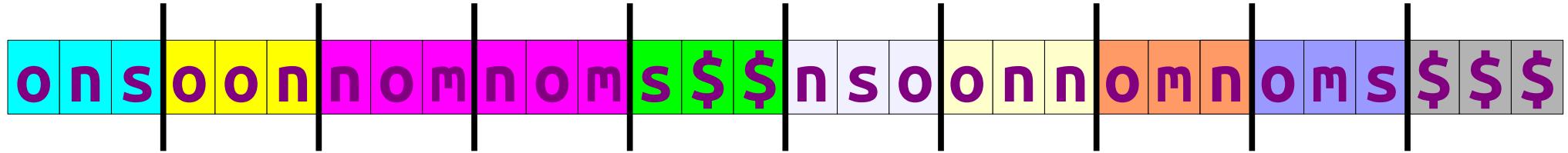
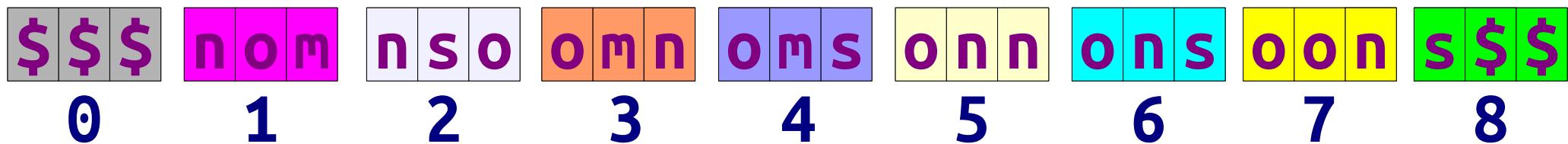
- ***The Trick:*** Treat each block of three characters as its own character.
- Determine the relative ordering of those characters by an $O(m)$ -time radix sort.
- Replace each block of three characters with the rank of its “metacharacter.”
- Recursively compute the suffix array of the resulting string.

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

on so oo on no m n o m n o m s \$ \$ \$ n so oo on no m n o m n o m s \$ \$ \$

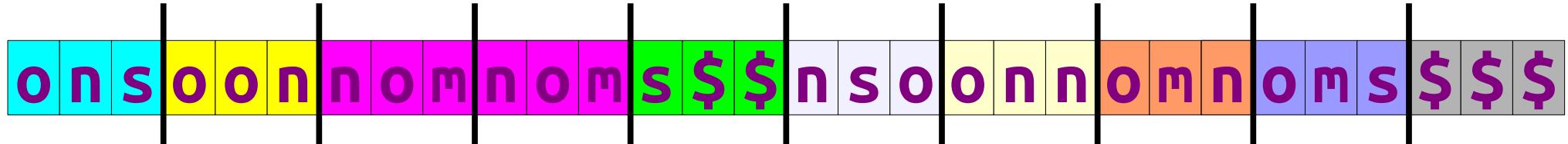
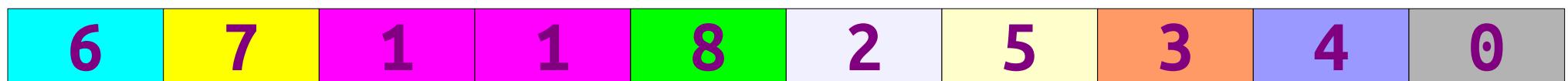
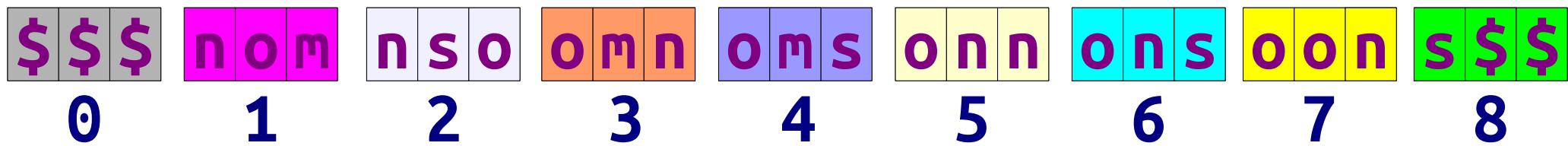
The Recursive Step

- ***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.



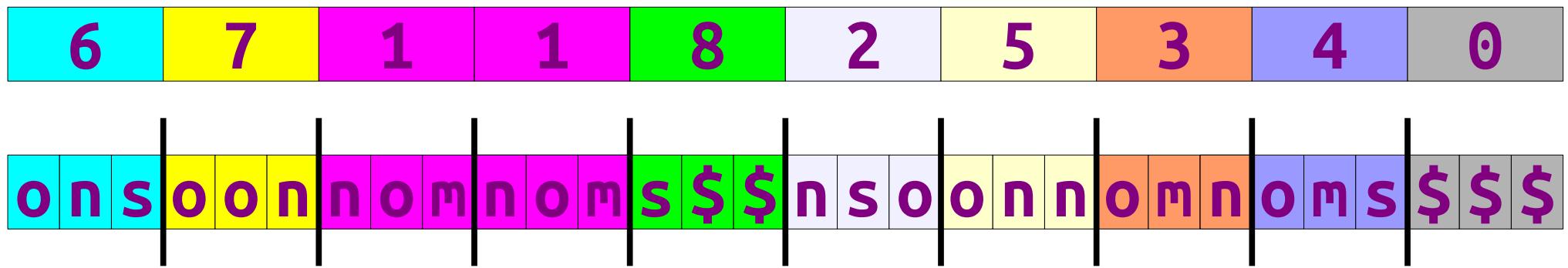
The Recursive Step

- ***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.



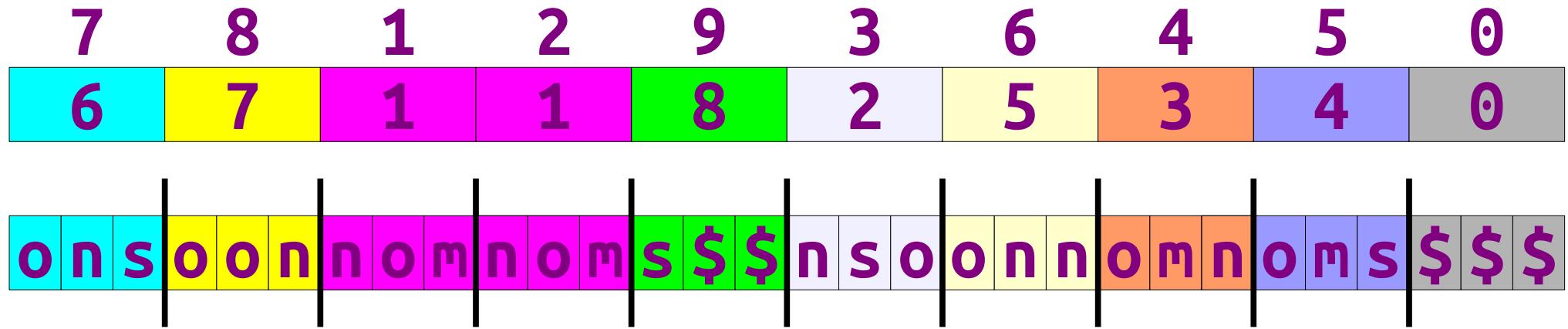
The Recursive Step

- ***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.



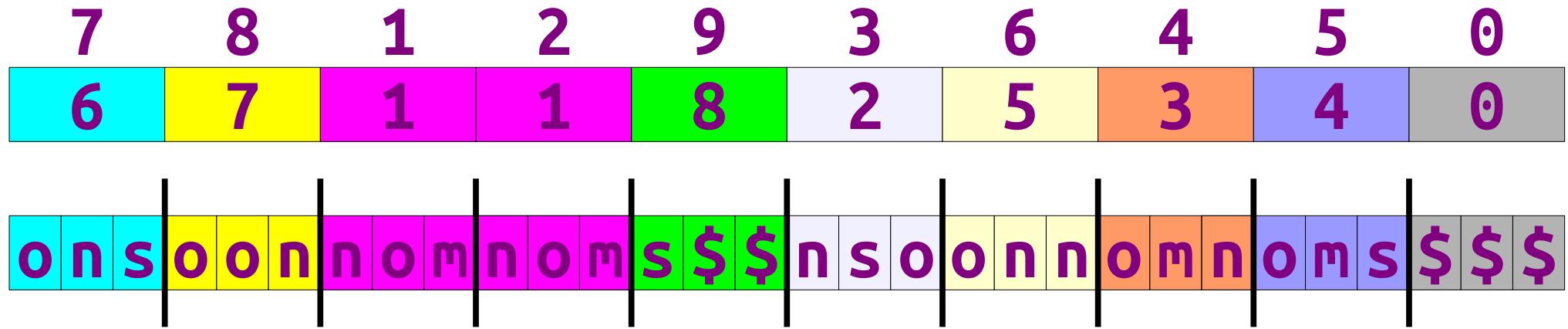
The Recursive Step

- ***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.



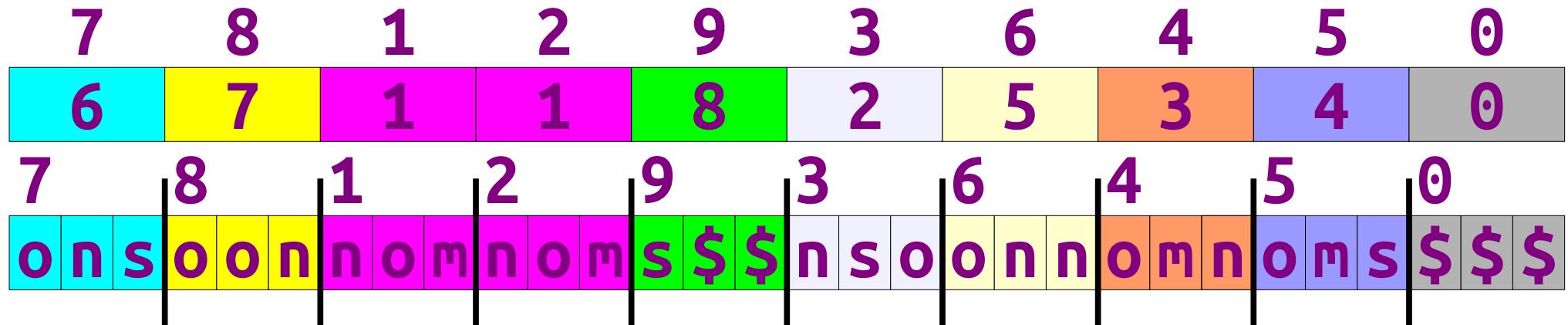
The Recursive Step

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



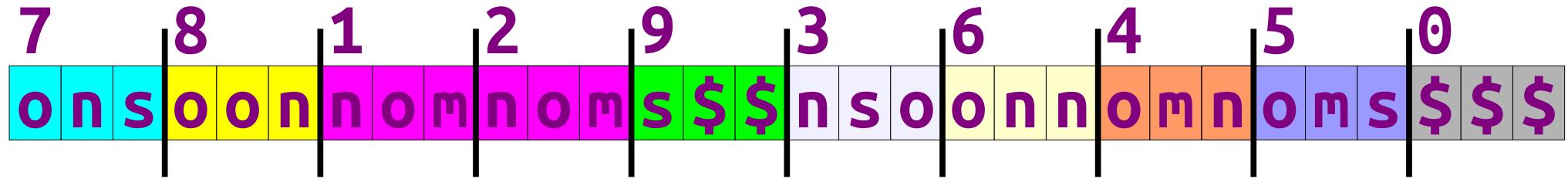
The Recursive Step

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



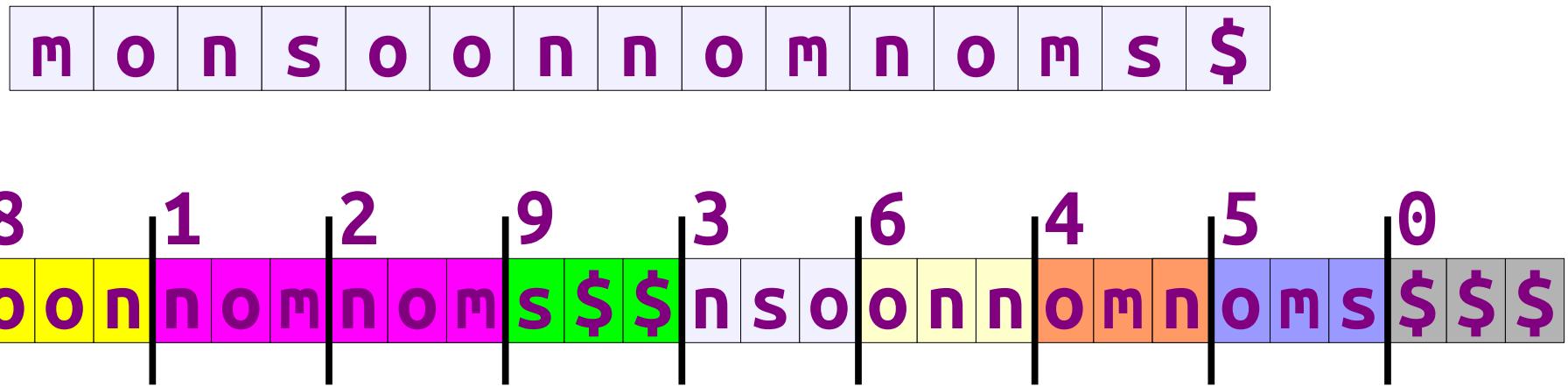
The Recursive Step

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



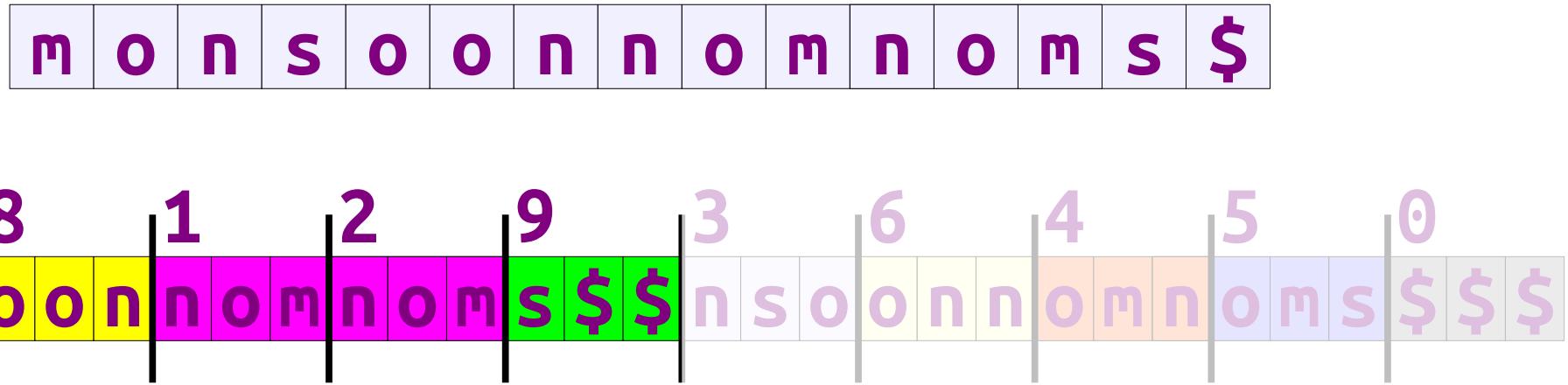
The Recursive Step

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



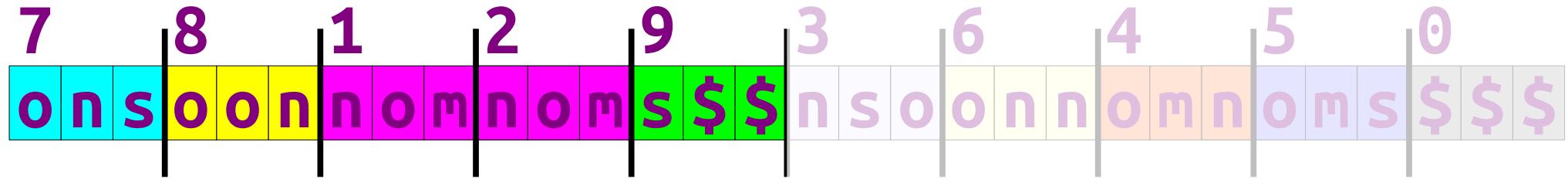
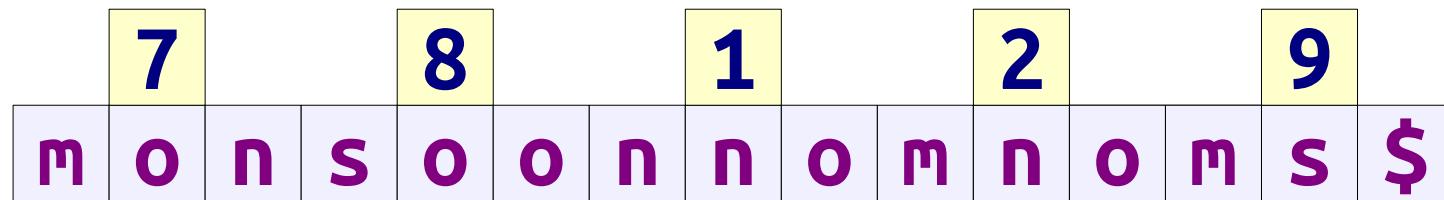
The Recursive Step

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



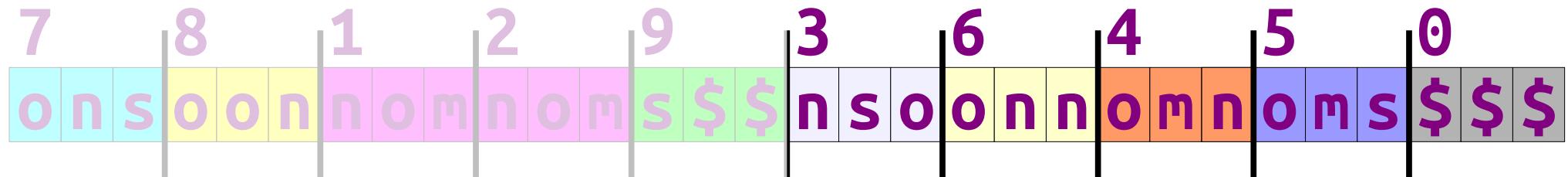
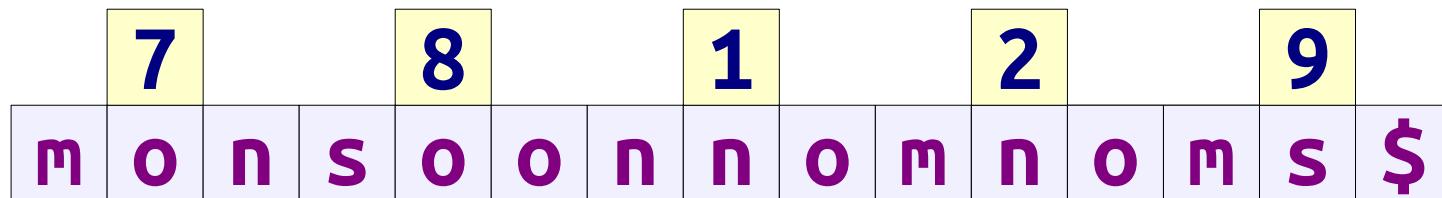
The Recursive Step

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



The Recursive Step

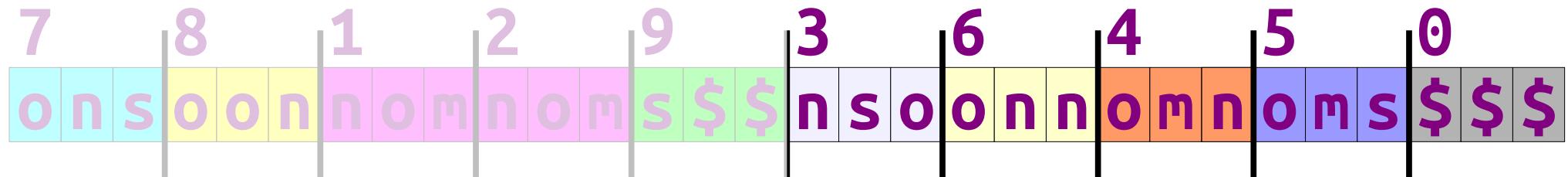
- Once we have this suffix array, we can use it to get the suffixes from T_1 and T_2 into sorted order.



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.

	7	3		8	6		1	4		2	5		9	0	
m	o	n	s	o	o	n	n	o	m	n	o	m	s	\$	



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.

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 .

	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!

	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	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	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	o	m	n
0	1	2	3	4	5	6	7	8	9

A diagram illustrating a modified radix sort. It shows two rows of 10 boxes each. The top row contains the digits 7, 3, 8, 6, 1, 4, 2, 5, 9, 0. The bottom row contains the letters m, o, n, s, o, o, n, o, m, n followed by a dollar sign \$. Below the boxes are indices 0 through 9 and letters A, B, C, D, E. The boxes are color-coded: the first two are purple, the next five are yellow, and the last three are light blue.

A Beautiful Insight

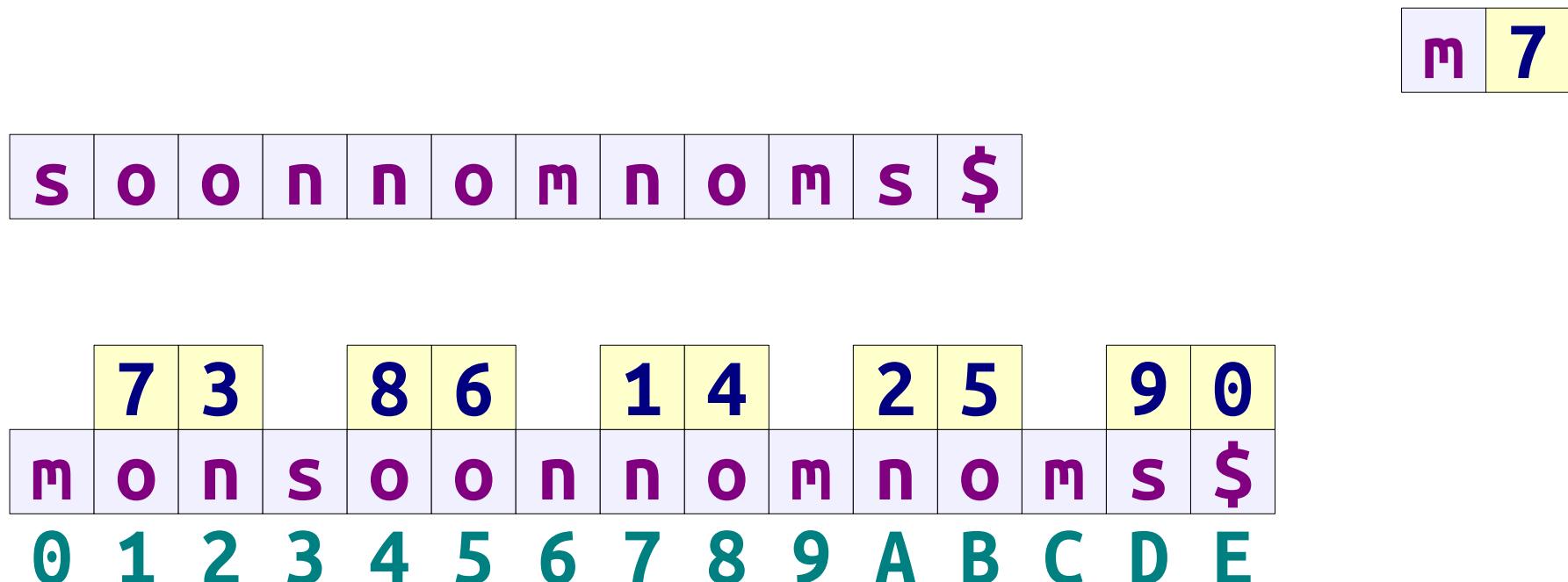
- **Claim:** If we know the relative ordering of suffixes at positions T_1 and T_2 , we can determine the relative order of suffixes in positions T_0 .
- **Idea:** Use a modified radix sort!

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!



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!

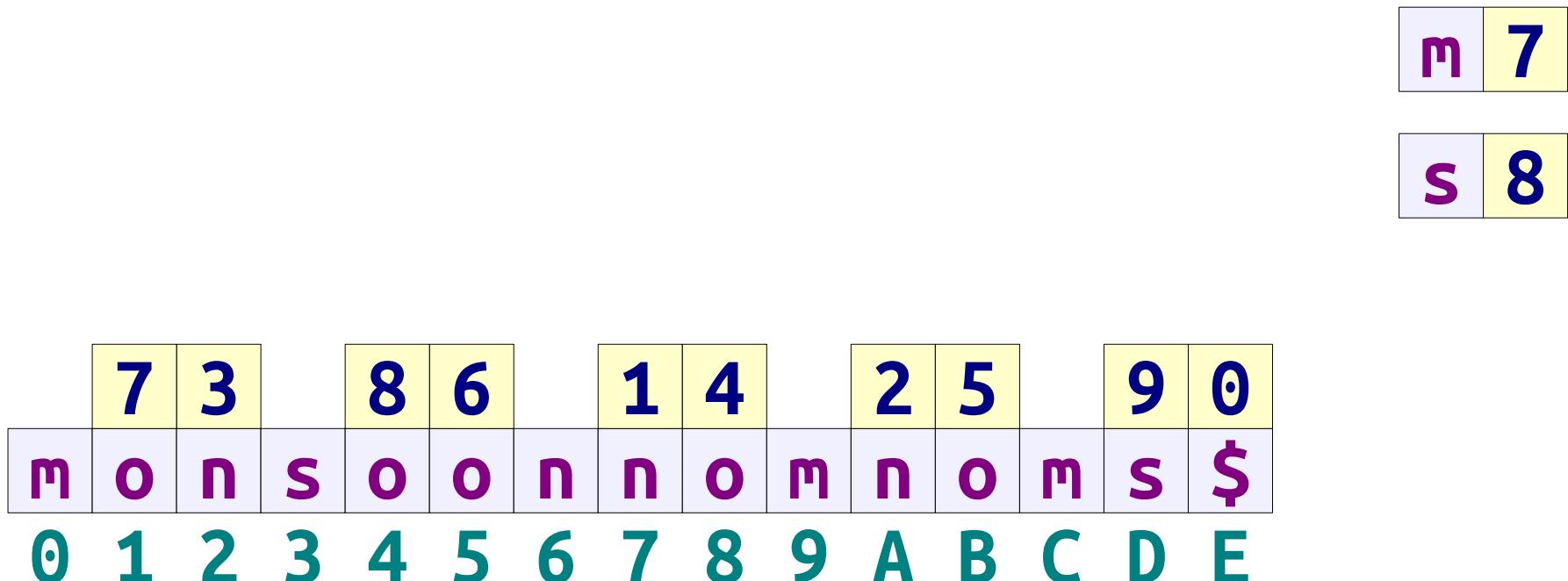
m	7
---	---

s	8
---	---

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

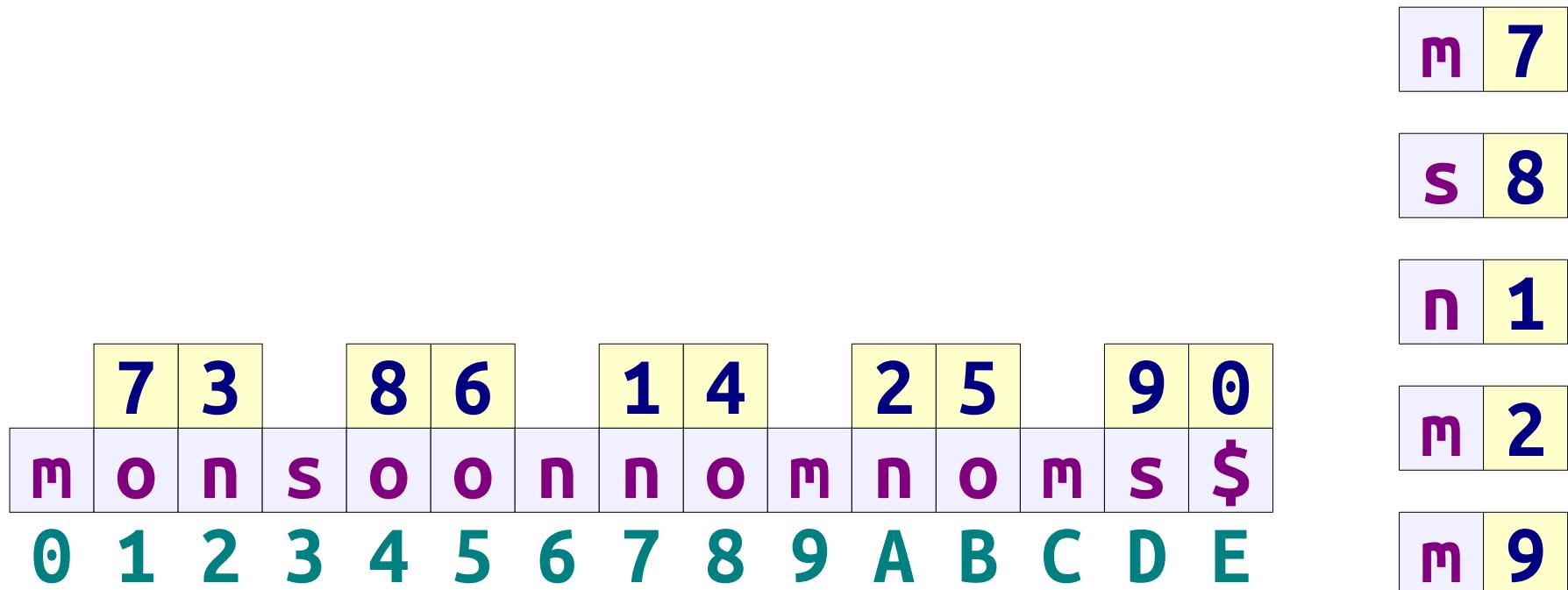
A Beautiful Insight

- **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!



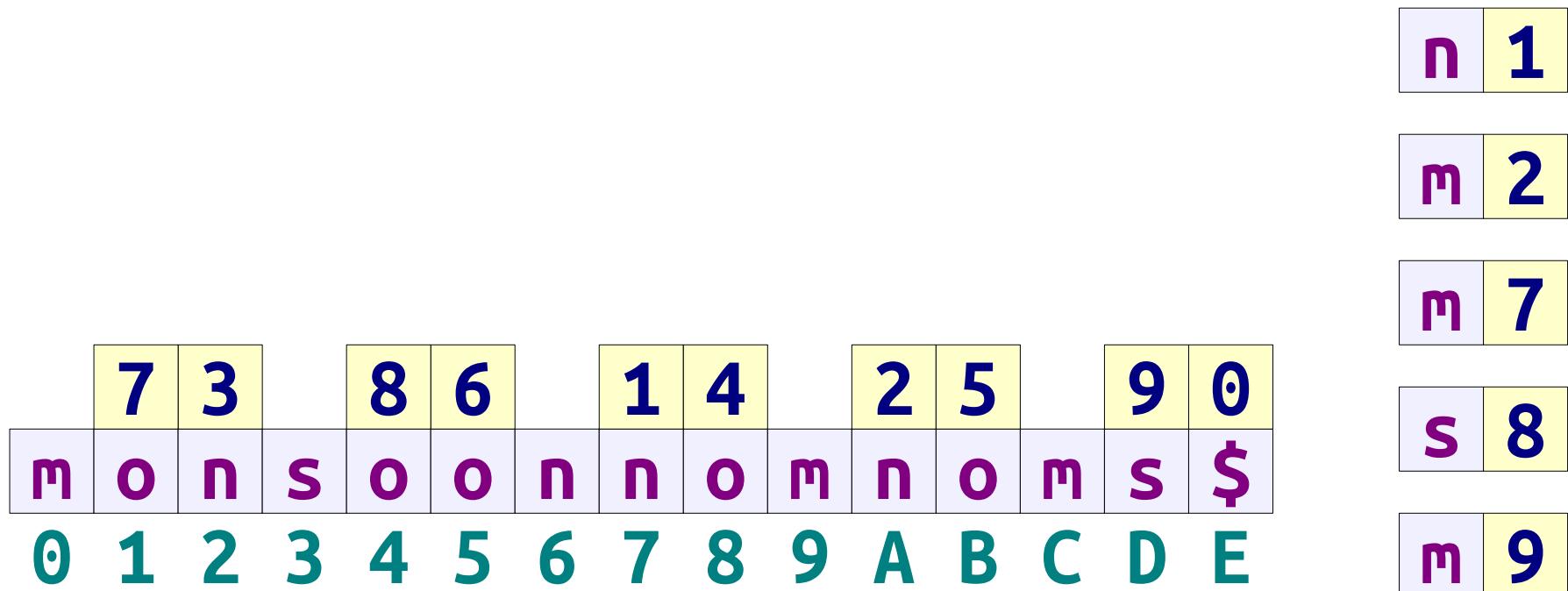
A Beautiful Insight

- **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!



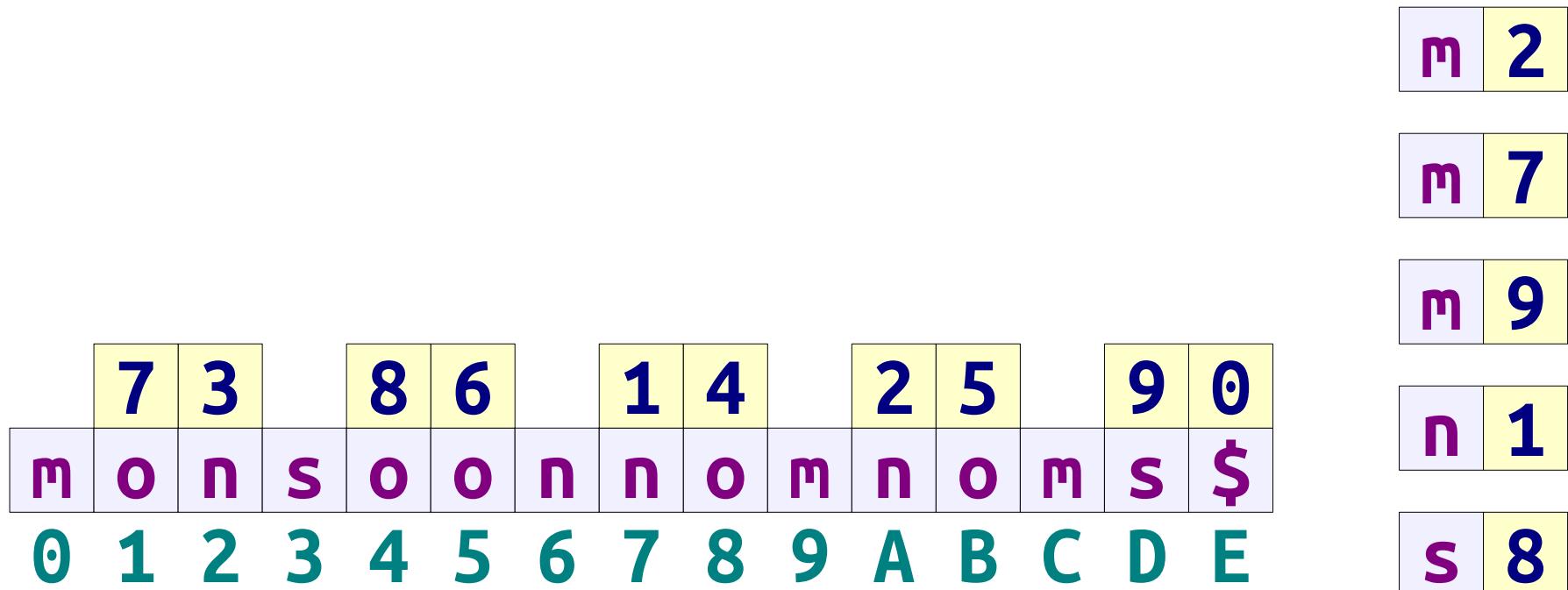
A Beautiful Insight

- **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!



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!



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

Relative ordering of suffixes at positions T_1 and T_2 :

- 2** (at position 1)
- 7** (at position 2)
- 9** (at position 3)
- 1** (at position 4)
- 8** (at position 5)
- 6** (at position 6)
- 3** (at position 7)
- 4** (at position 8)
- 0** (at position 9)
- 2** (at position A)
- 5** (at position B)
- 2** (at position C)
- 9** (at position D)
- 0** (at position E)

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 $\Sigma + m$.
 - 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.

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

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

9 0 C 6 3

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

9 0 C 6 3

E 7 A 2 8 B 5 1 4 D

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

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

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

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

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

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

E



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

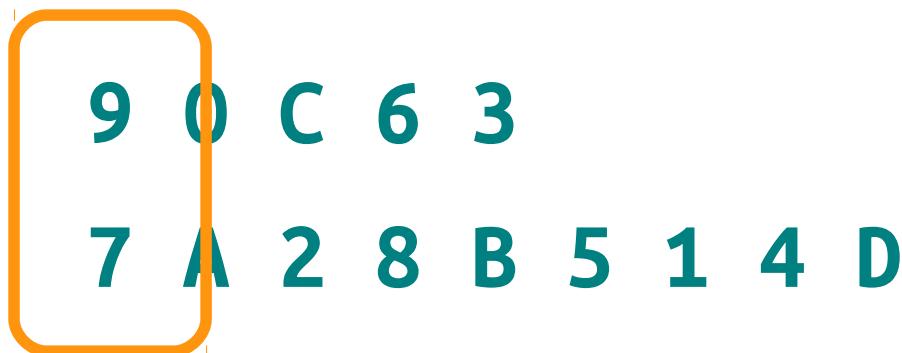
The Merging

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

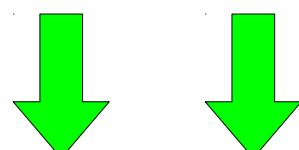
E

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

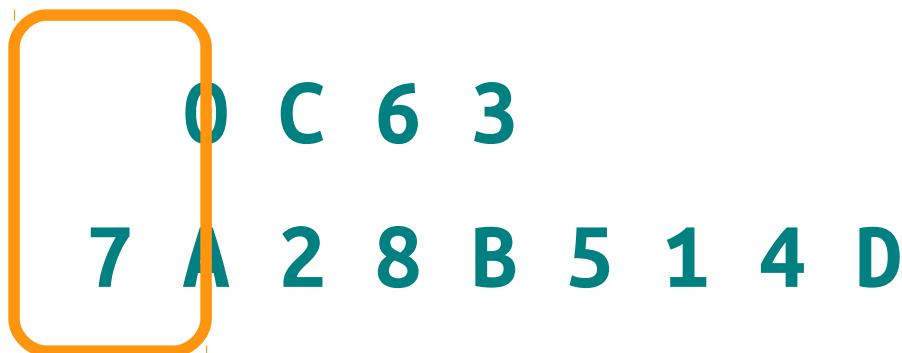


E



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

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

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

E 9

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

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

E 9

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

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

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

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

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

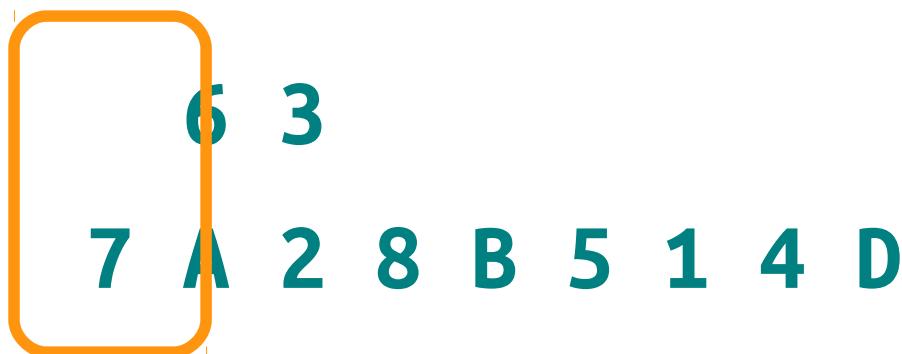
C 6 3
7 A 2 8 B 5 1 4 D

E 9 0



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



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

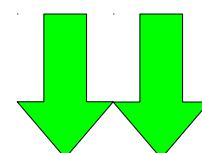
1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	\$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

6 3
7 A

2 8 B 5 1 4 D

E 9 0 C



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

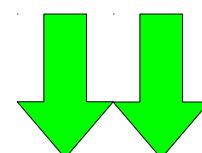
6	3
7	A

2 8 B 5 1 4 D

E 9 0 C

n 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

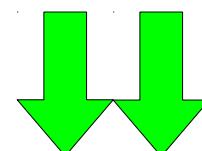
6	3
7	A

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

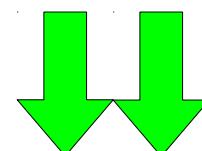
6 3
7 A

2 8 B 5 1 4 D

E 9 0 C

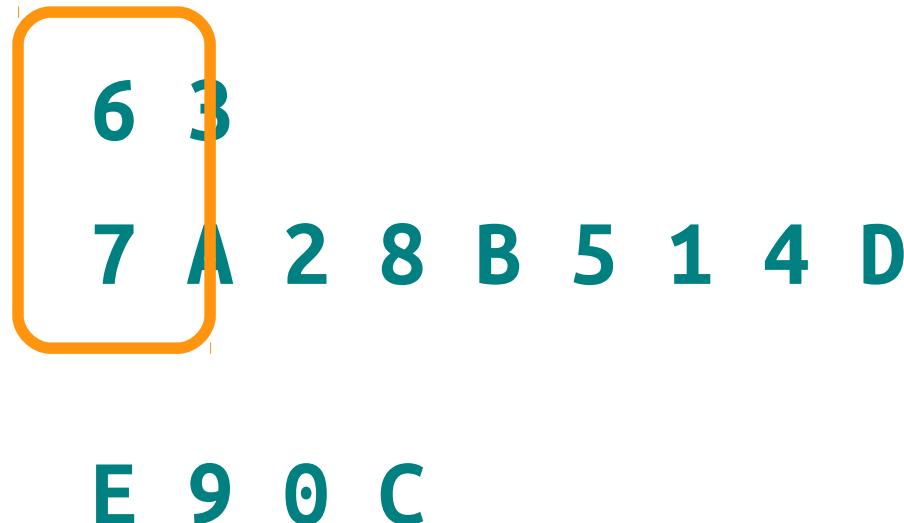
n 1

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

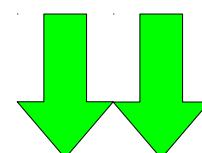
The Merging



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

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

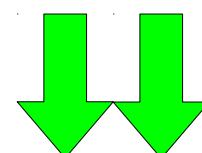
6	3
7	A

2 8 B 5 1 4 D

E 9 0 C

n	1
---	---

n	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

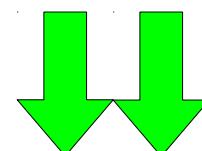
6 3
7 A

2 8 B 5 1 4 D

E 9 0 C

n 1

n 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

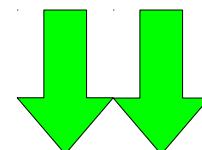
6 3
7 A

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

3
7 A

2 8 B 5 1 4 D

E 9 0 C 6

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

3
7 A

2 8 B 5 1 4 D

E 9 0 C 6

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

3
7 A

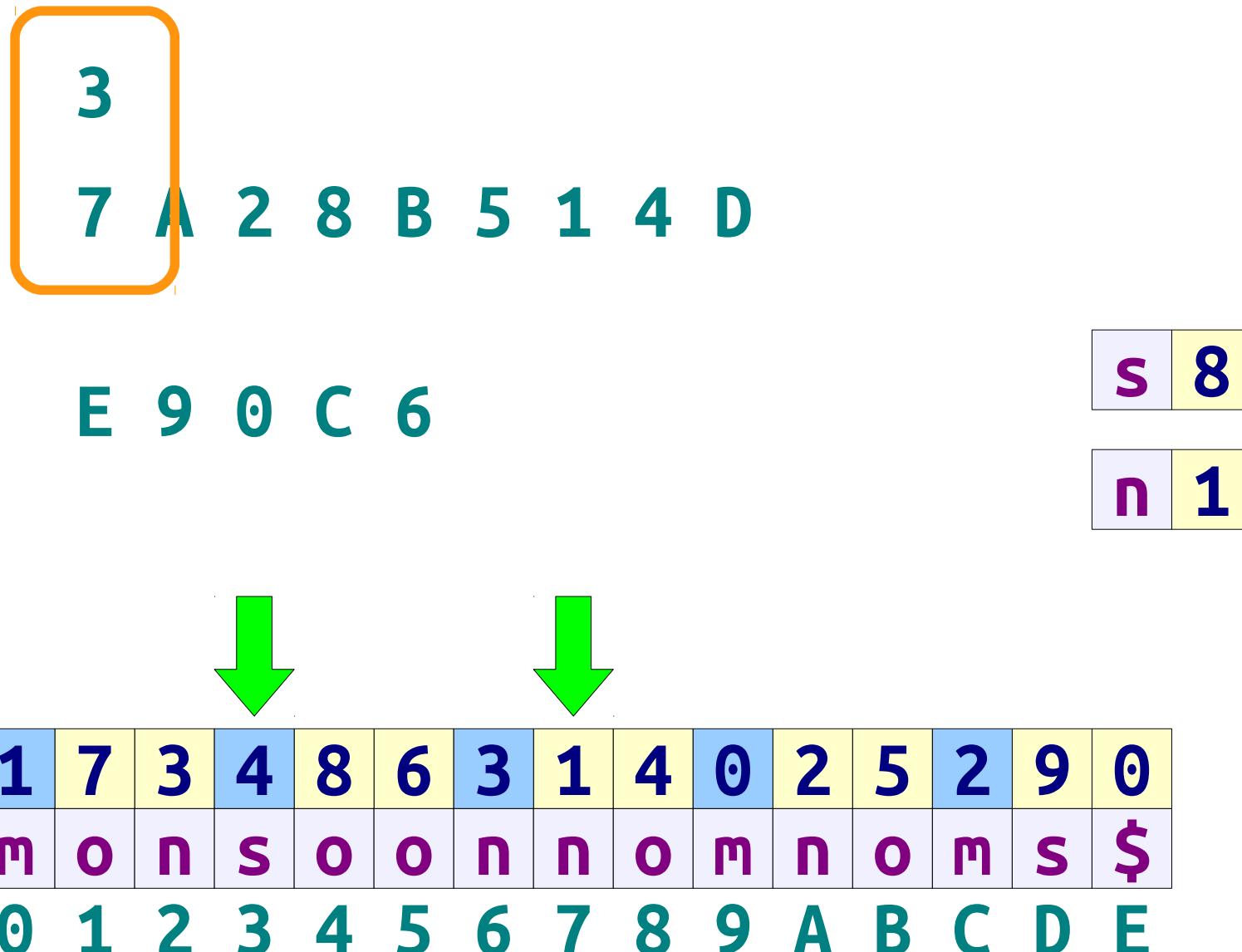
2 8 B 5 1 4 D

E 9 0 C 6

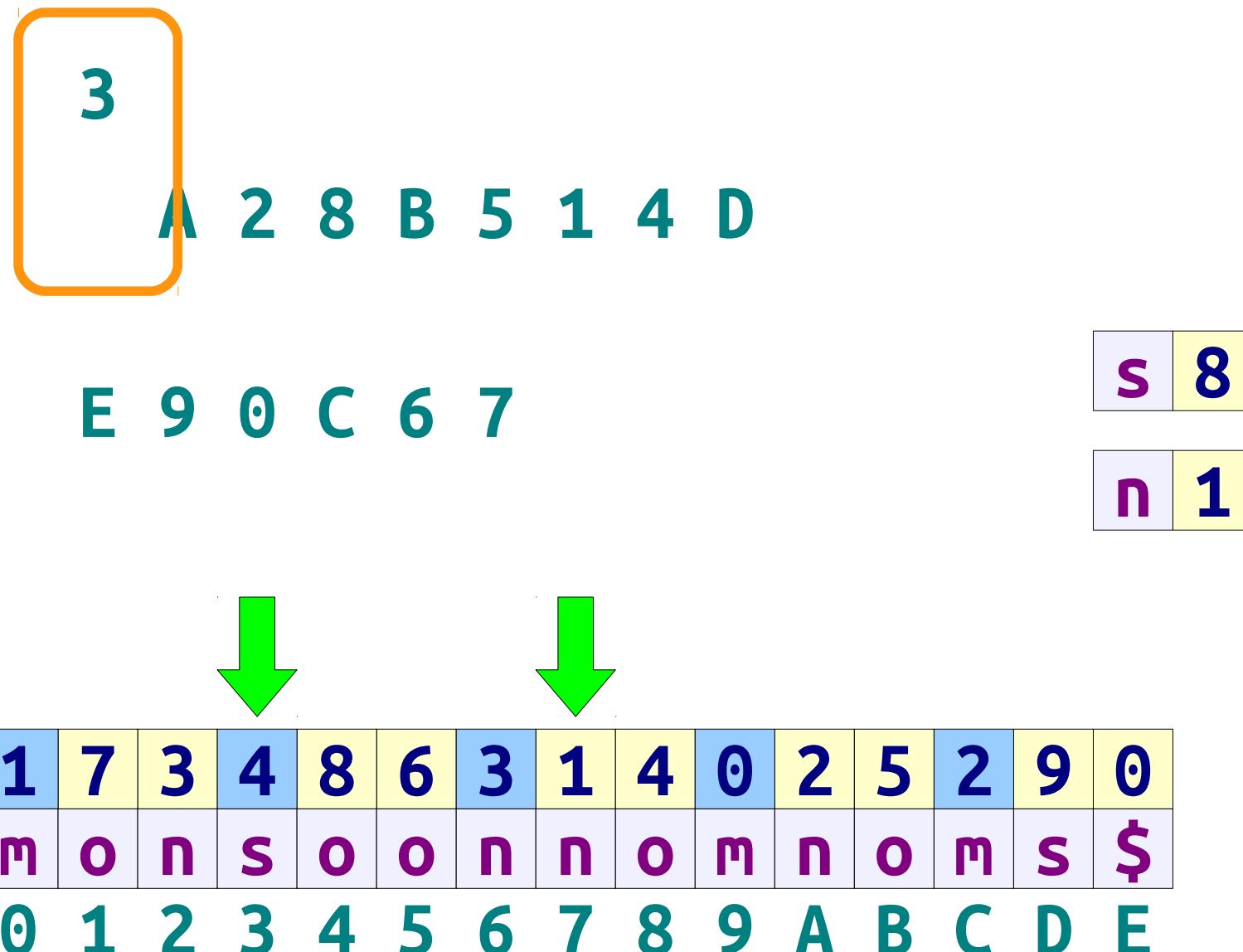


1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	\$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging



The Merging



The Merging

3
A 2 8 B 5 1 4 D

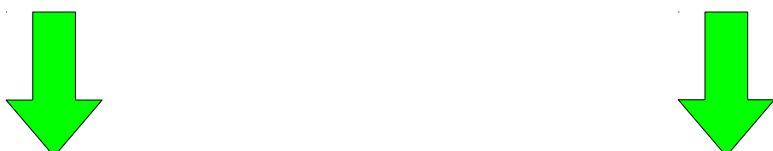
E 9 0 C 6 7

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

3
A 2 8 B 5 1 4 D

E 9 0 C 6 7



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

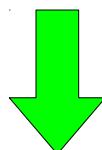
The Merging

3
A 2 8 B 5 1 4 D

E 9 0 C 6 7

s 8

n 5



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

3
2 8 B 5 1 4 D

E 9 0 C 6 7 A

s 8

n 5



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

3
2 8

B 5 1 4 D

E 9 0 C 6 7 A

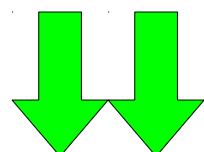
1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	\$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging

3
2 8

B 5 1 4 D

E 9 0 C 6 7 A



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

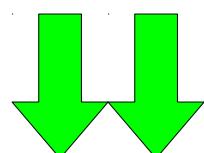
3
2 8

B 5 1 4 D

E 9 0 C 6 7 A

s 8

n 4

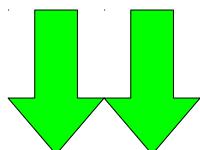


1	7	3	4	8	6	3	1	4	0	2	5	2	9	0
m	o	n	s	o	o	n	n	o	m	n	o	m	s	\$
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E

The Merging



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

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.

The Merging

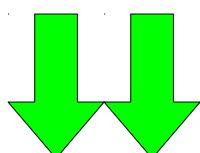
3
2 8

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

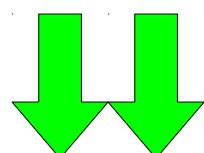
3
2 8

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

3
2 8

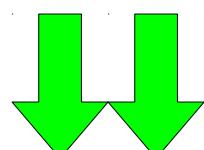
B 5 1 4 D

E 9 0 C 6 7 A

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

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

3

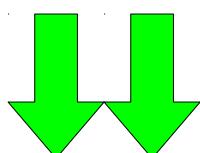
8 B 5 1 4 D

E 9 0 C 6 7 A 2

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

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

3
8 B

5 1 4 D

E 9 0 C 6 7 A 2

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

3
B 5 1 4 D

E 9 0 C 6 7 A 2 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

3
B 5

1 4 D

E 9 0 C 6 7 A 2 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

3
5 1 4 D

E 9 0 C 6 7 A 2 8 B

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

3
5 1 4 D

E 9 0 C 6 7 A 2 8 B

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

3
1 4 D

E 9 0 C 6 7 A 2 8 B 5

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

3
1 4 D

E 9 0 C 6 7 A 2 8 B 5

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

3
4 D

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

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

3
4 D

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

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

3
D

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

3
D

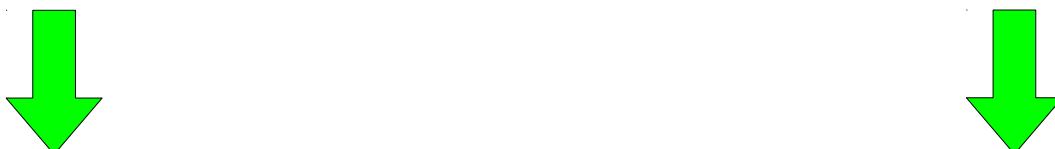
E 9 0 C 6 7 A 2 8 B 5 1 4

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

3
D

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

3
D

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

s 8

s 0



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

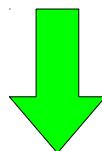
The Merging

3

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

s 8

s 0



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

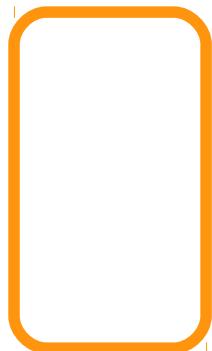
The Merging

3

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

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

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

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: **$O(m)$** .

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 **$\Theta(m)$** .

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 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.