Suffix Arrays

Problem Set Five is due in the box up front.

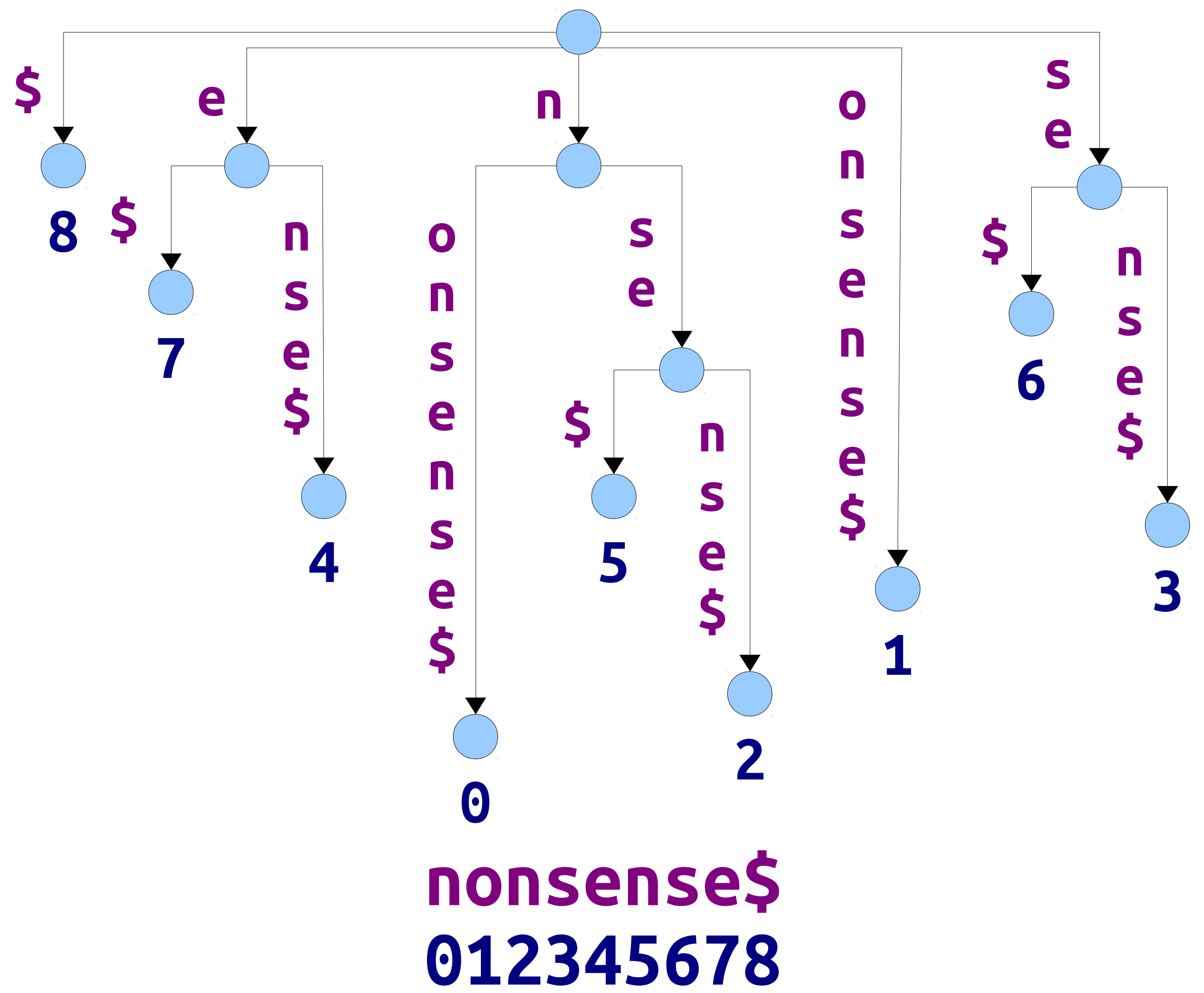
Outline for Today

- **Review from Last Time**
  - Quick review of suffix trees.
- **Suffix Arrays**
  - A space-efficient data structure for substring searching.
- **LCP Arrays**
  - A 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

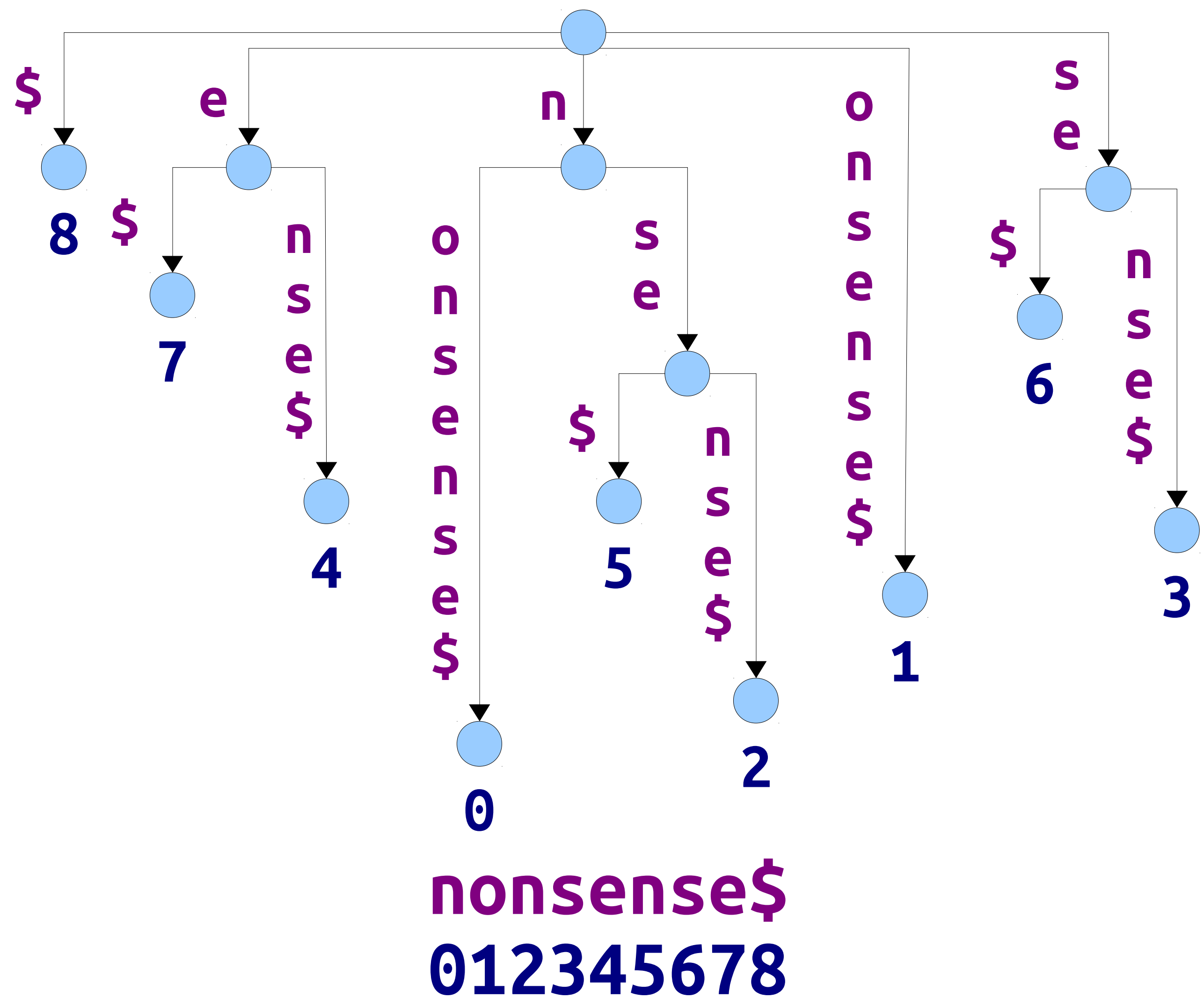
Suffix Trees

- 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, words for the start/end index and 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	onsense\$
6	se\$
3	sense\$

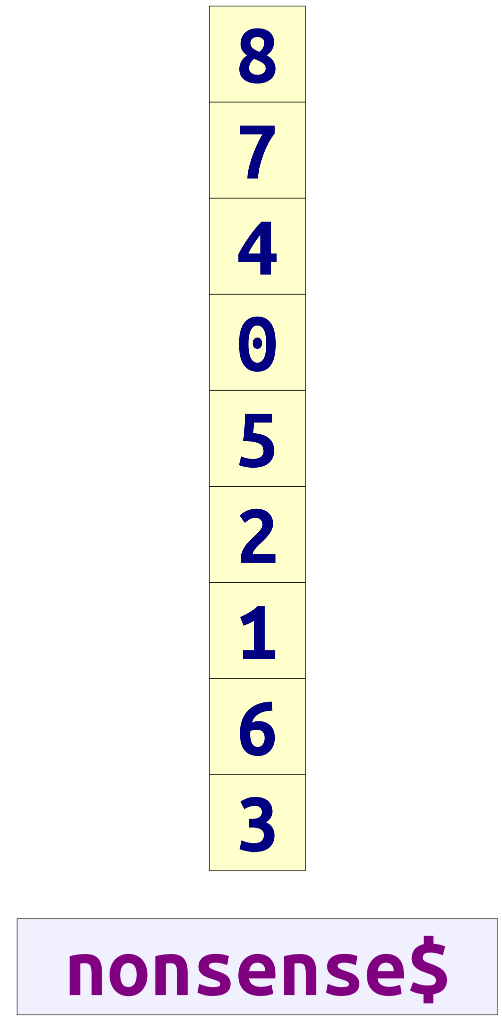
Representing Suffix Arrays

- Suffix arrays are typically stored as an array of the start positions of the suffixes.
- 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 stored as an array of the start positions of the suffixes.
- Space required: $\Theta(m)$.
- More precisely, space for $T\$$, plus one extra word for each character.



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$

nse

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$

nse

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$

nse

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$

nse

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$

nse

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$

nse

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$

nse

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$

nse

Analyzing the Runtime

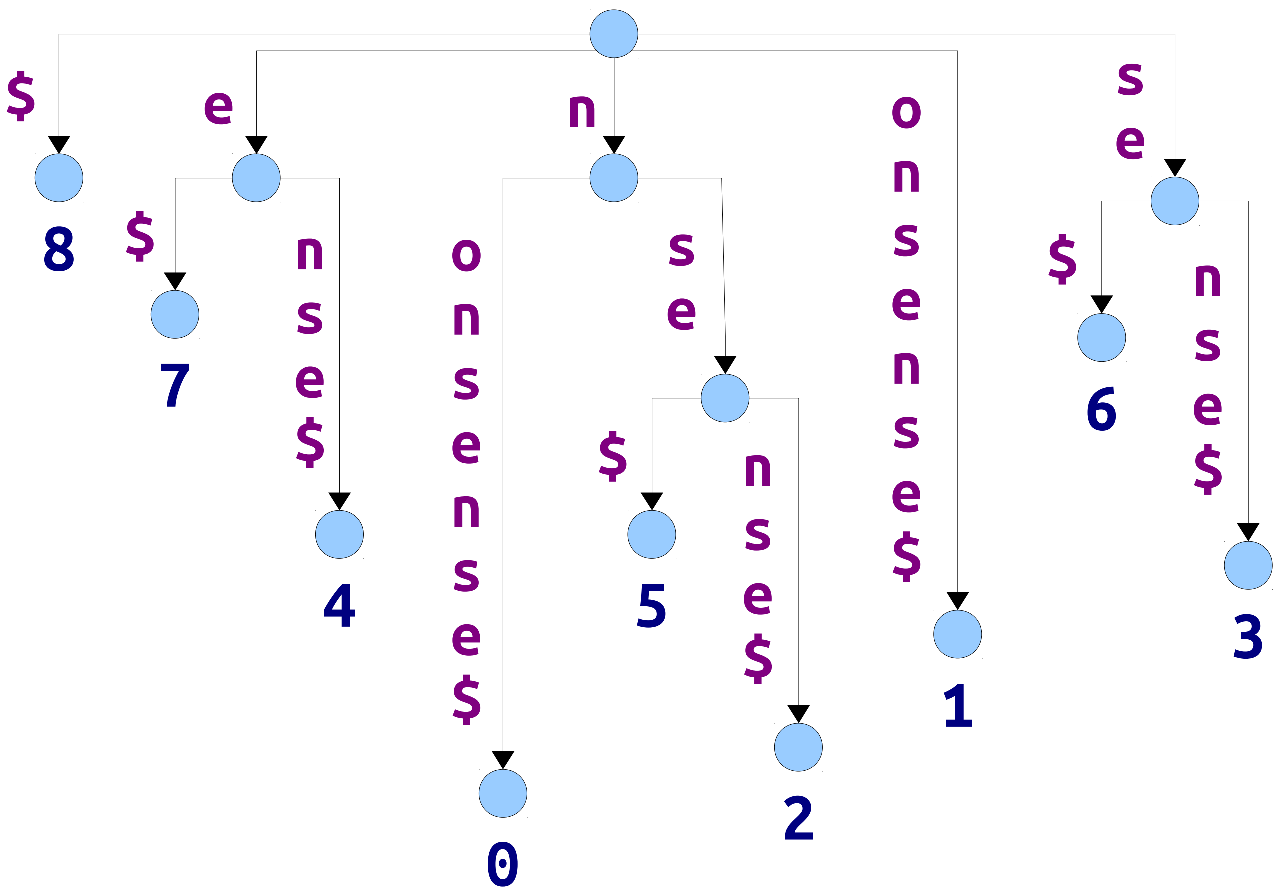
- The binary search will require O(log m) probes into the suffix array.
- Each comparison takes time O(n): have to compare P against the current suffix.
- Time for binary searching: O(n log m).
- Time to report all matches after that point: O(z).
- Total time: **O(n log m + z)**.

A Useful Observation

A Loss of Structure

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

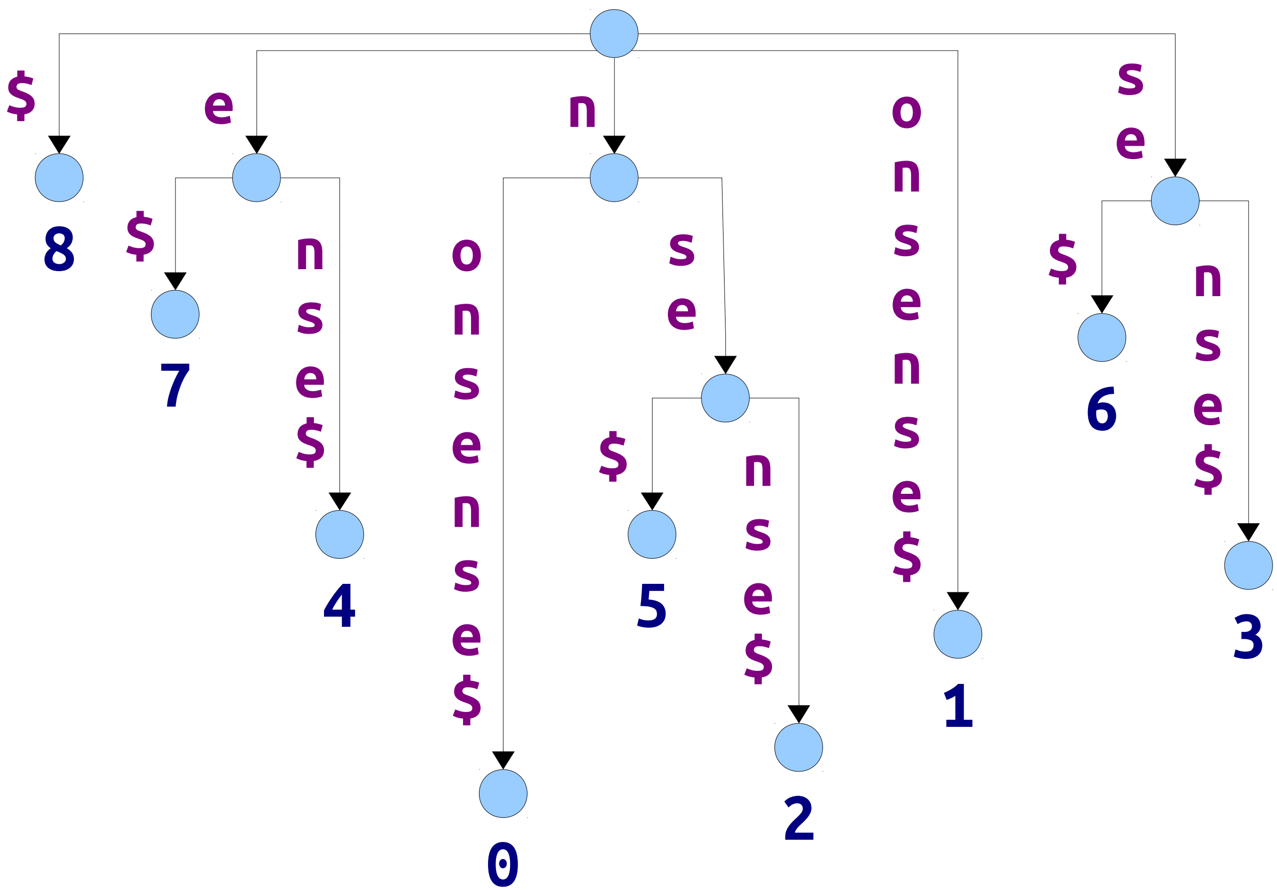
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

Suffix Trees and Suffix Arrays

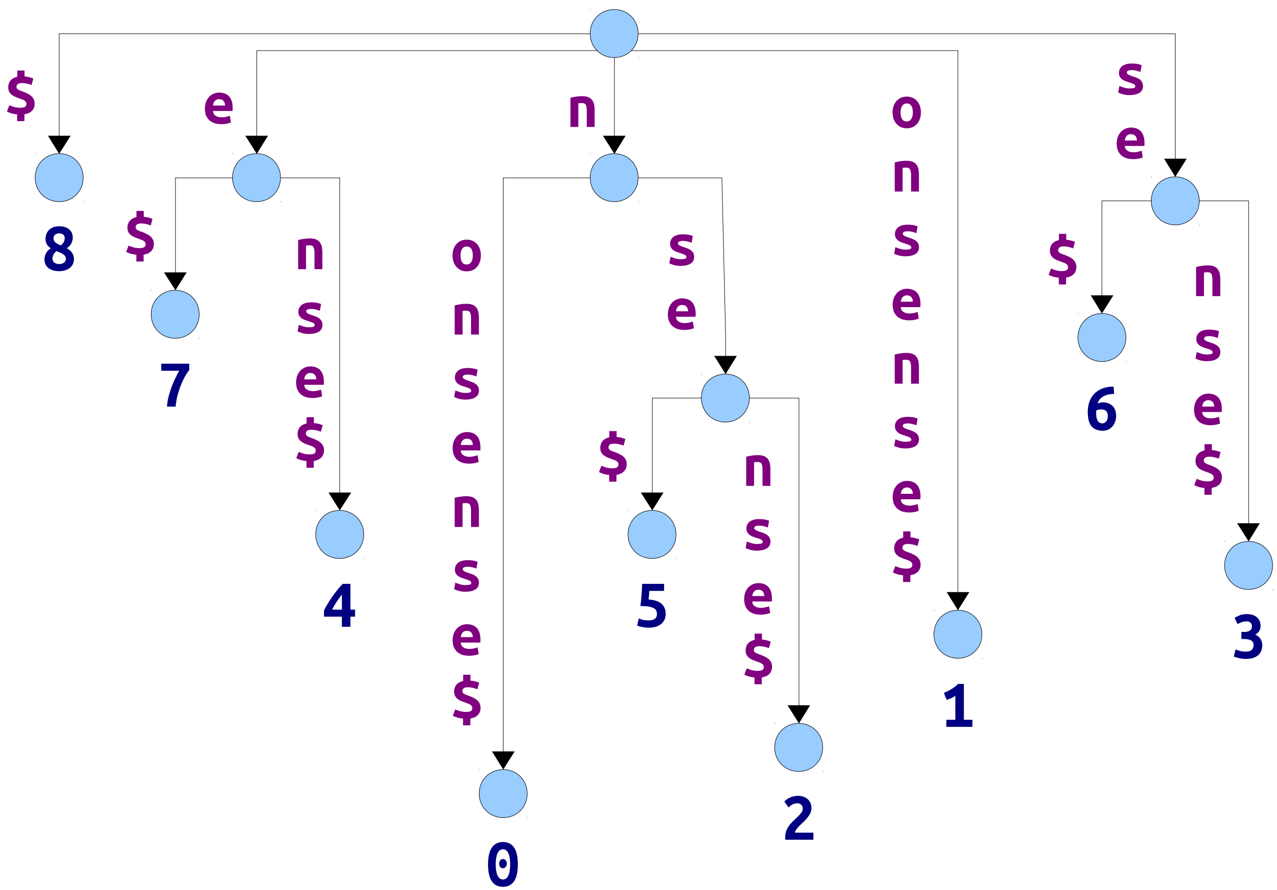


8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

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

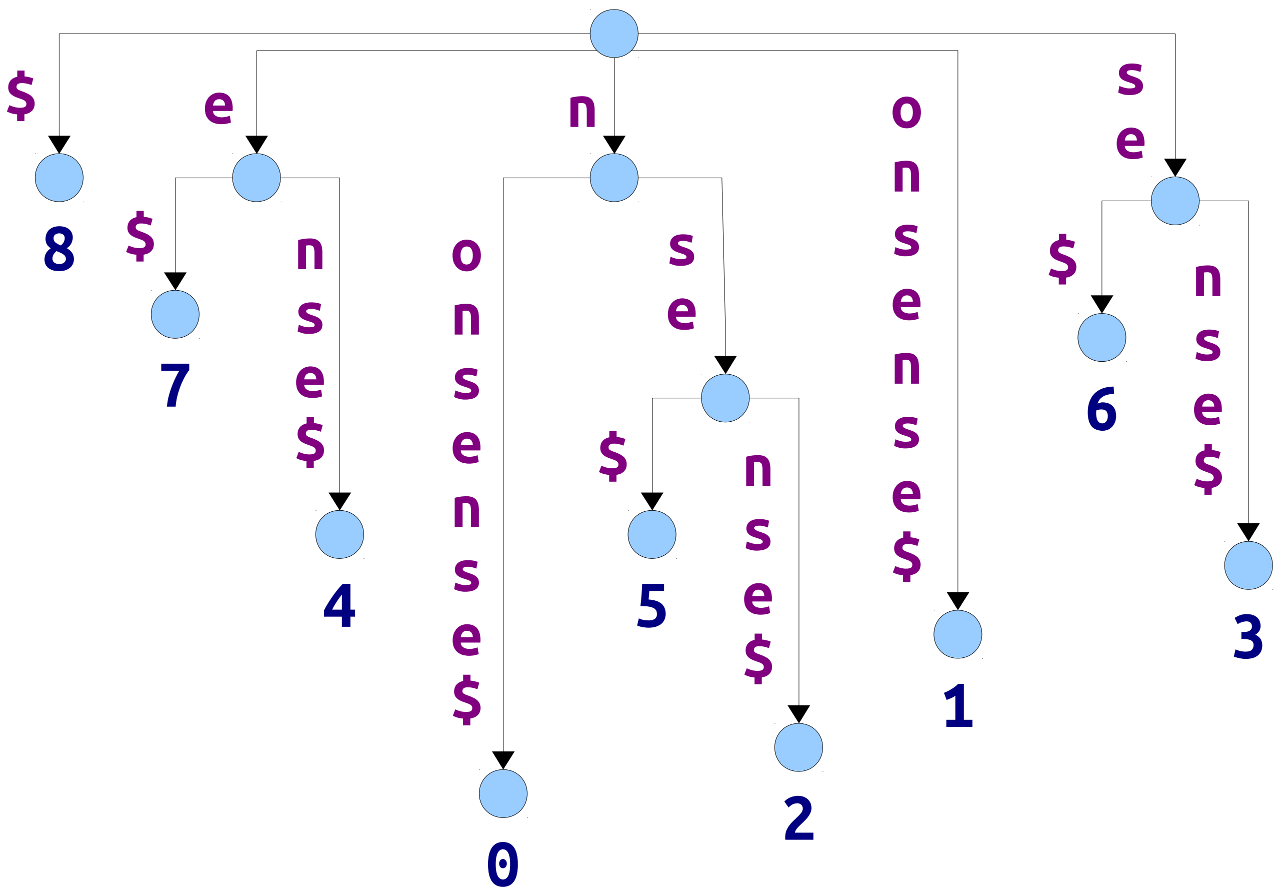
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

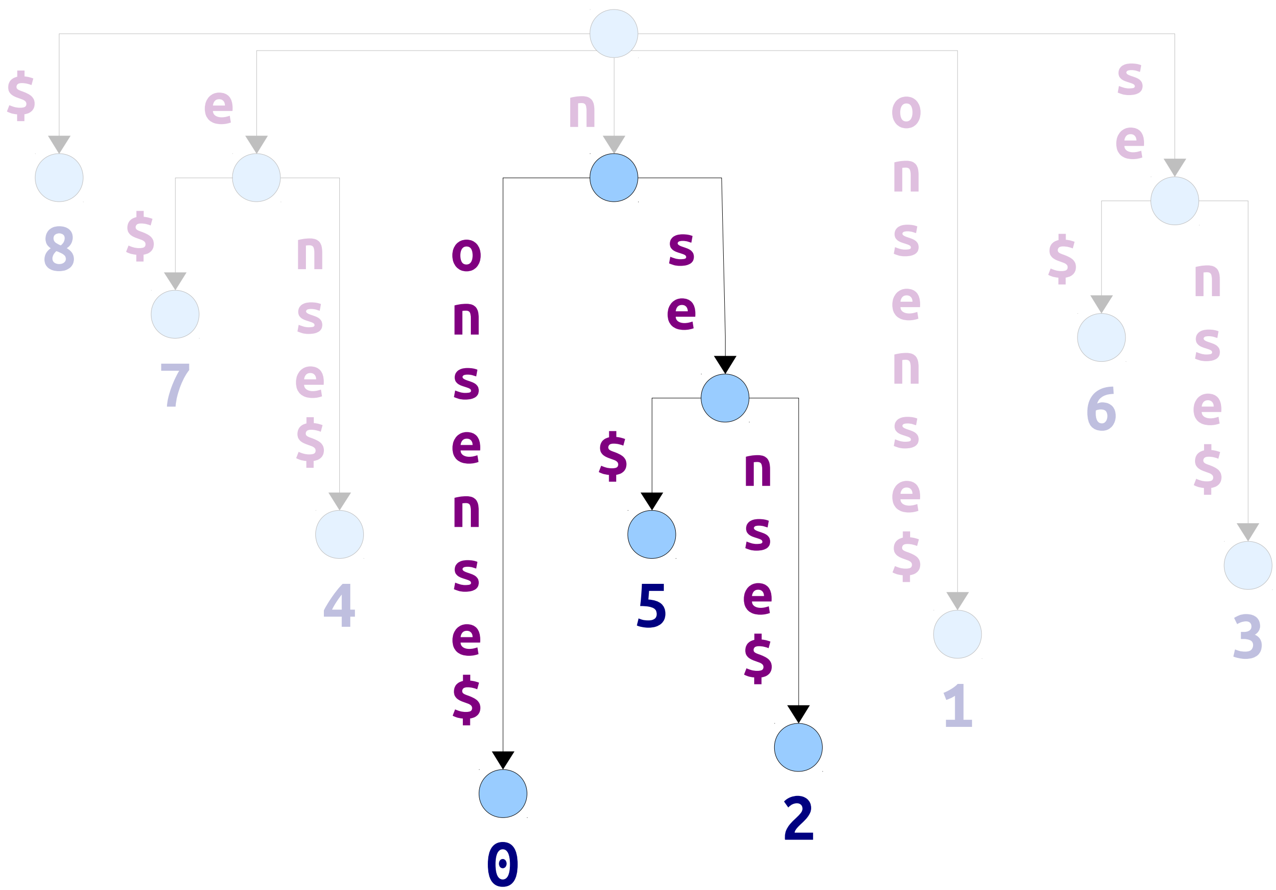
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

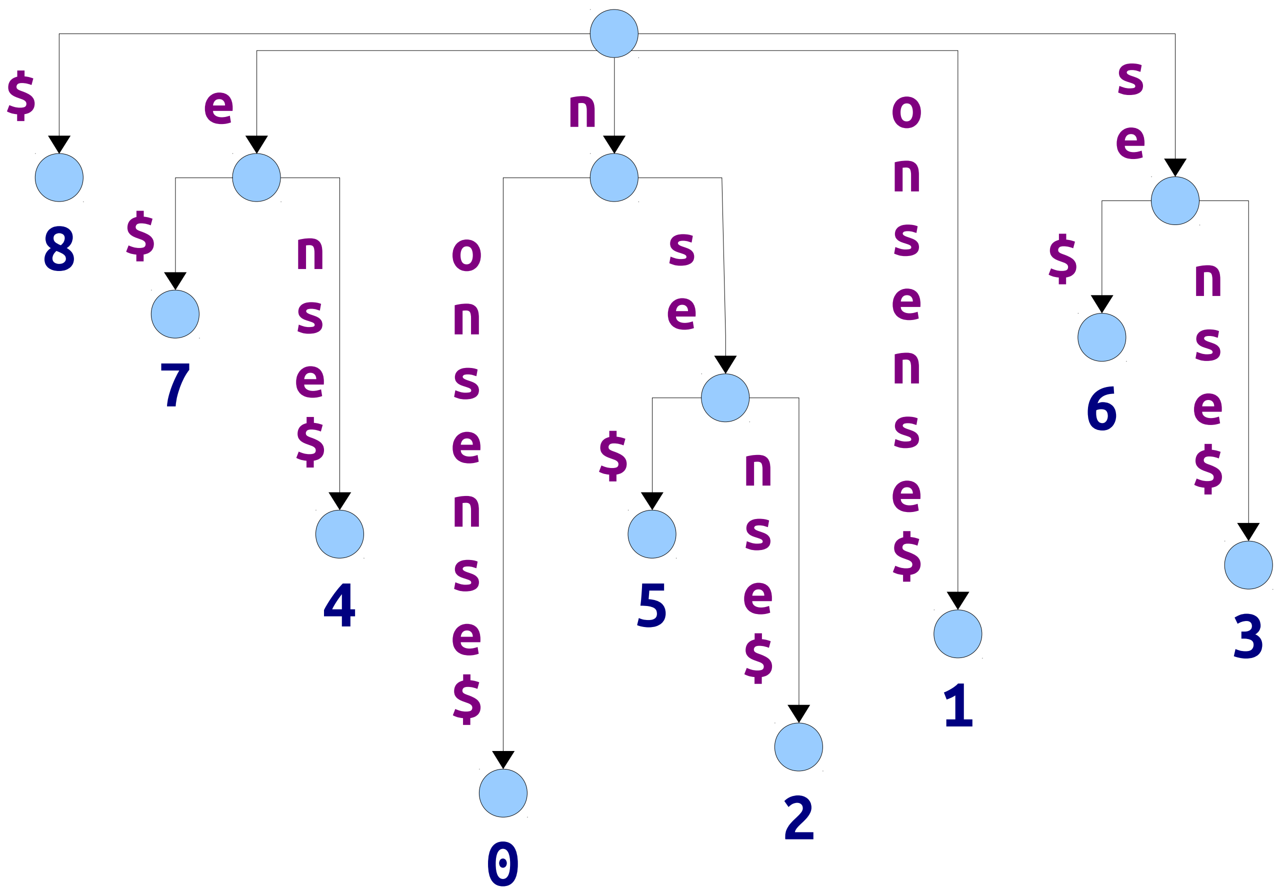
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

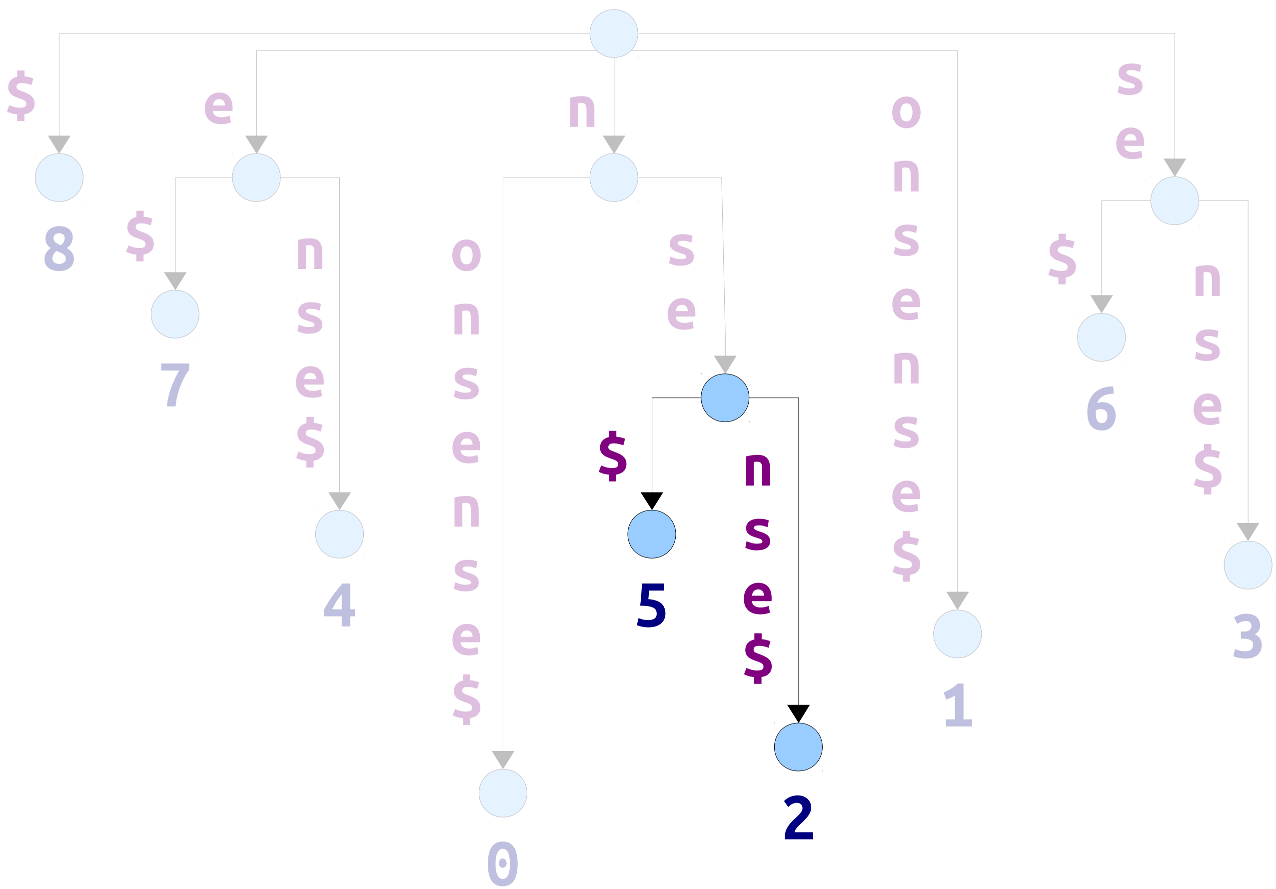
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

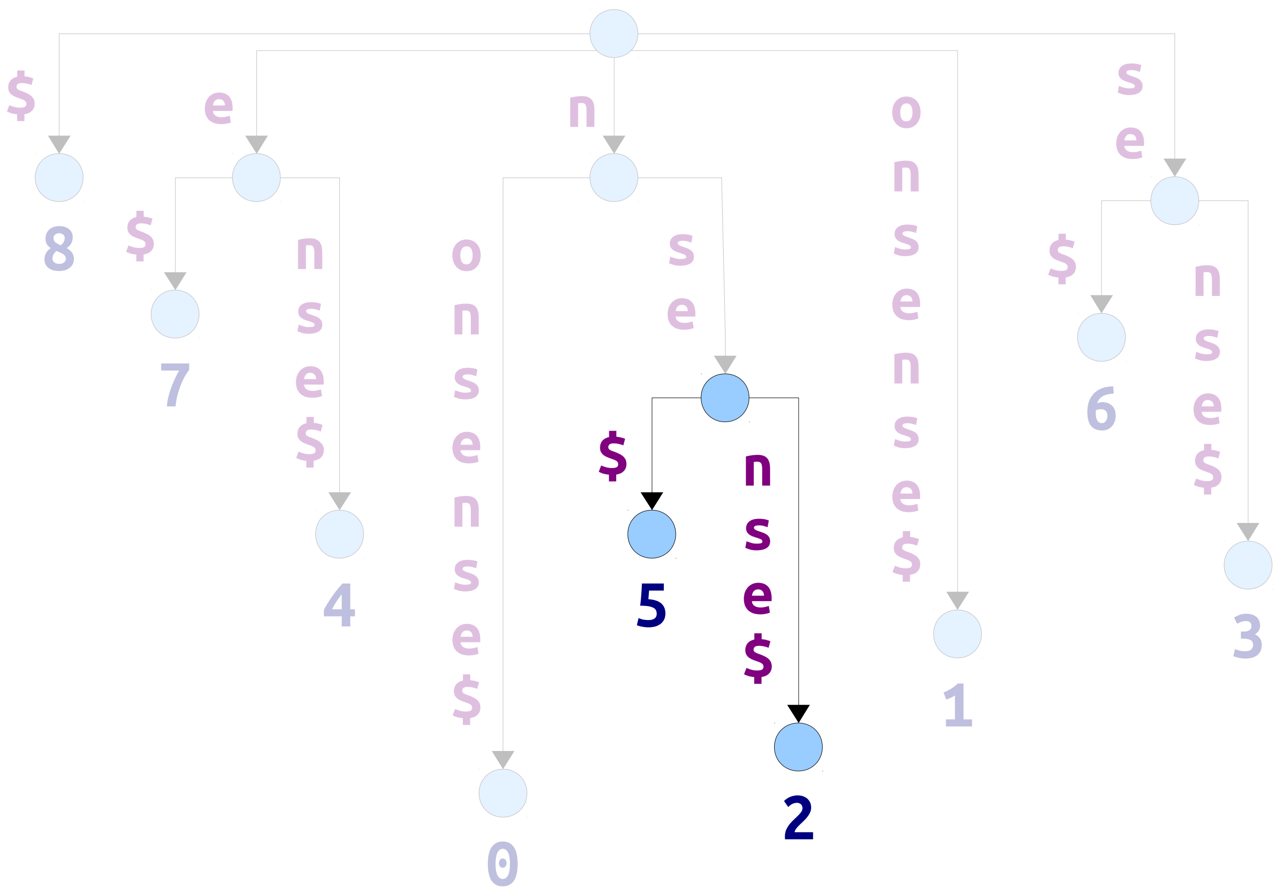
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

Suffix Trees and Suffix Arrays



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

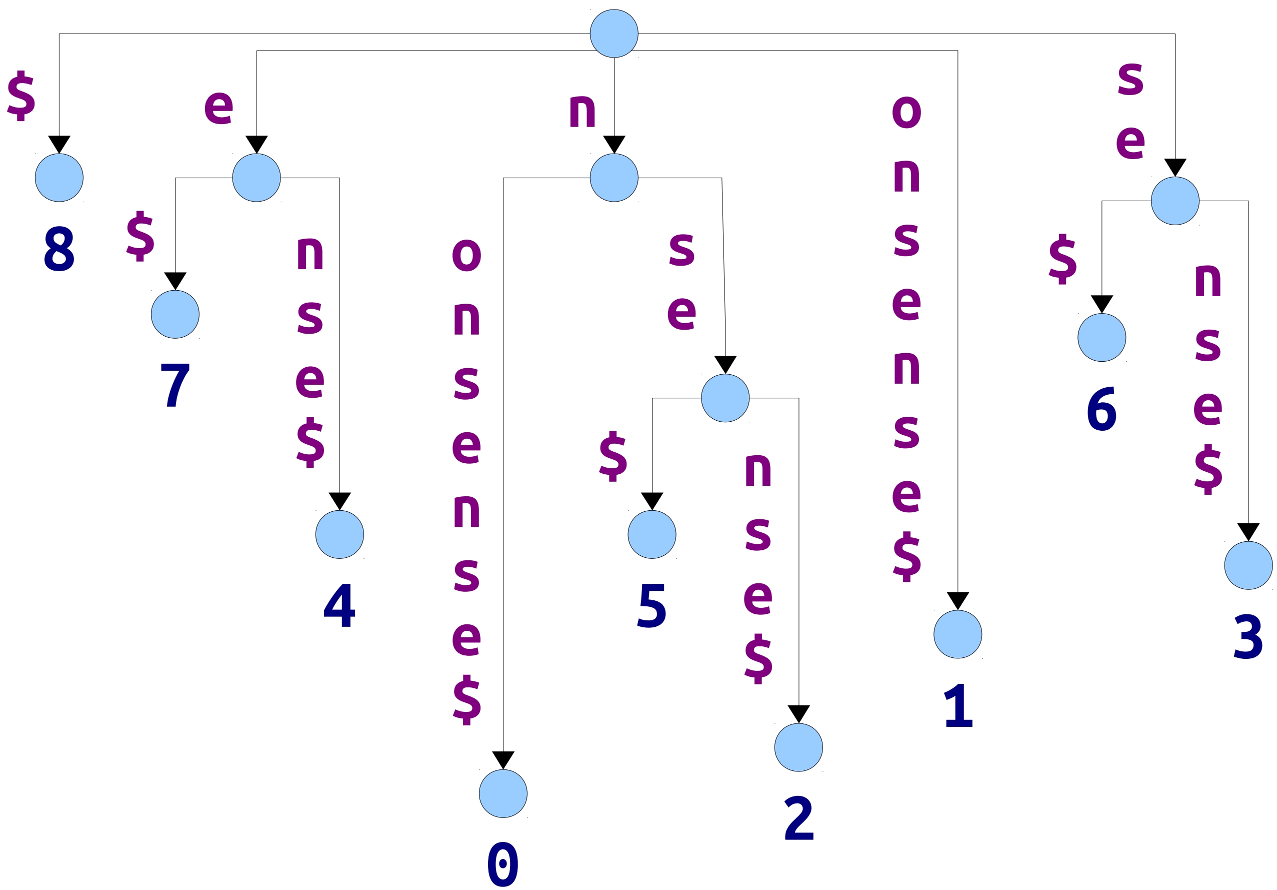
nonsense$
012345678

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

Longest Common Prefixes

- Given two strings x and y, the **longest common prefix** or (**LCP**) of x and y is the longest prefix of x that is also a prefix of y.
- The LCP of x and y is denoted lcp(x, y).
- LCP information is a fundamental link between suffix trees and suffix arrays.

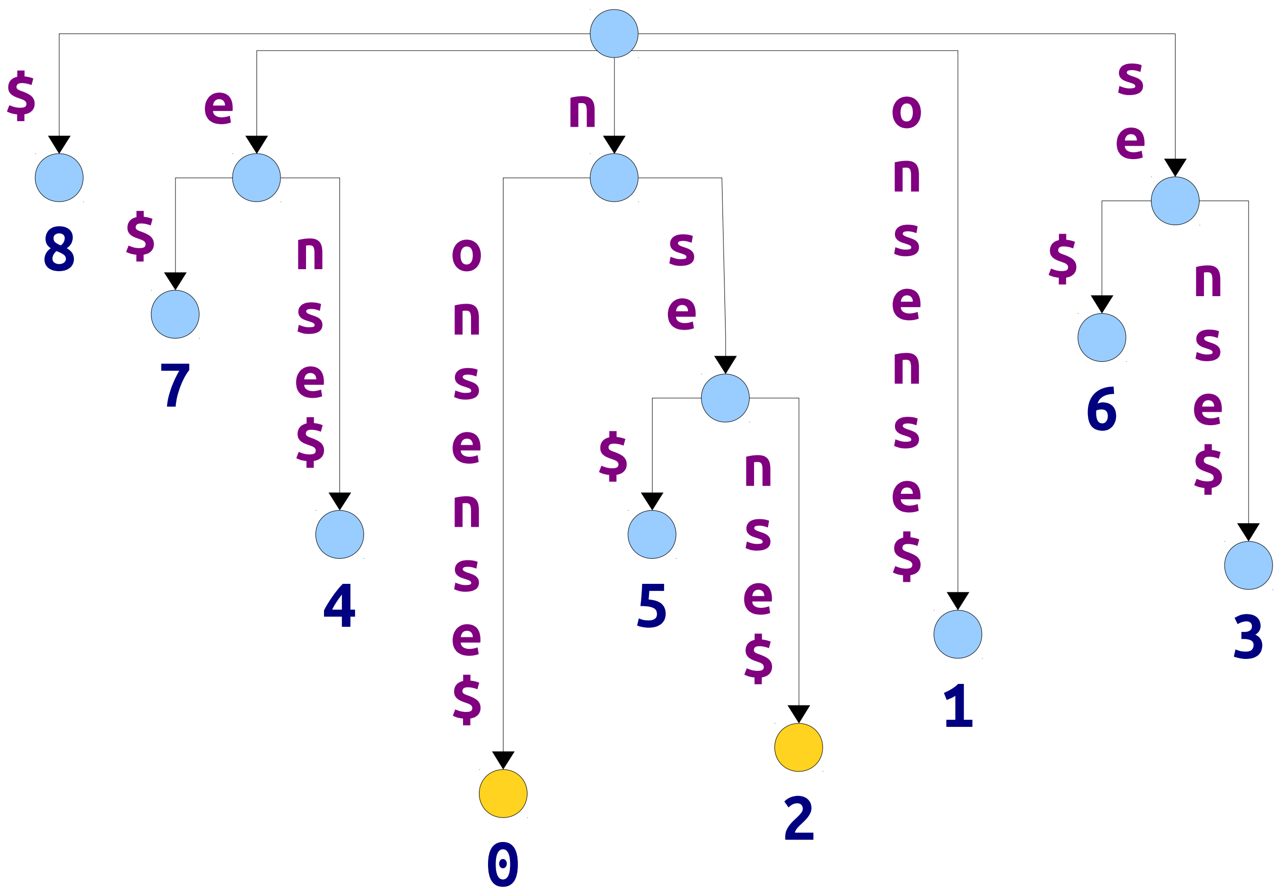
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

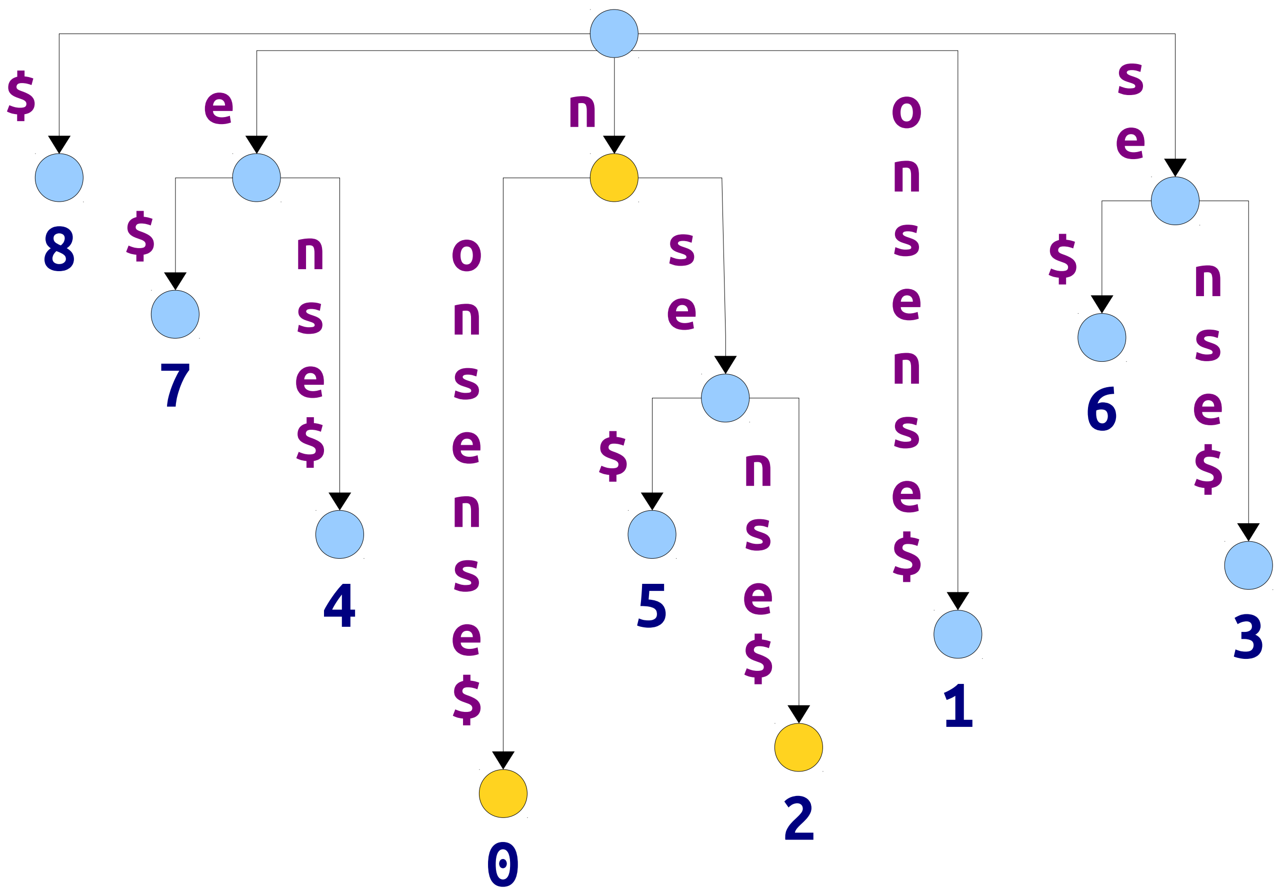
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

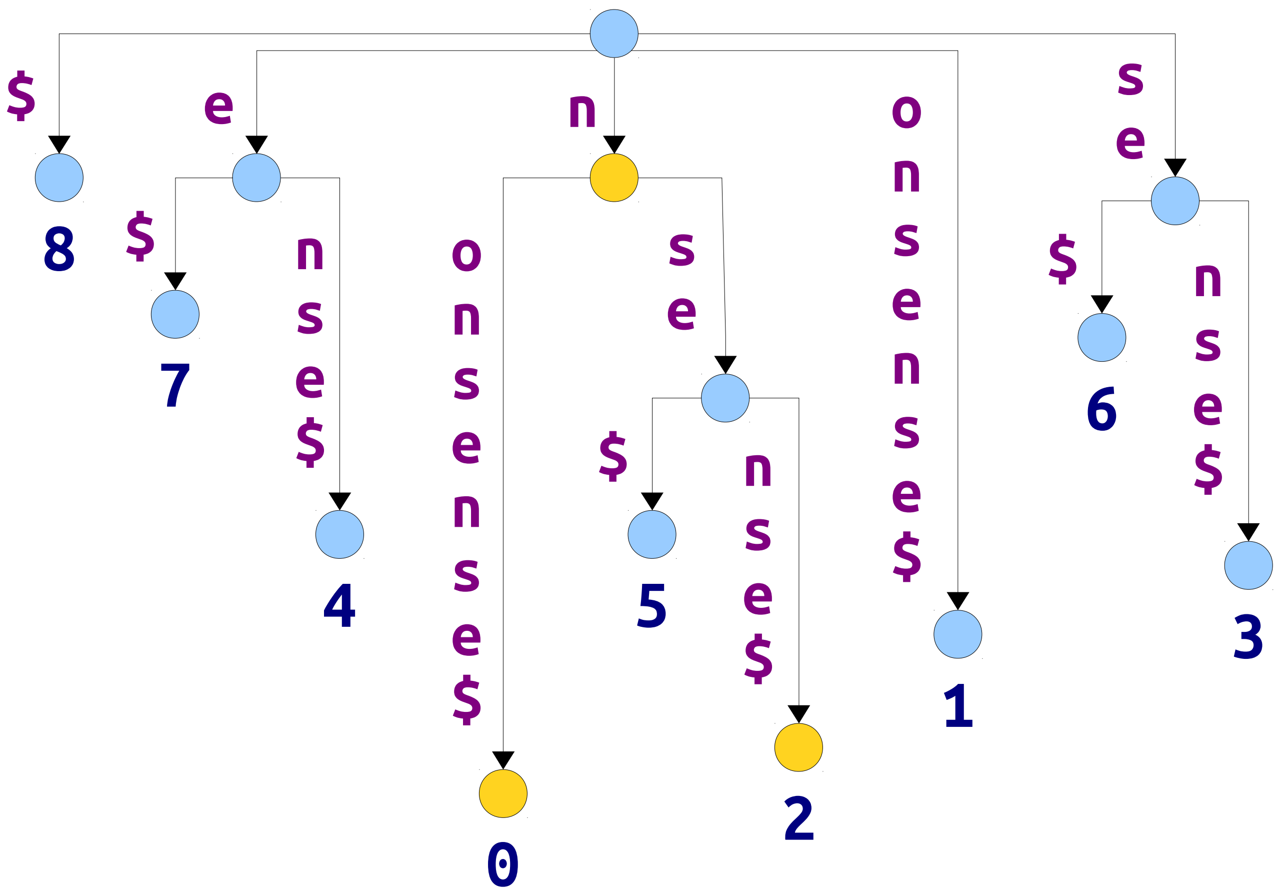
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

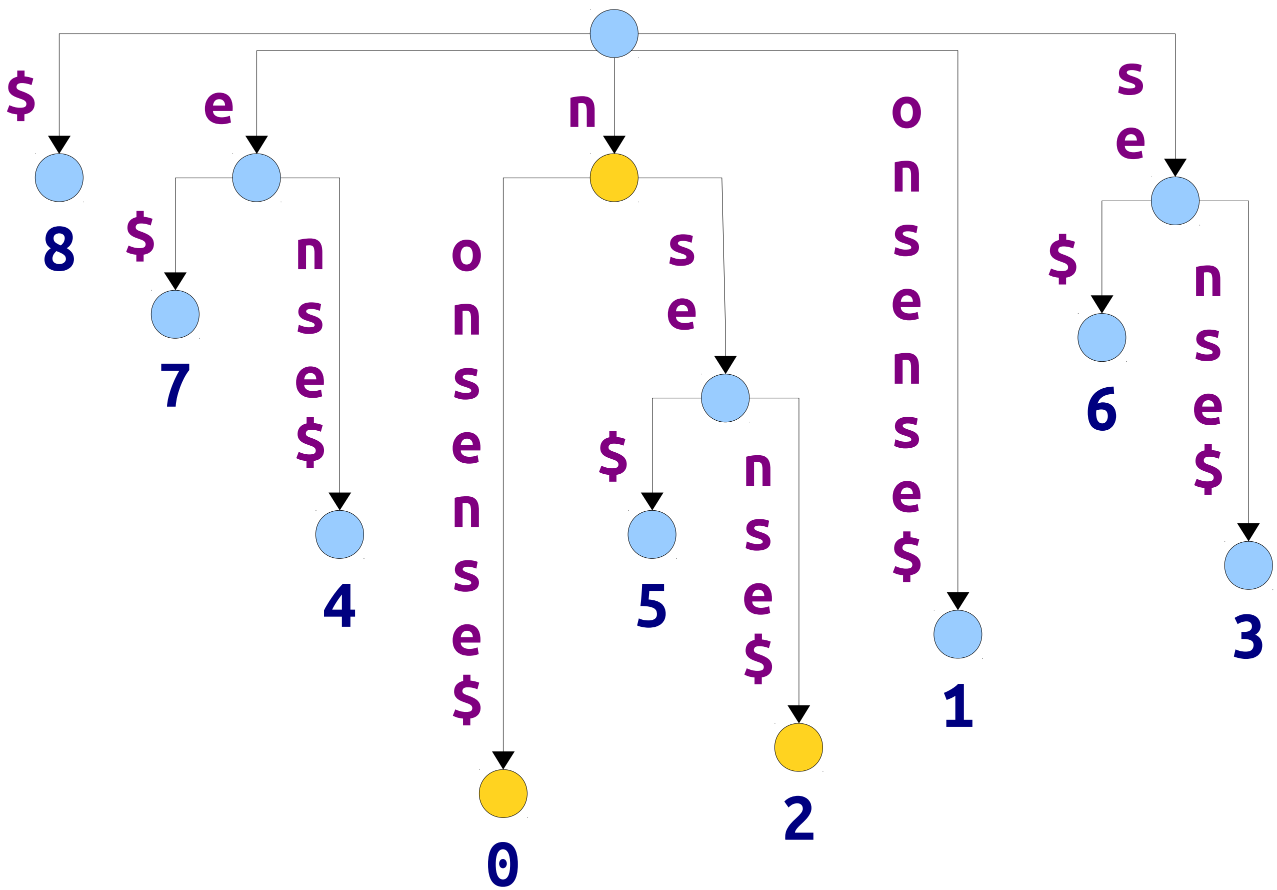
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

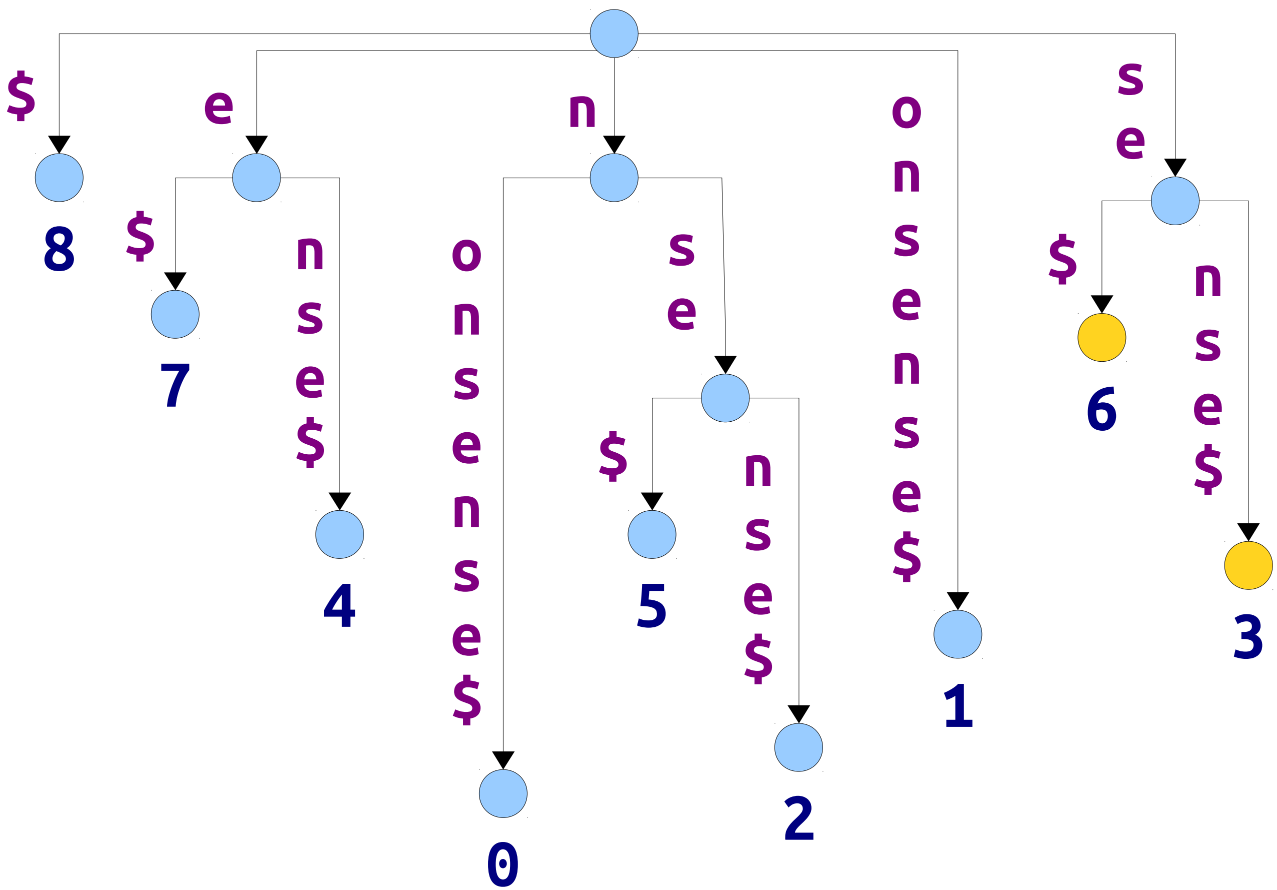
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

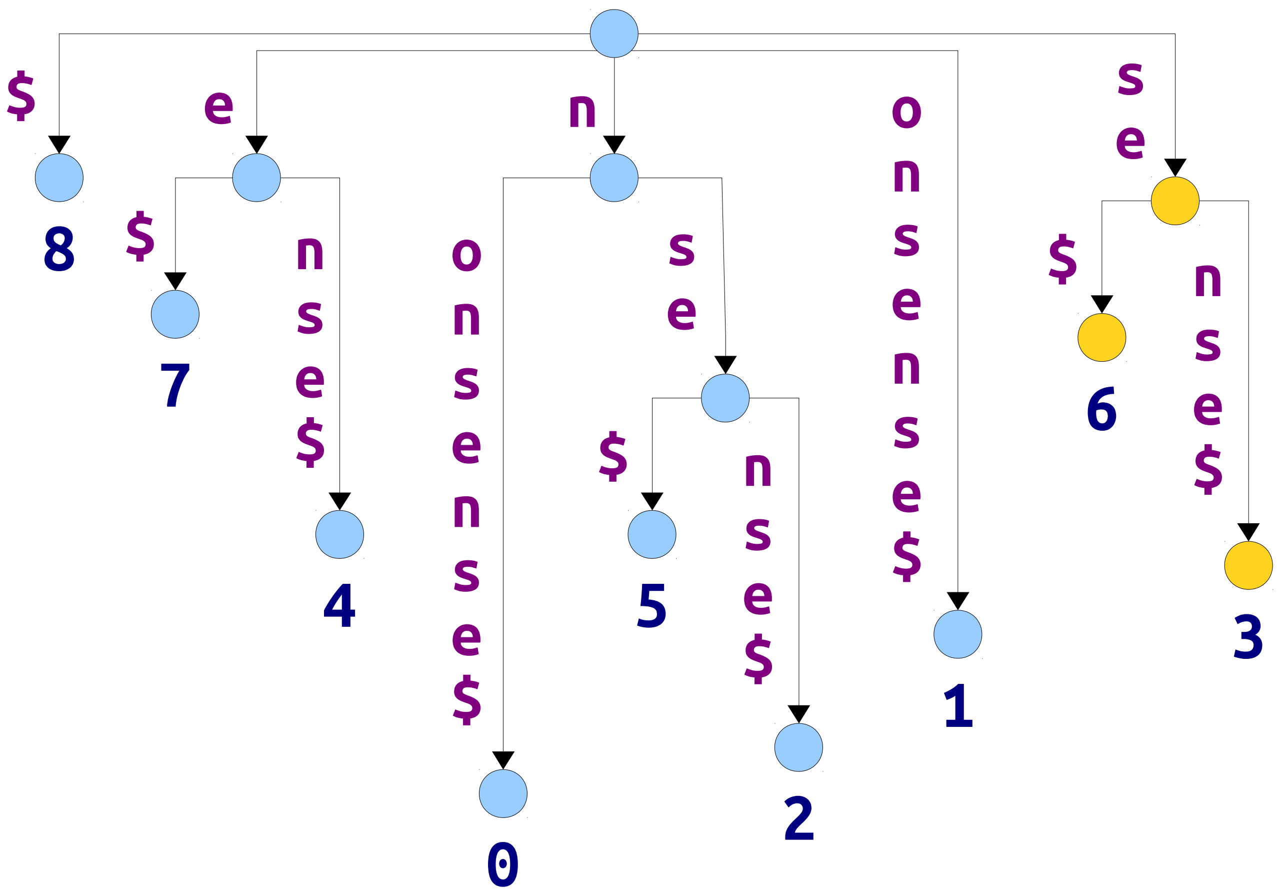
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

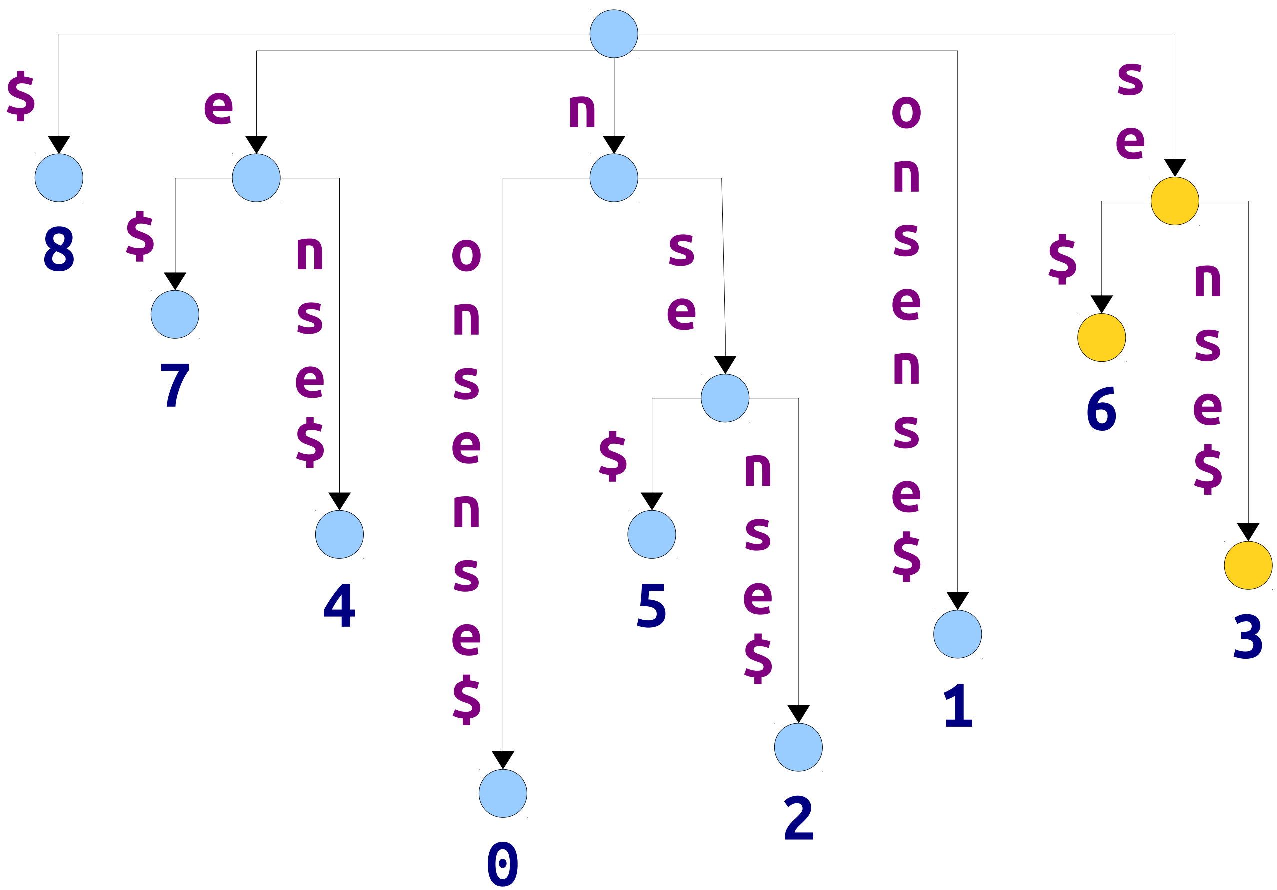
Suffix Trees and Suffix Arrays



nonsense$
012345678

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

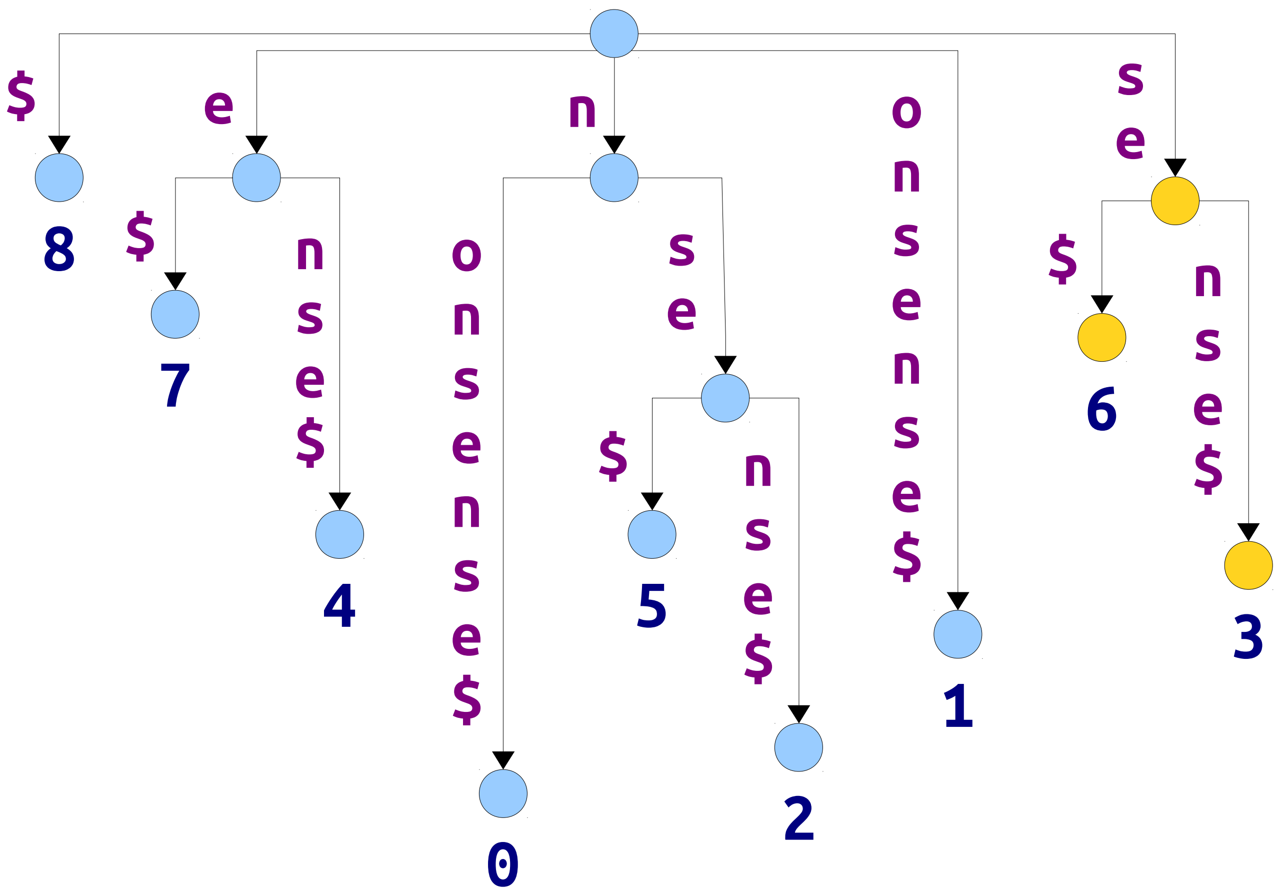
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

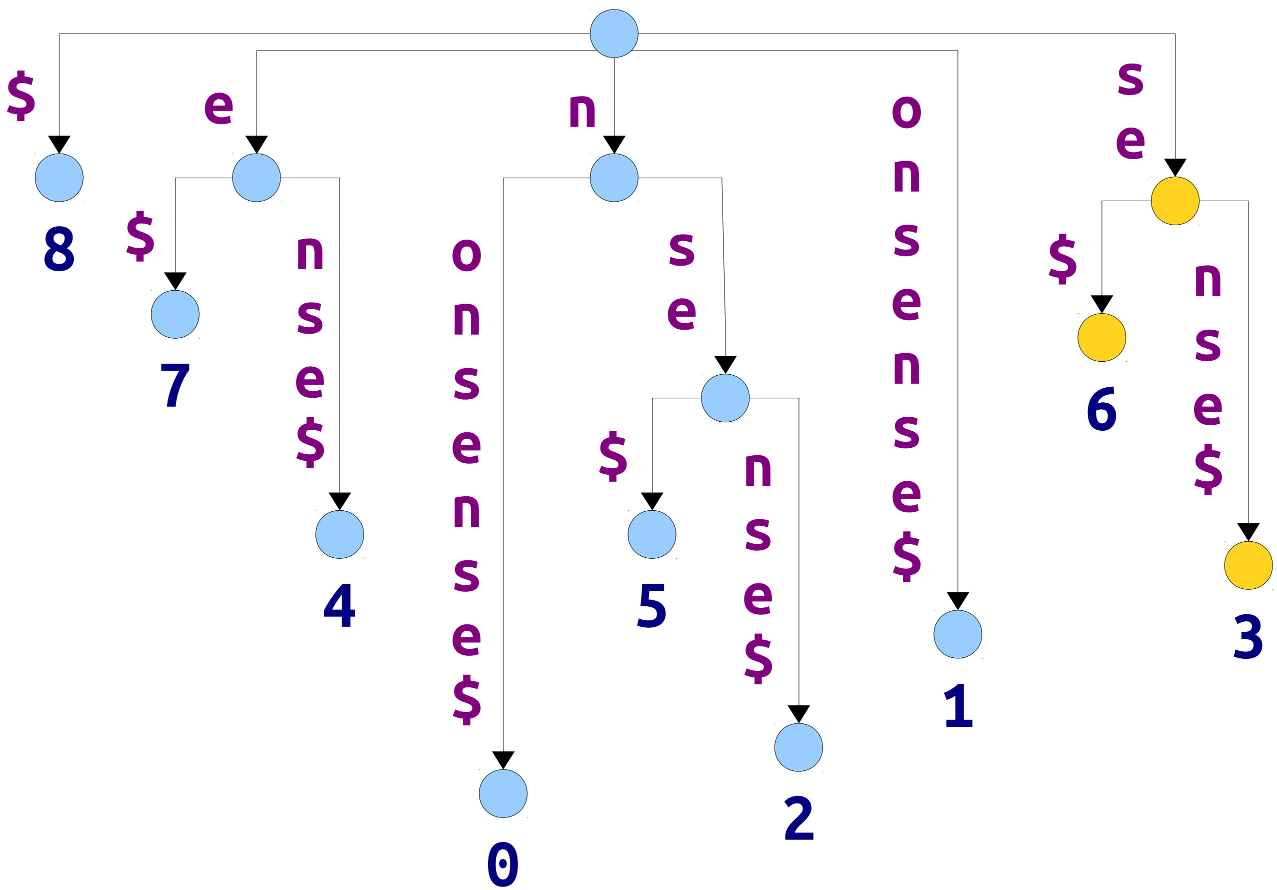
Suffix Trees and Suffix Arrays



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

nonsense$
012345678

Suffix Trees and Suffix Arrays



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

nonsense$
012345678

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

Suffix Arrays with LCP

- Fast computation of LCP information is critical in speeding up algorithms on suffix arrays.
- **Claim:** With O(m) preprocessing, we can answer LCP queries on a suffix array in time O(1).
  - Details later.
- Assuming we can do this, we can give faster algorithms for many suffix array operations.

Speeding Up String Searching

- **Recall:** Can search a suffix array of *T* for a pattern *P* in time O(*n* log *m*).
- **Claim:** Assuming we can quickly compute LCPs, can speed this up to O(*n* + log *m*).
- **Intuition:** Do a normal binary search over the array, but avoid revisiting characters of *n* during the search.
- Use LCP information to keep track of how many characters have been matched.

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.
- Using generalized suffix trees and LCA, can preprocess in time $O(m)$ to answer queries in time $O(1)$.
- **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.
- This problem can be solved trivially if we construct a **generalized suffix array** for T_1 and T_2 augmented with LCP information.
- Additionally, store an inverse table mapping each original suffix to its position in the array.

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

Computing LCP Information

- We've been assuming LCP information can be preprocessed in time O(m) with queries in time O(1).
- Achieving this requires some creativity.

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.
- Check the paper for details; note that there's a typo in their pseudocode; "`j + h`" should be "`k + h`."

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

Pairwise LCP

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

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

Pairwise LCP

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

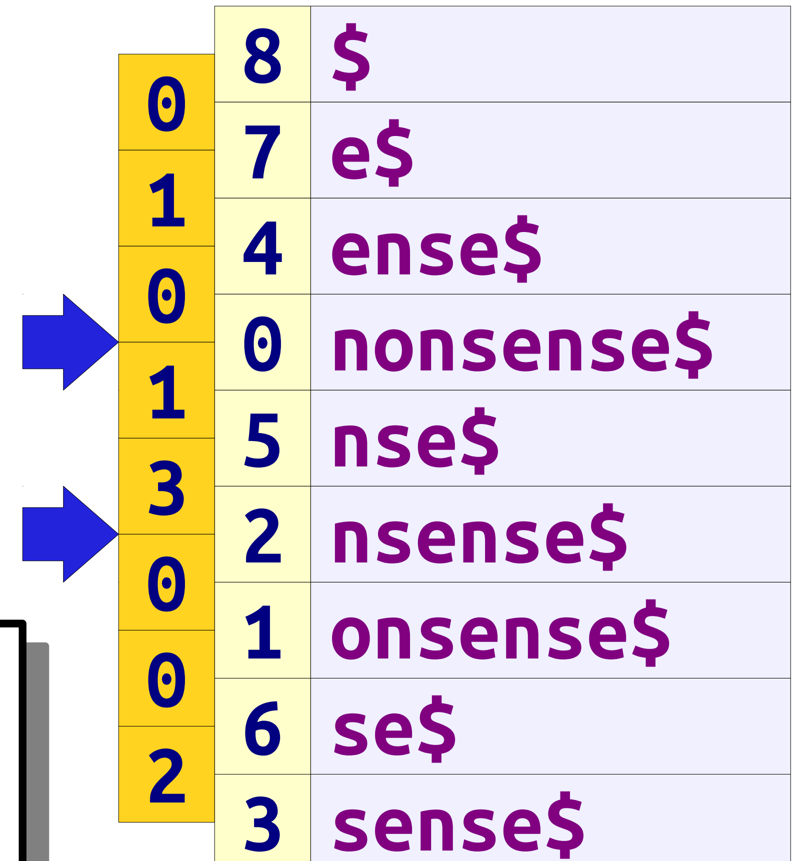
Claim: For any $0 < i < j < m$:

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

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

Pairwise LCP

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

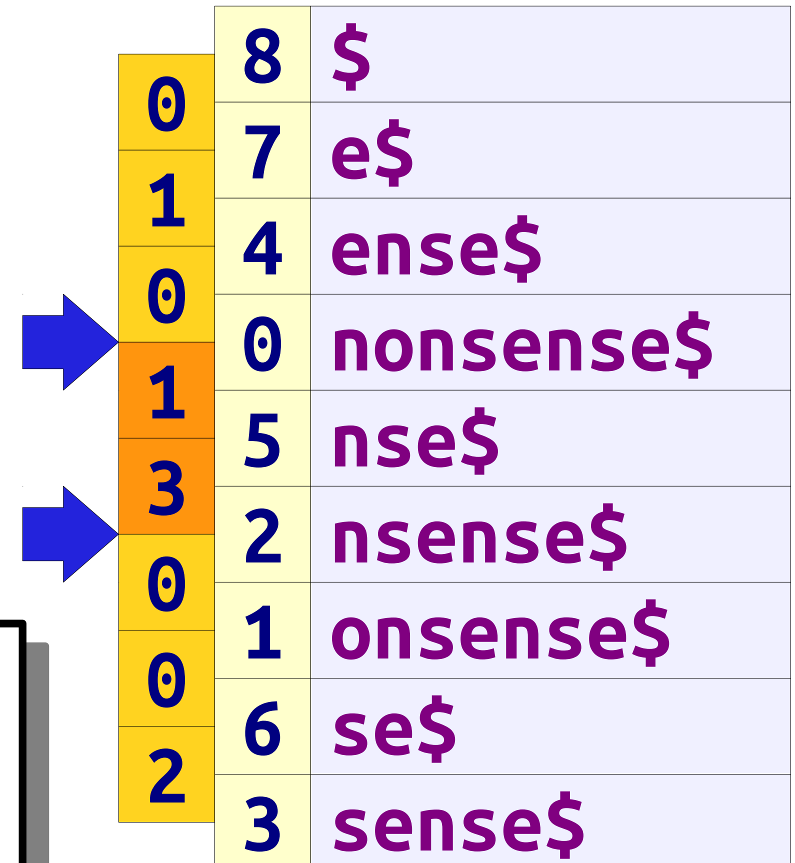


Claim: For any $0 < i < j < m$:

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

Pairwise LCP

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



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

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

Computing LCPs

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

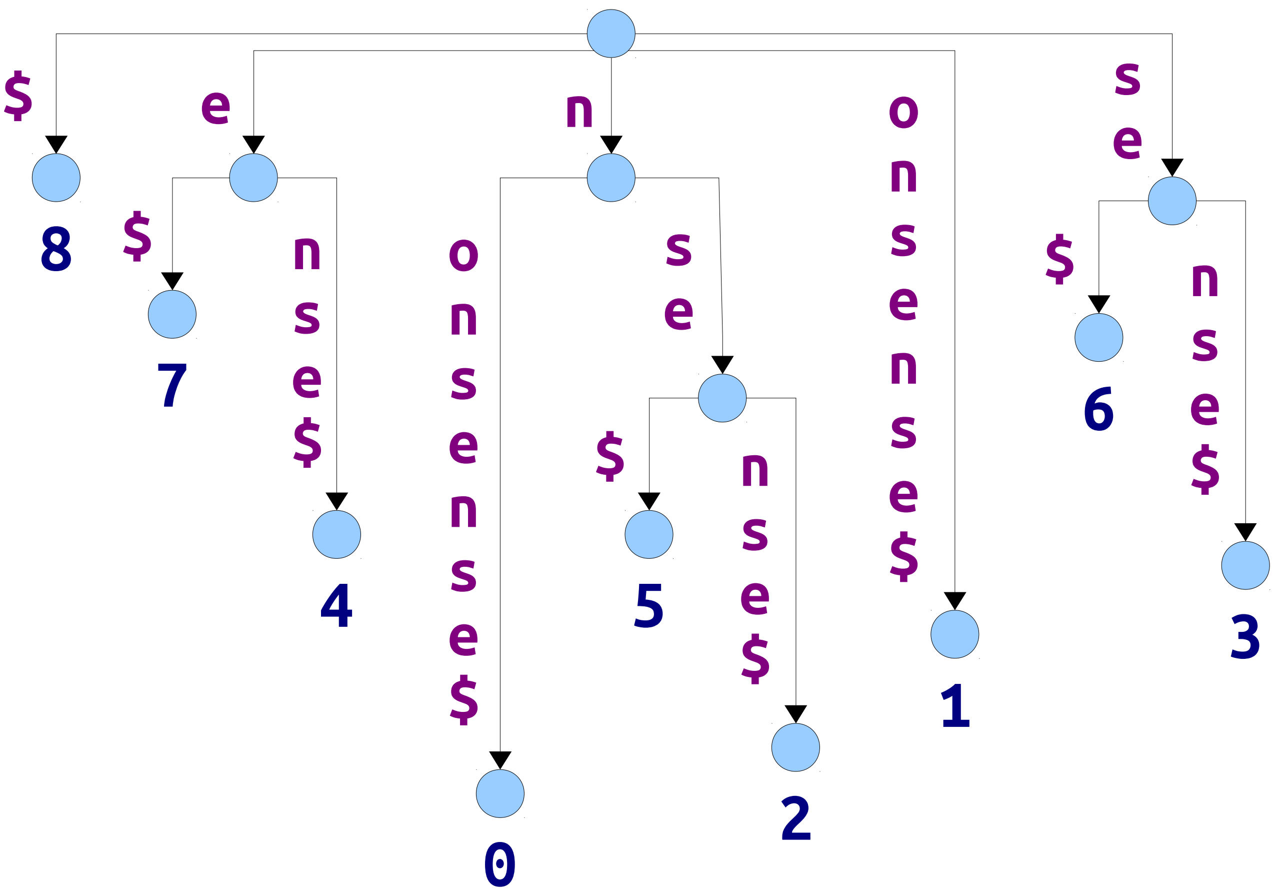
Constructing Suffix Trees

Constructing Suffix Trees

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

From Suffix Arrays to Suffix Trees

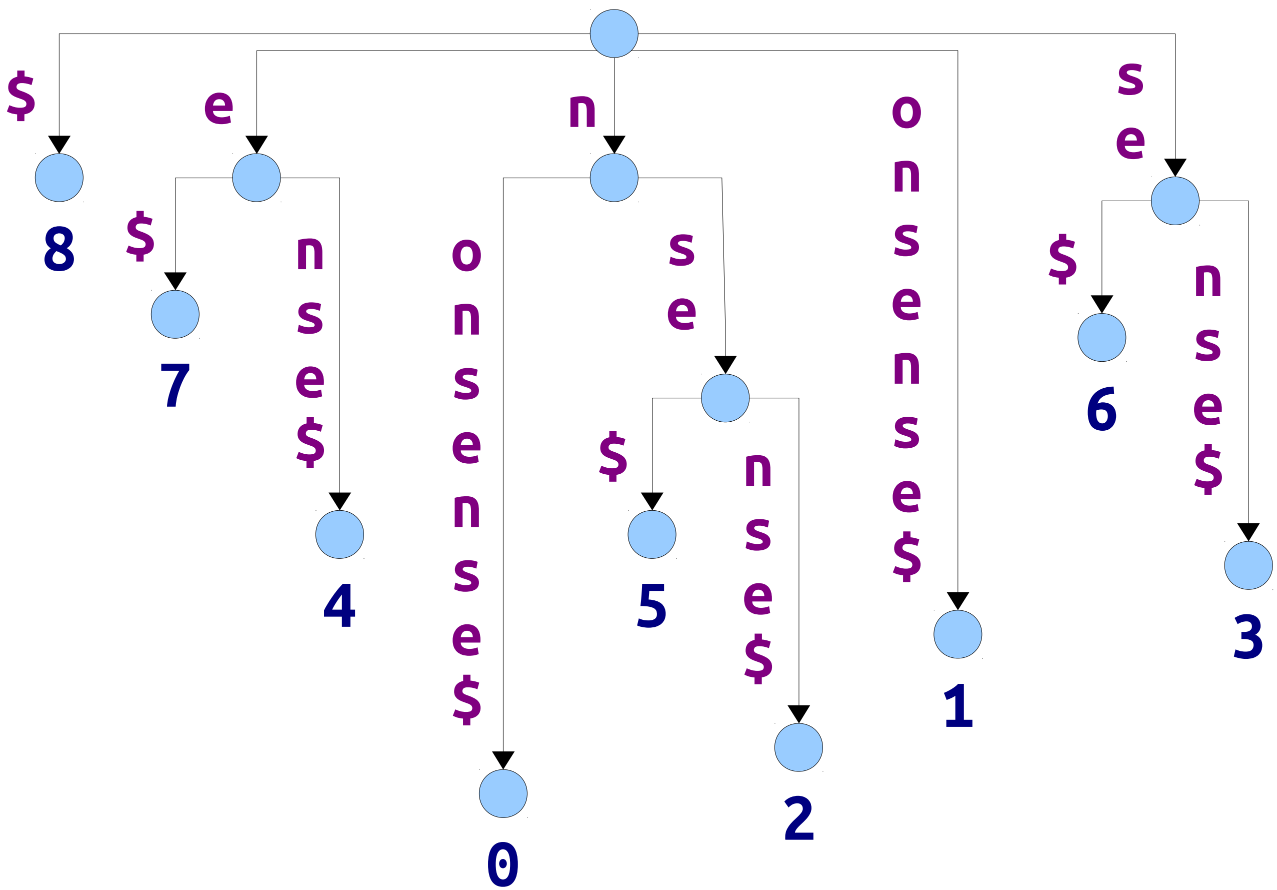
Using LCP



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

nonsense$
012345678

Using LCP

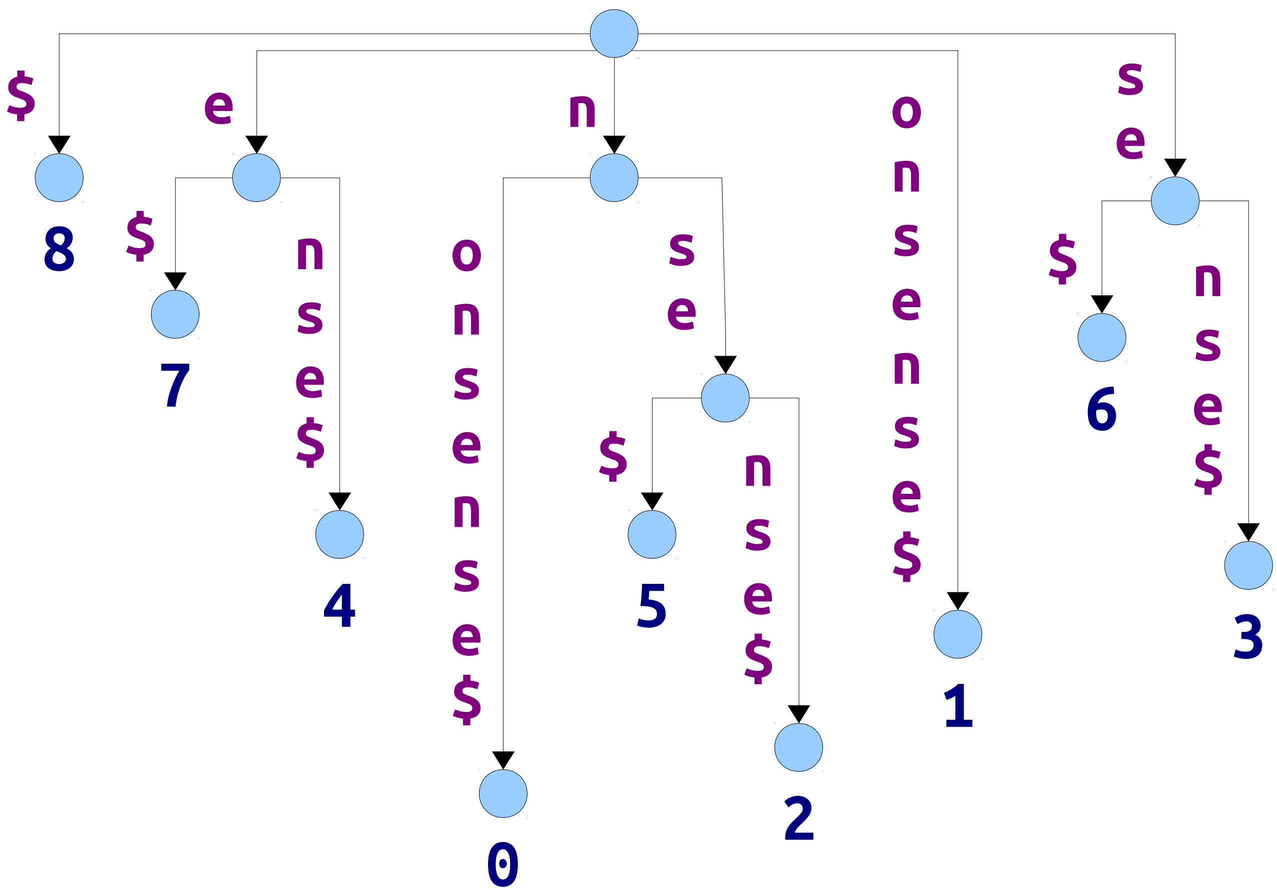


LCP	SA	Suffix
	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

nonsense$
012345678

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

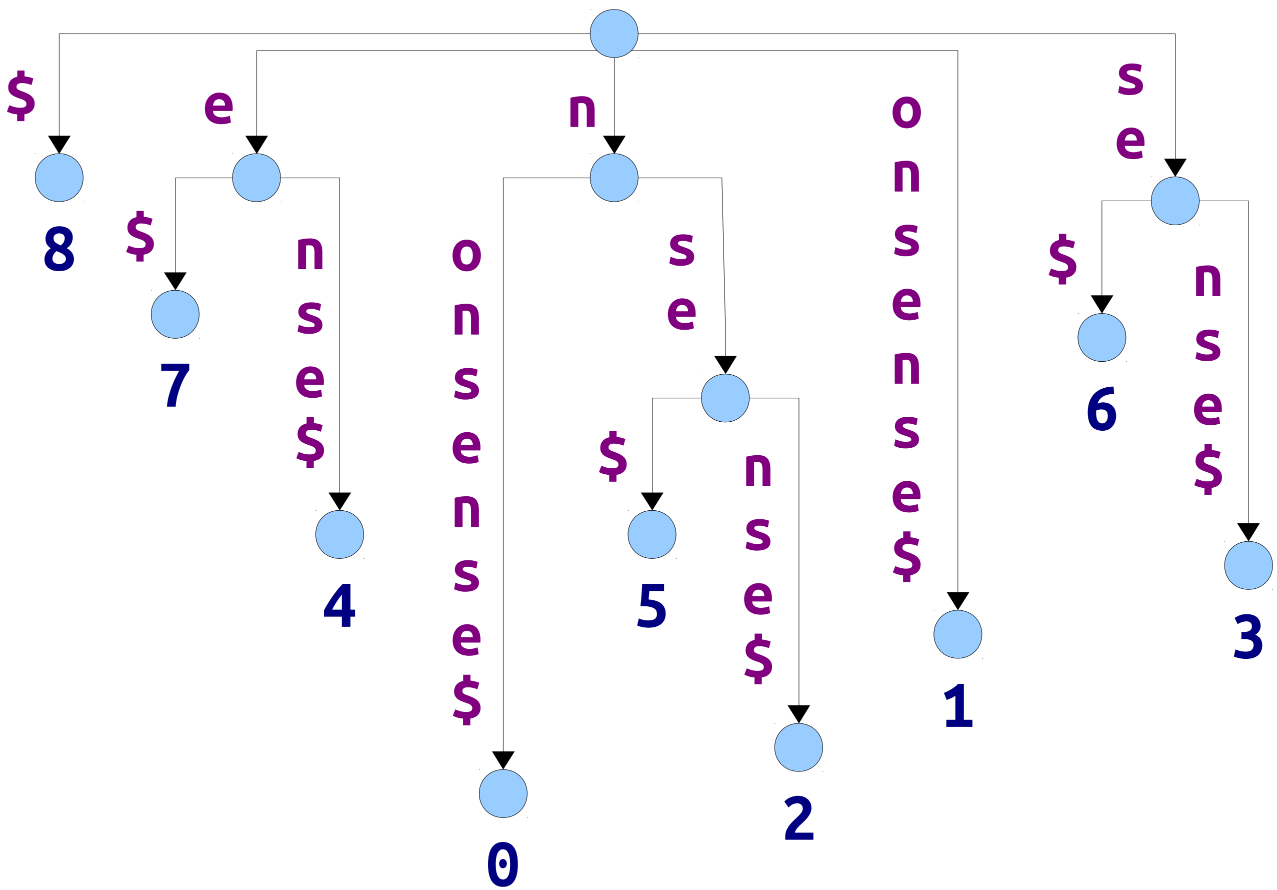
Using LCP



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

nonsense$
012345678

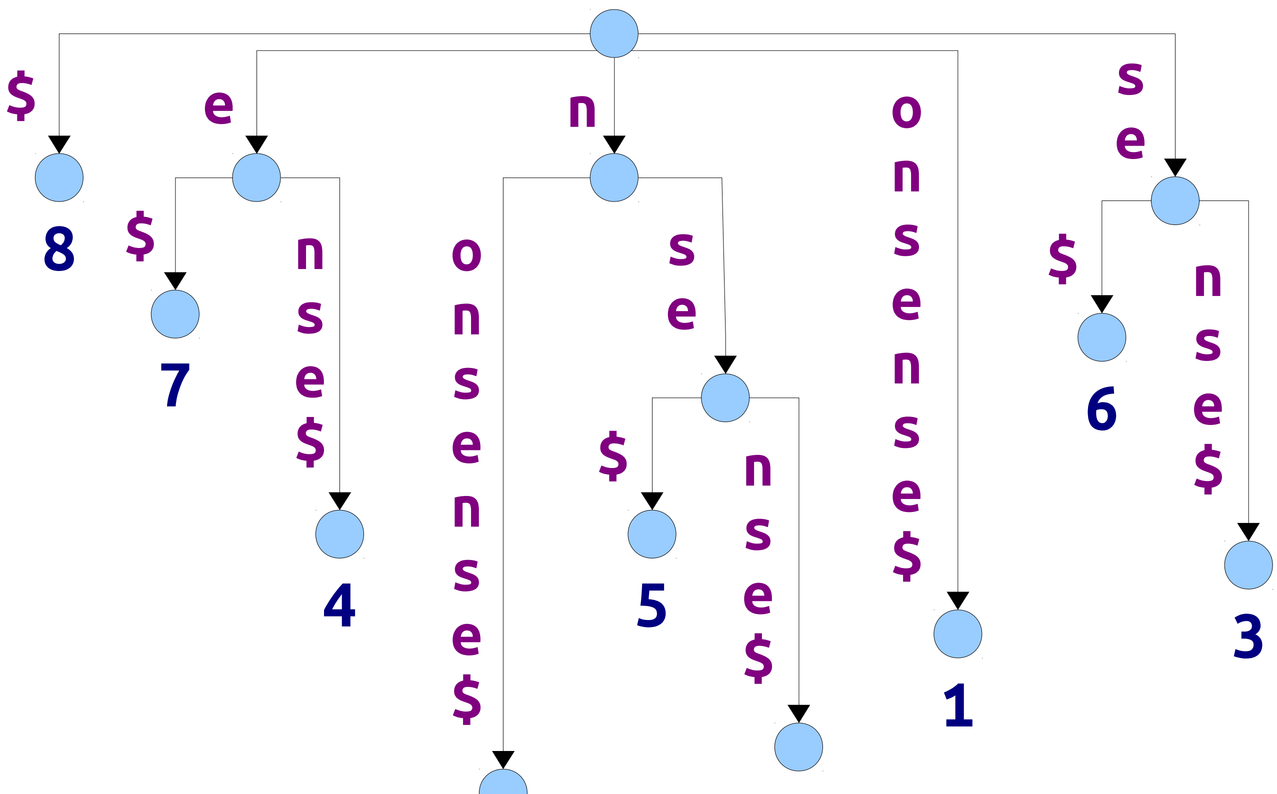
Using LCP



nonsense$
012345678

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

Using LCP



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

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

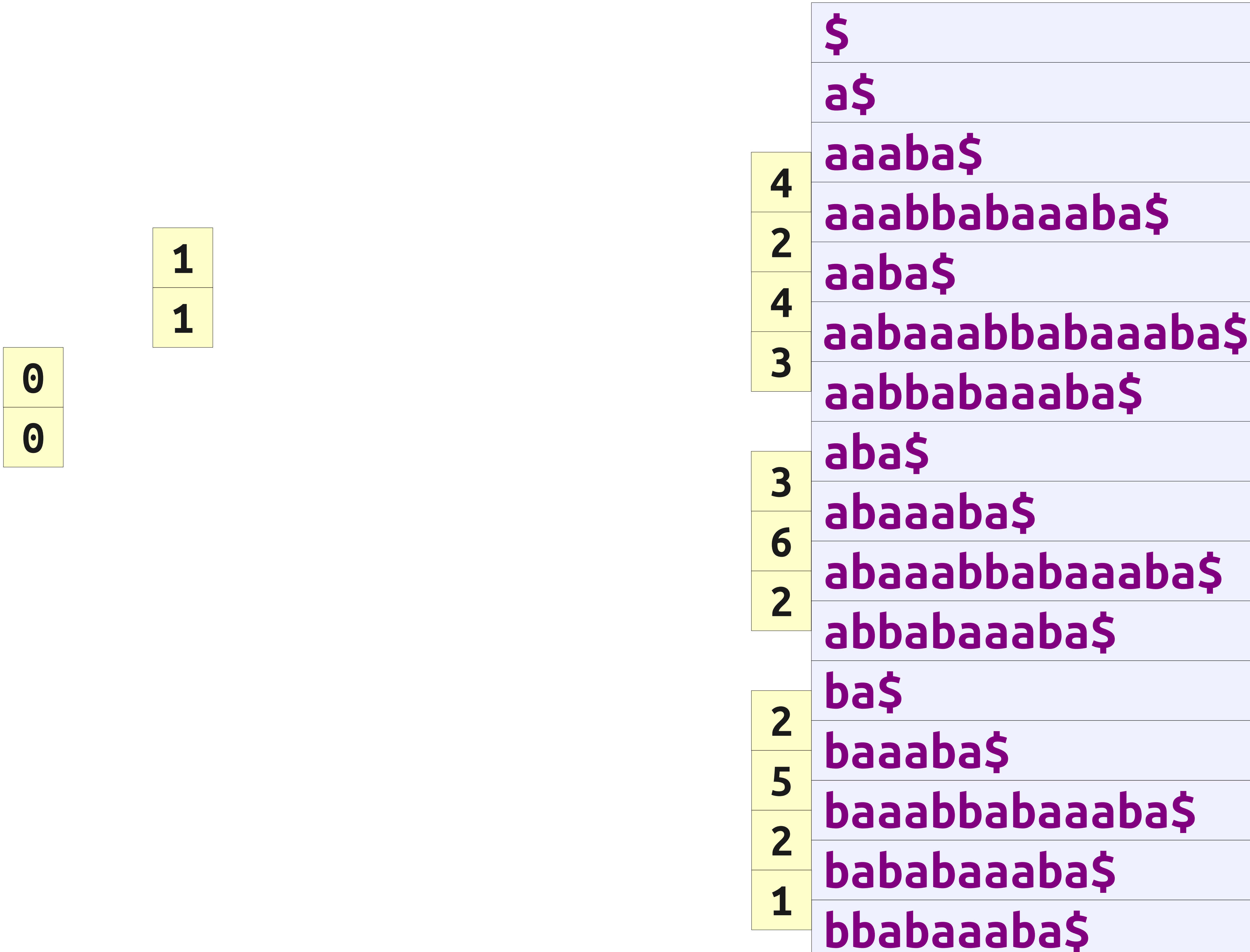
	$
0	a$
1	aaaba$
4	aaabbabaaaba$
2	aaba$
4	aabaaabbabaaaba$
3	aabbabaaaba$
1	aba$
3	abaaaba$
6	abaaabbabaaaba$
2	abbabaaaba$
0	ba$
2	baaaba$
5	baaabbabaaaba$
2	bababaaaba$
1	bbabaaaba$

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

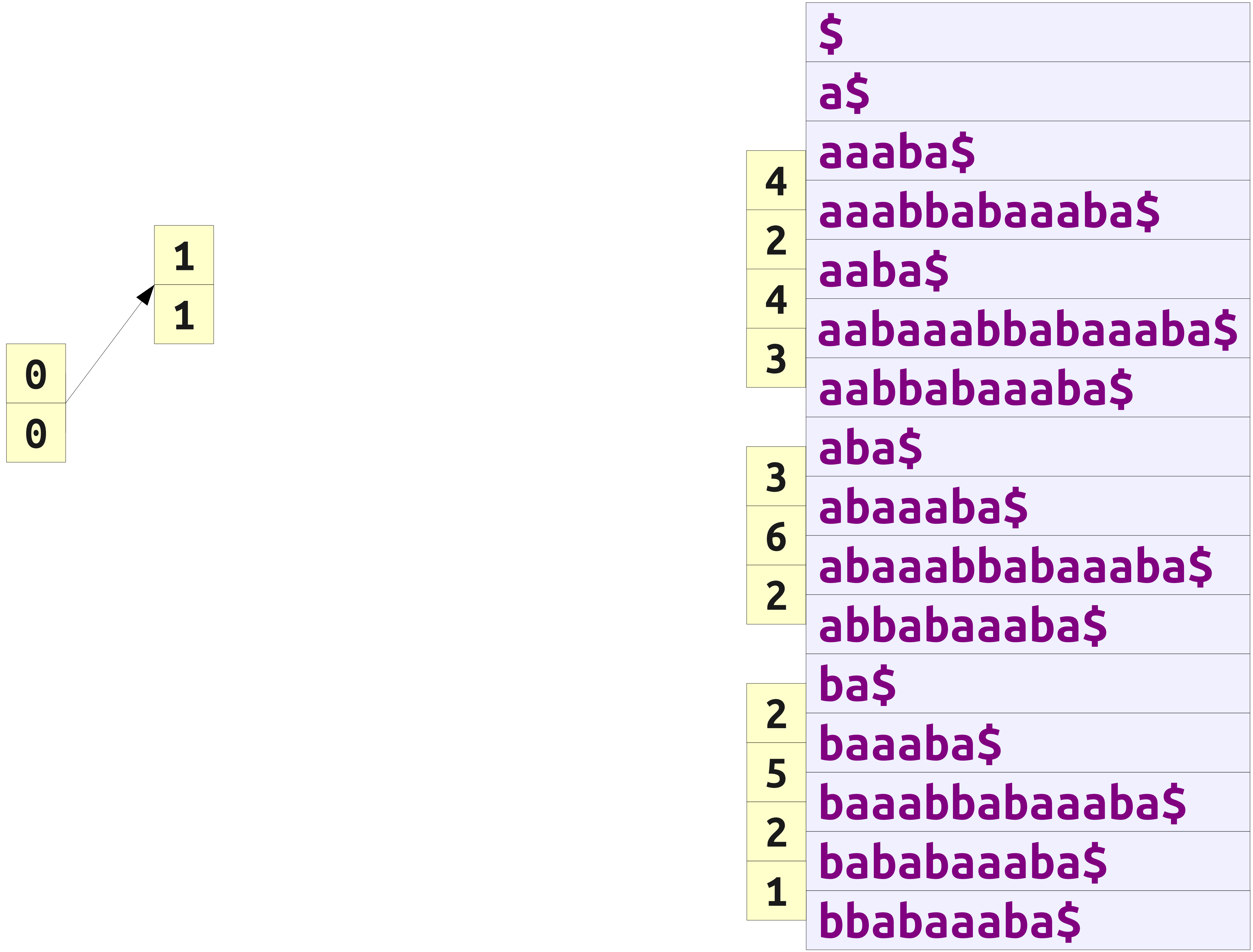
0	
0	

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

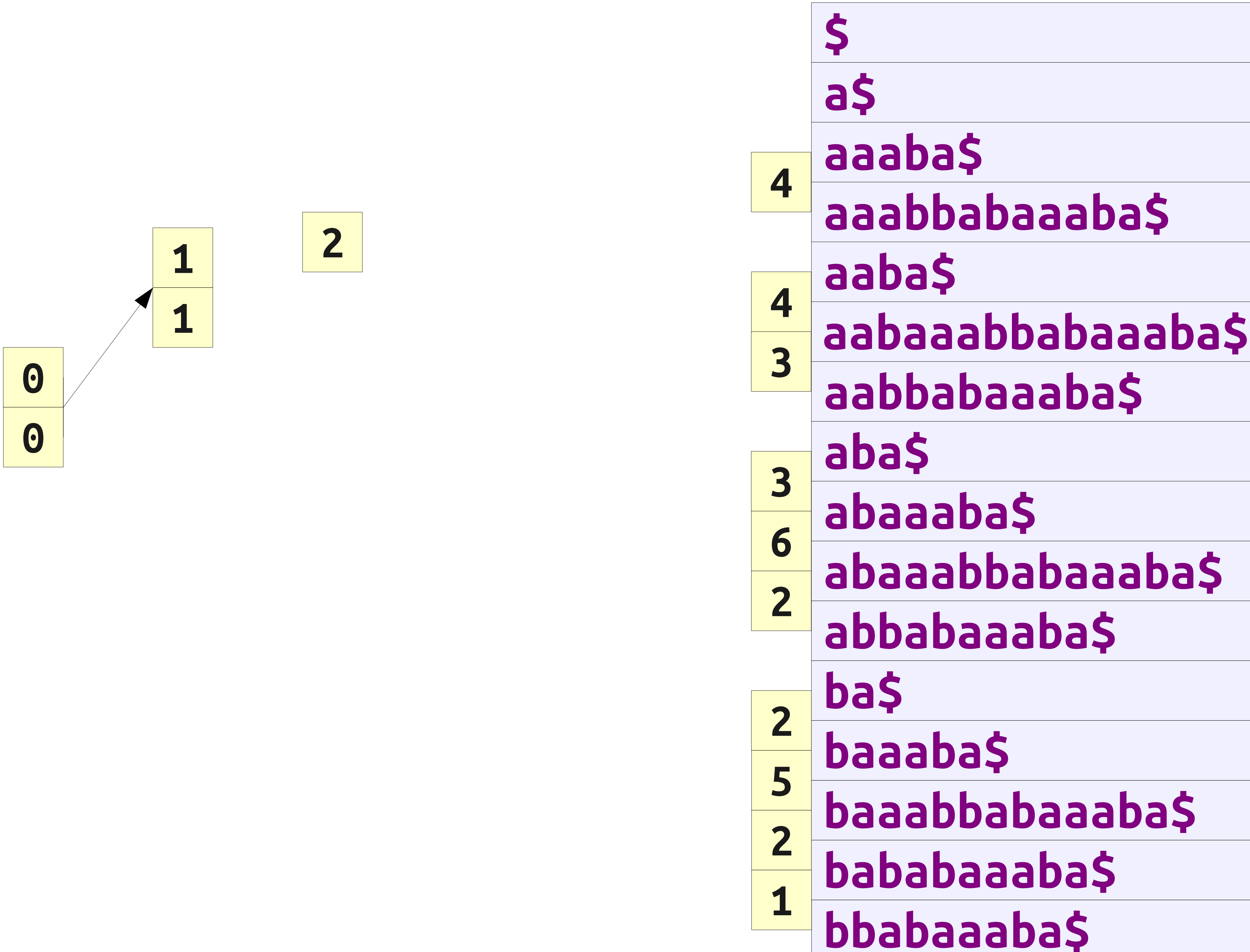
			$
			a$
	1		aaaba$
		4	aaabbabaaaba$
		2	aaba$
		4	aabaaabbabaaaba$
0		3	aabbabaaaba$
0	1		aba$
		3	abaaaba$
		6	abaaabbabaaaba$
		2	abbabaaaba$
			ba$
		2	baaaba$
		5	baaabbabaaaba$
		2	bababaaaba$
		1	bbabaaaba$



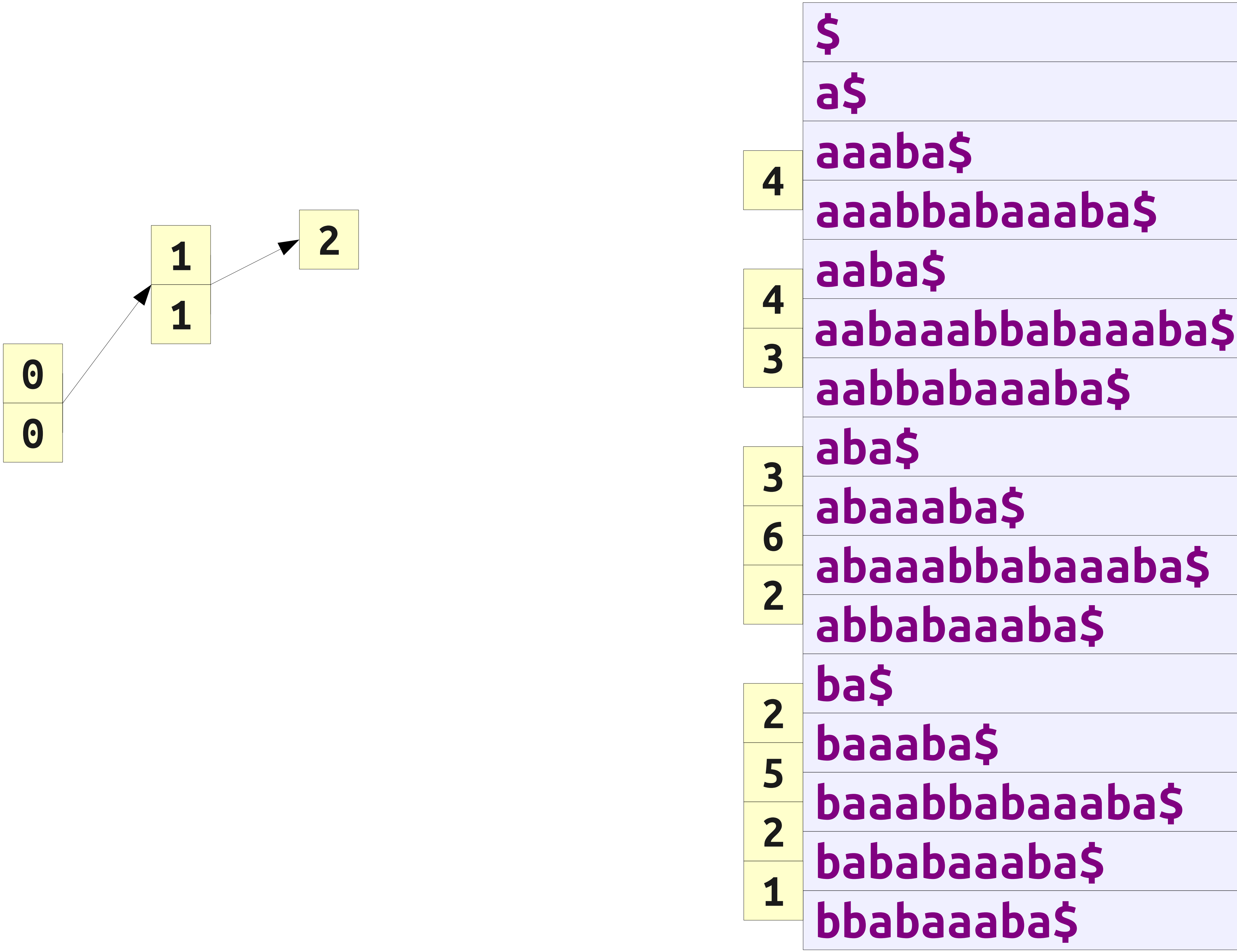
$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$
4
2
4
3
3
6
2
2
5
2
1
1
1
0
0



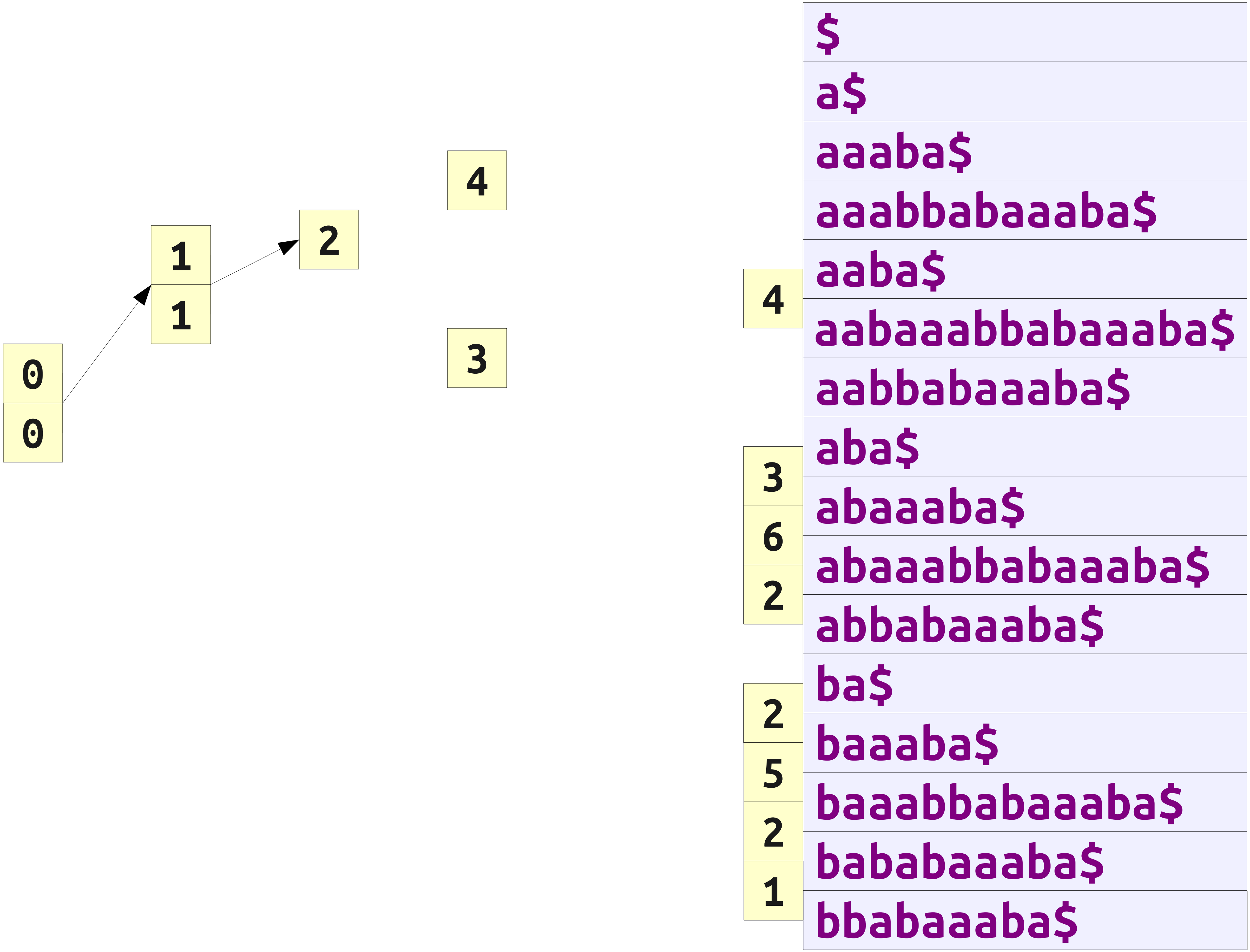
0
0
1
1
$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$
4
2
4
3
3
6
2
2
5
2
1



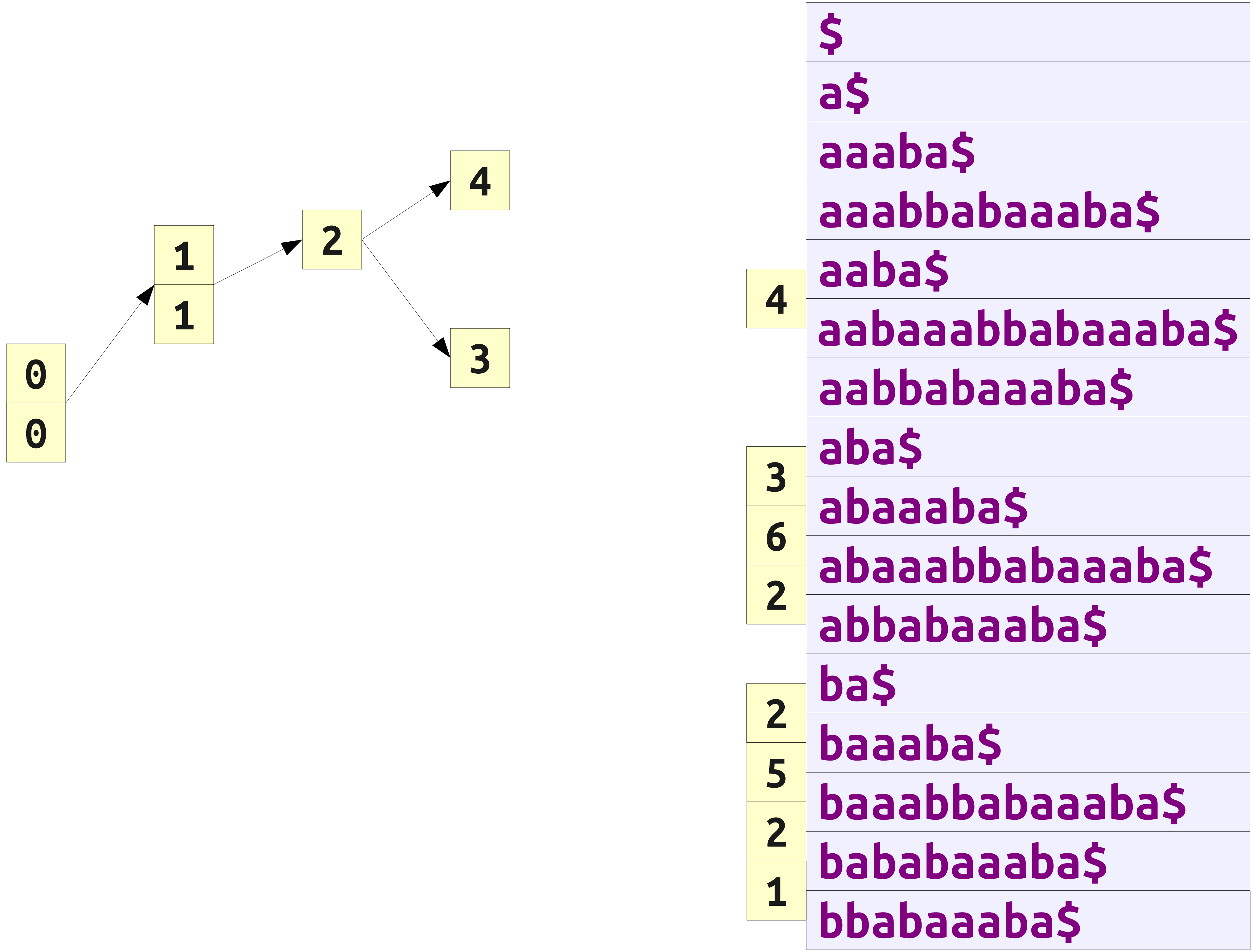
0
0
1
1
2
$
a$
4
aaaba$
aaabbabaaaba$
4
aaba$
3
aabaaabbabaaaba$
aabbabaaaba$
3
aba$
6
abaaaba$
2
abaaabbabaaaba$
abbabaaaba$
2
ba$
5
baaaba$
2
baaabbabaaaba$
1
bababaaaba$
bbabaaaba$

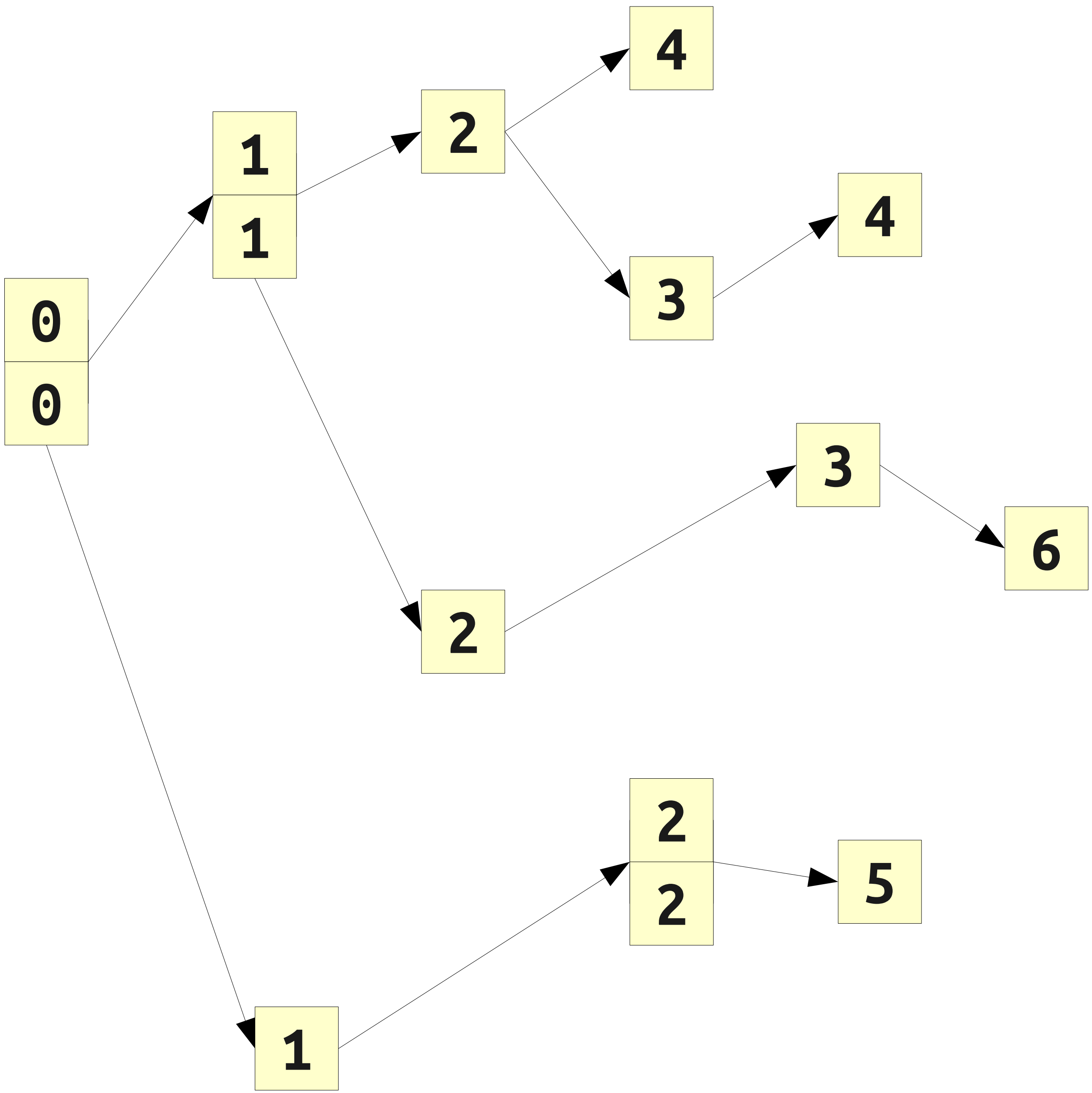


0
0
1
1
2
$
a$
aaaba$
4
aaabbabaaaba$
aaba$
4
aabaaabbabaaaba$
3
aabbabaaaba$
aba$
3
abaaaba$
6
abaaabbabaaaba$
2
abbabaaaba$
ba$
2
baaaba$
5
baaabbabaaaba$
2
bababaaaba$
1
bbabaaaba$



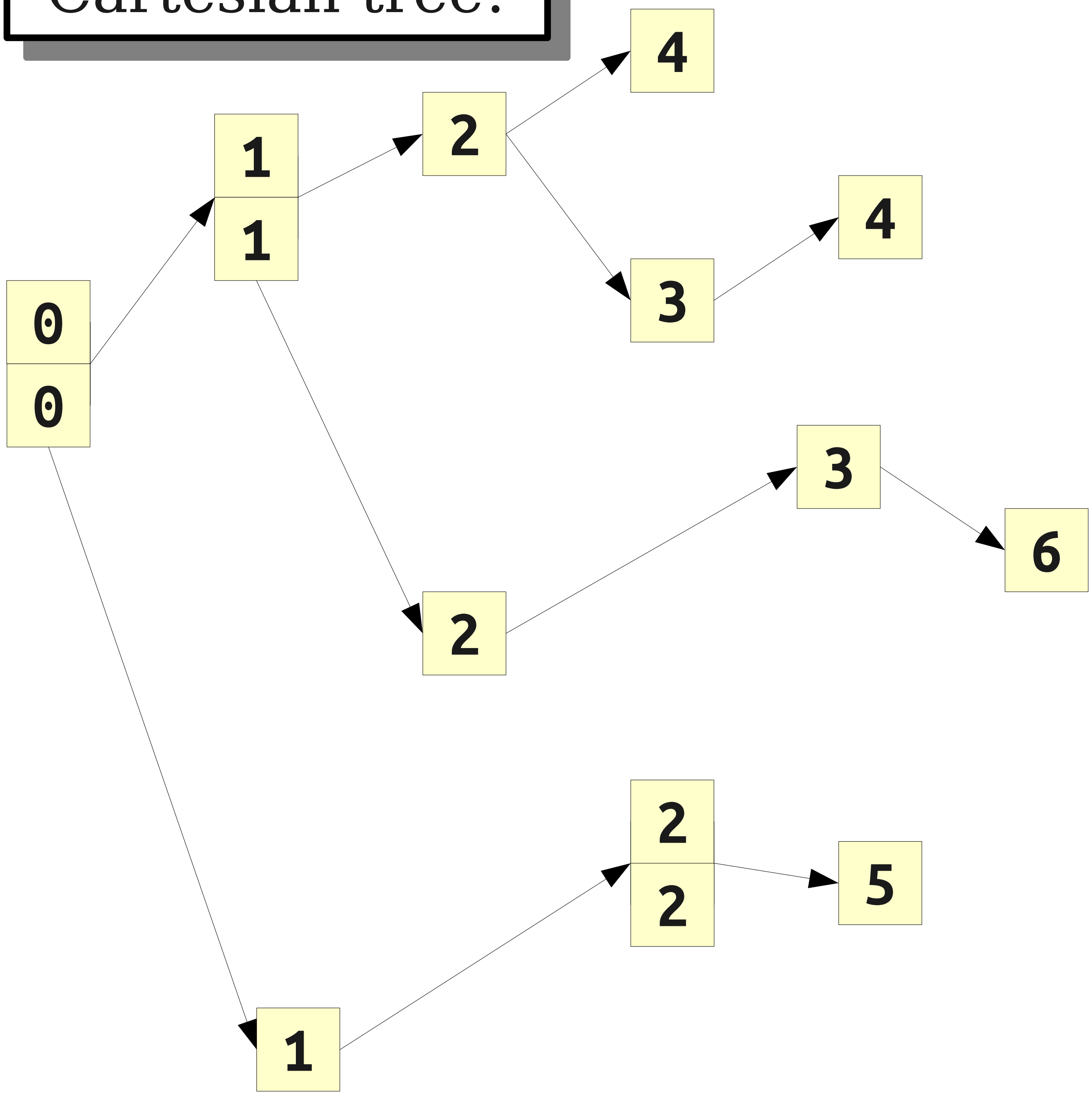
0
0
1
1
2
4
3
4
3
6
2
2
5
2
1
$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
babaaaba$
bbabaaaba$


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

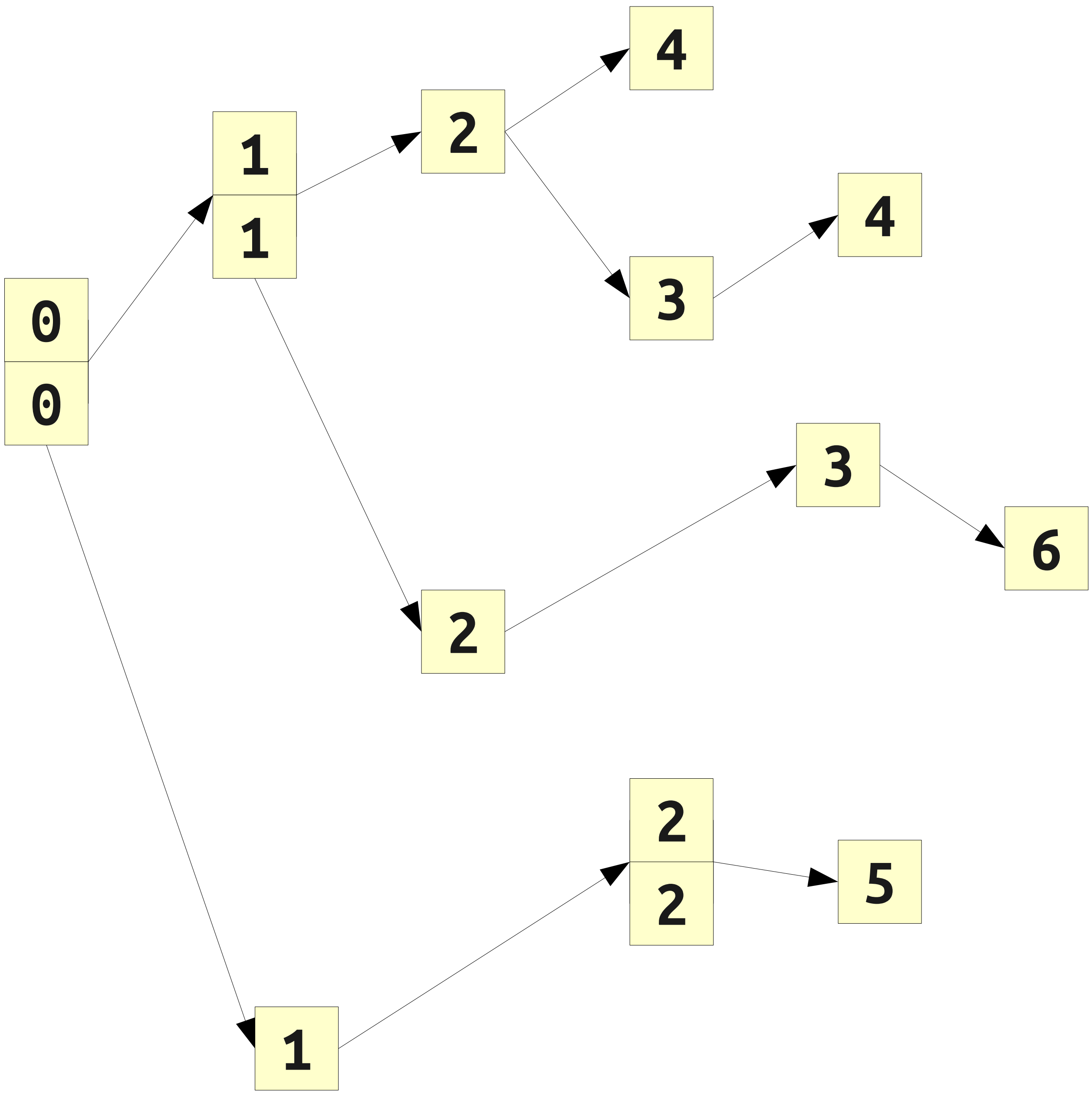


$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$

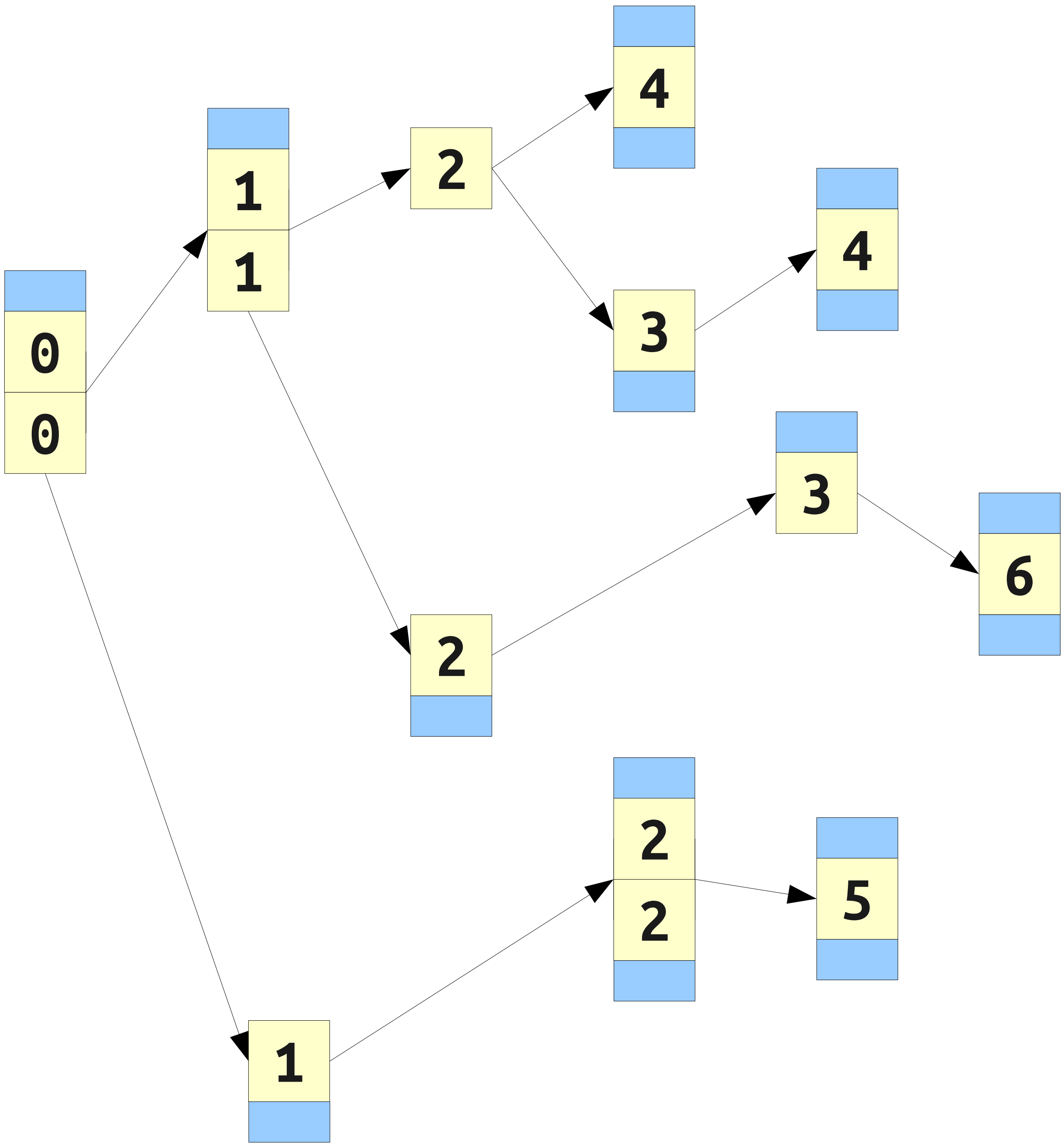
This is a slightly modified Cartesian tree!



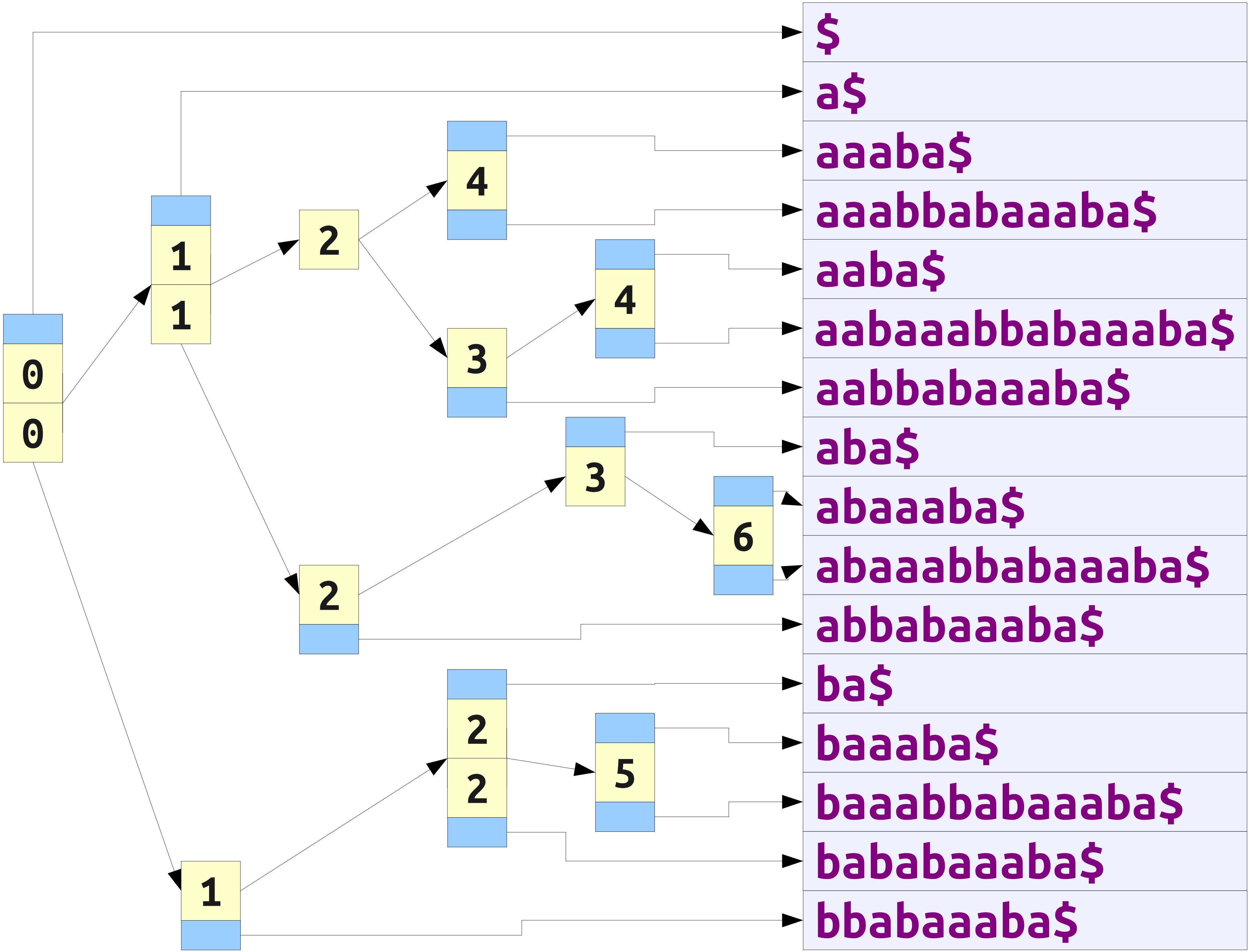
$
a$
aaaba$
aaabbabaaaba$
aaba$
aabaaabbabaaaba$
aabbabaaaba$
aba$
abaaaba$
abaaabbabaaaba$
abbabaaaba$
ba$
baaaba$
baaabbabaaaba$
bababaaaba$
bbabaaaba$



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



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



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

A Linear-Time Algorithm

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

Time-Out For Announcements!

Problem Set 6

- Problem Set 6 goes out right now. It's due next Wednesday at the start of class.
  - Play around with suffix trees, suffix arrays, and their properties!
- Problem Set 5 has been graded and will be returned at the end of lecture.

Final Project Proposals

- Proposals for the final project are due next Wednesday at the start of lecture.
- Details in the handout. Briefly:
  - Choose a project group.
  - Choose a data structure.
  - Choose a paper.
  - Choose something “interesting” to do.
- You don't need to write much, but we strongly recommend not doing this at the last minute!

Final Project Logistics

- We've updated our requirements for the final project as follows:
  - Read and summarize a research paper describing your particular data structure.
  - Choose something “interesting” that will help contribute to your overall understanding of that data structure.
  - Write a paper summarizing the original paper and describing your “interesting” contribution.
  - Give a 15-20 minute presentation about your data structure and additional work to the course staff. These presentations will be open to the public.
- Presentations will be in Week 10; more details next week.

Your Questions!

"Is there any expectation for deliverables for final projects? For example, code/slides/reports/…"

Yep! We'll provide more details soon, but you'll need to at least submit your writeup and your slides.

And the proposal. ☺

“Does memcached count as a data structure? if yes, can you give a few suggestions for final project on memcached?”

From what I've read, this would not count as a data structure. It's more of a distributed system. Sorry!

The Hard Part: Building Suffix Arrays

Building Suffix Arrays

- Suffix arrays can be constructed in linear time via a variety of different algorithms.
- The algorithm we'll cover today is called **DC3** (**D**ifference **C**over, size **3**) and is a beautiful (but tricky!) divide-and-conquer algorithm.
- Rather than starting off there, let's begin with some slower solutions.

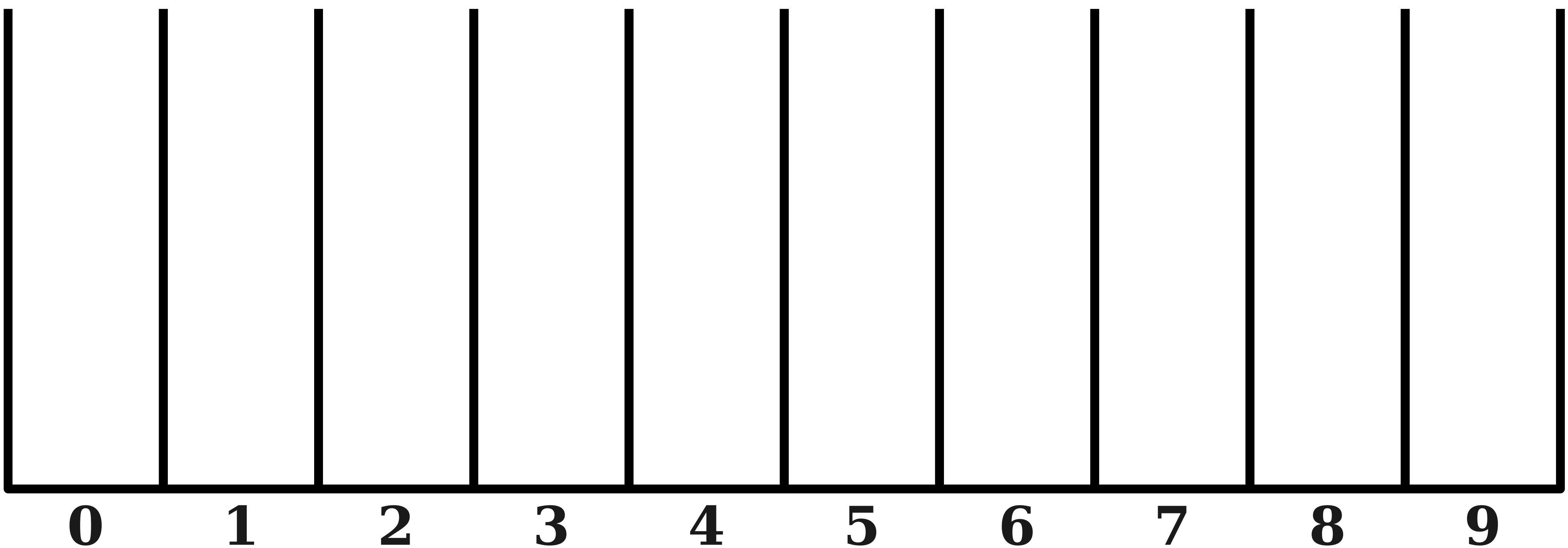
A Naïve Algorithm

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

Speeding up with Radix Sort

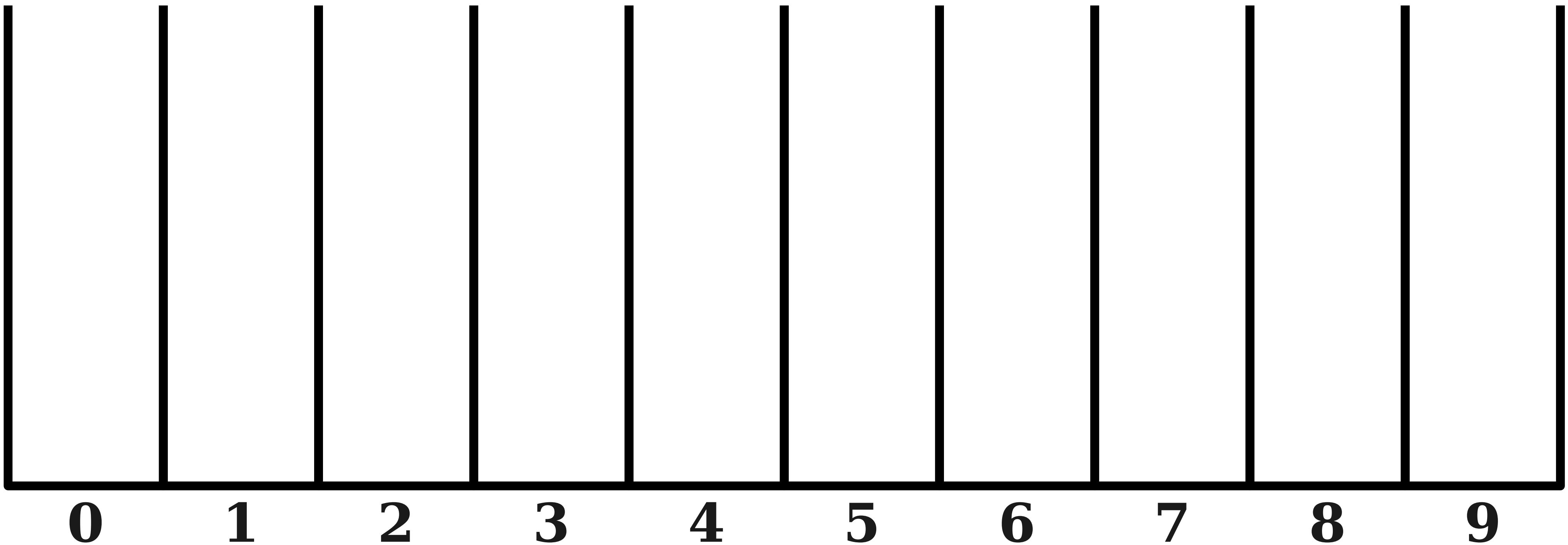
- We can improve the performance of this algorithm by using **radix sort**.
- Radix sort is a string sorting algorithm that runs in time $O(k(m + |\Sigma|))$, where
  - m is the number of strings,
  - k is the maximum length of each string, and
  - Σ is the alphabet in question.
- Assumes the alphabet consists of integers between 0 and $|\Sigma|$ - 1, inclusive.

Radix Sort



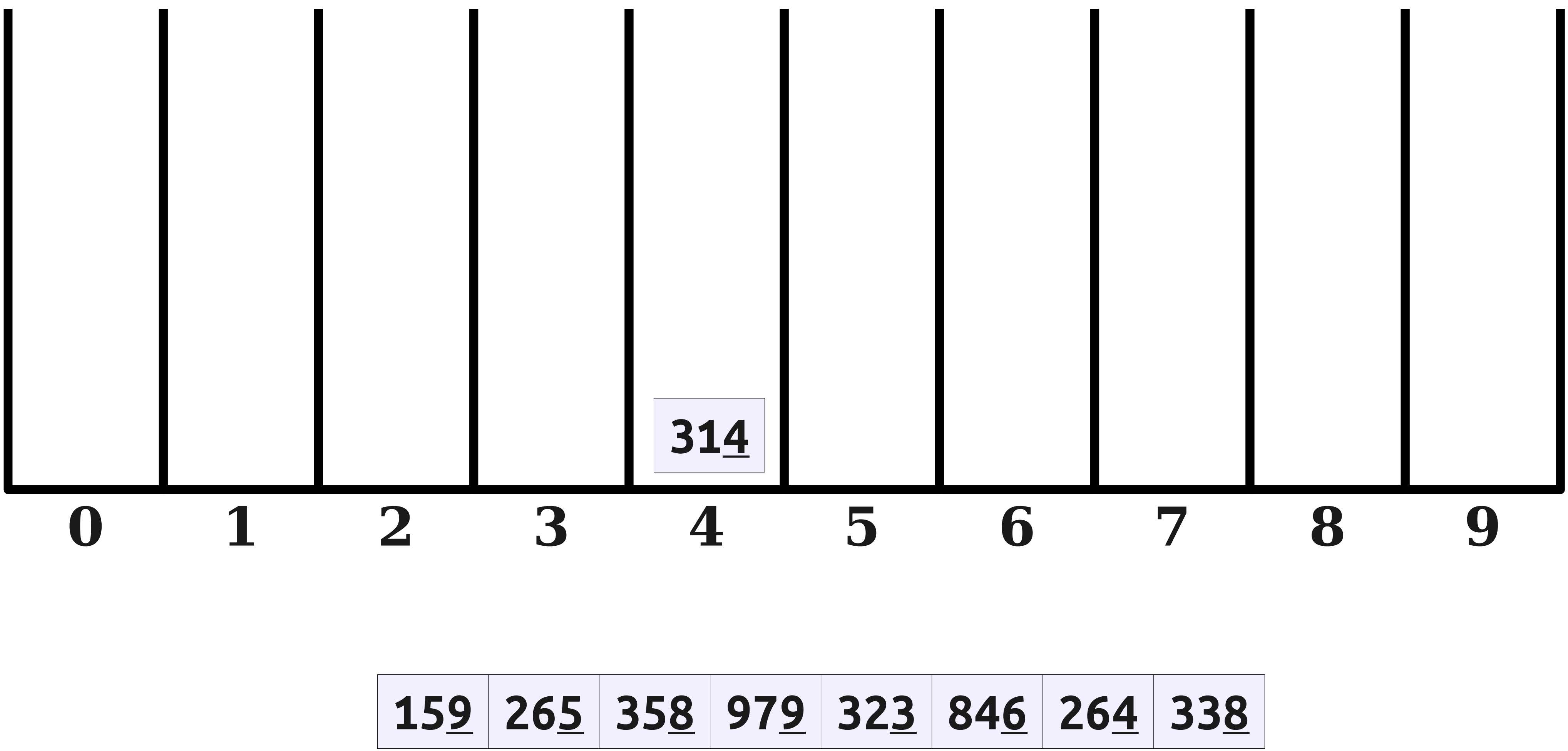
314	159	265	358	979	323	846	264	338

Radix Sort

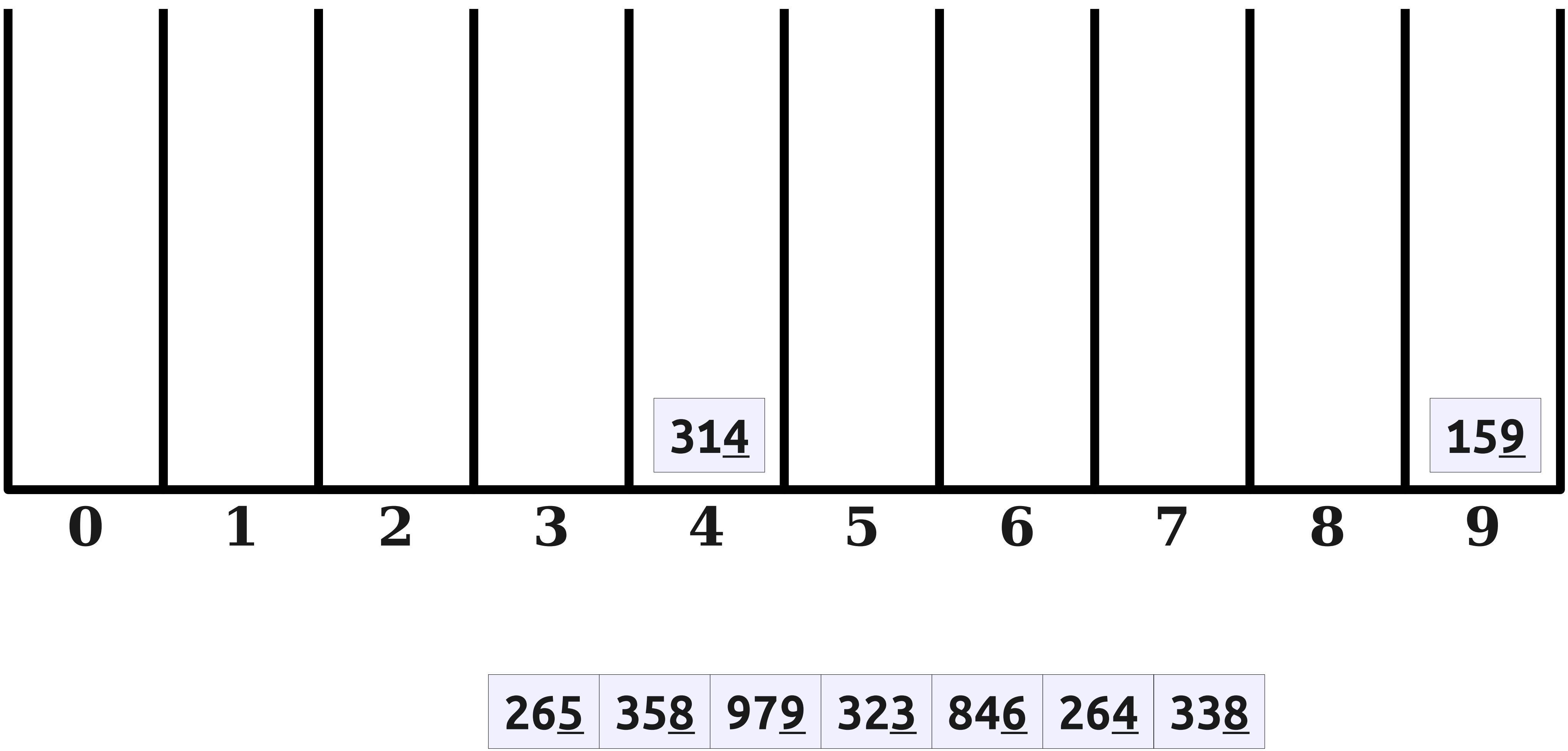


314	159	265	358	979	323	846	264	338

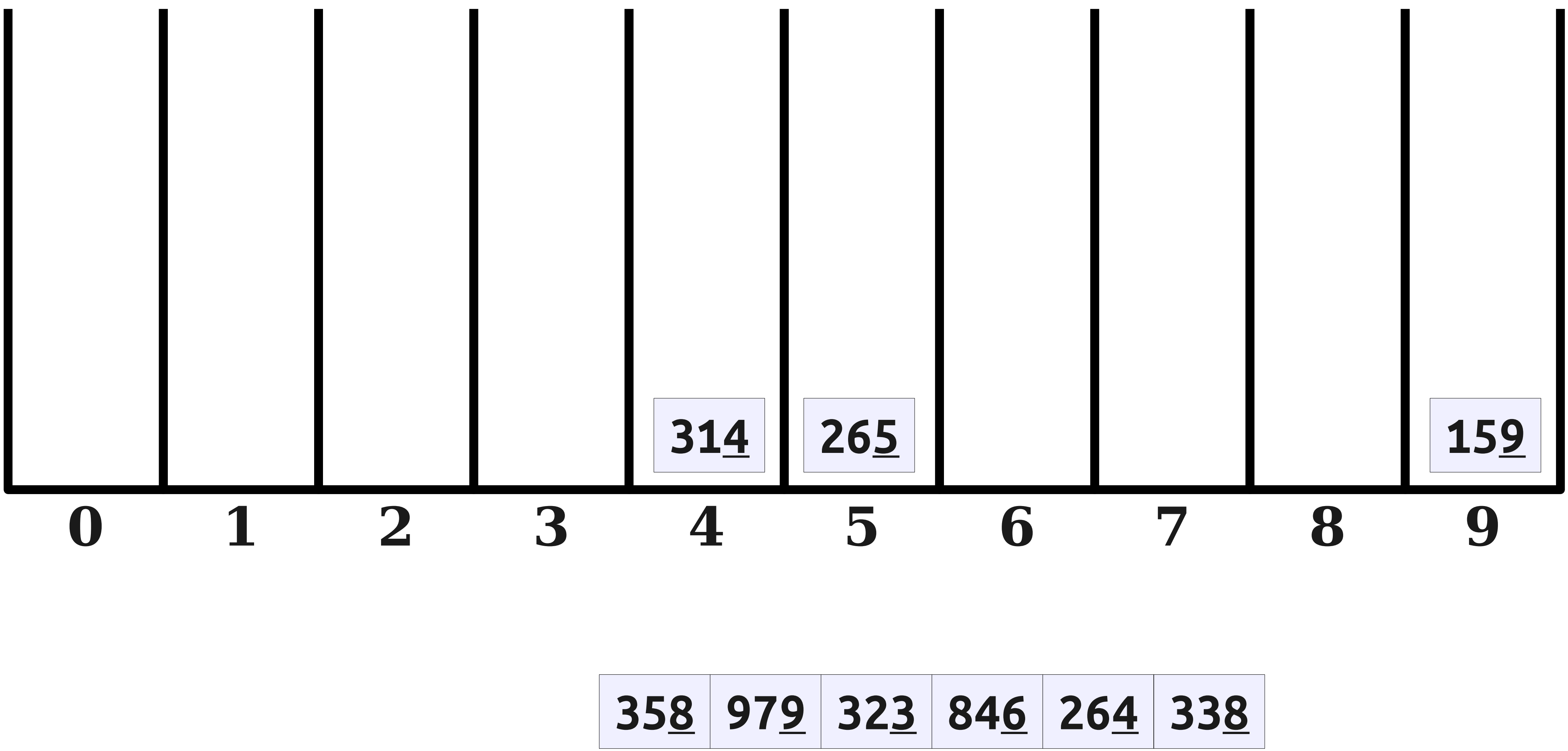
Radix Sort



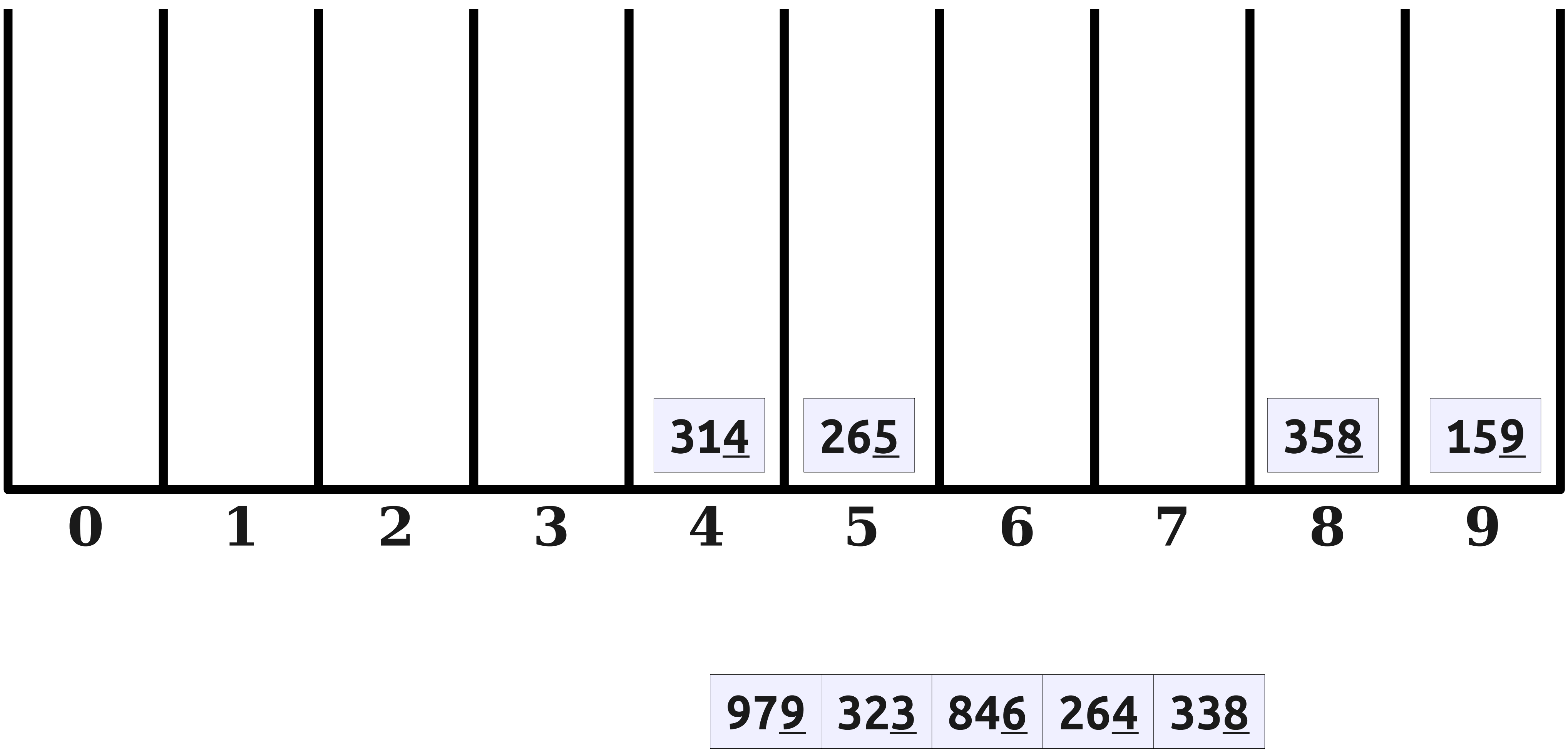
Radix Sort



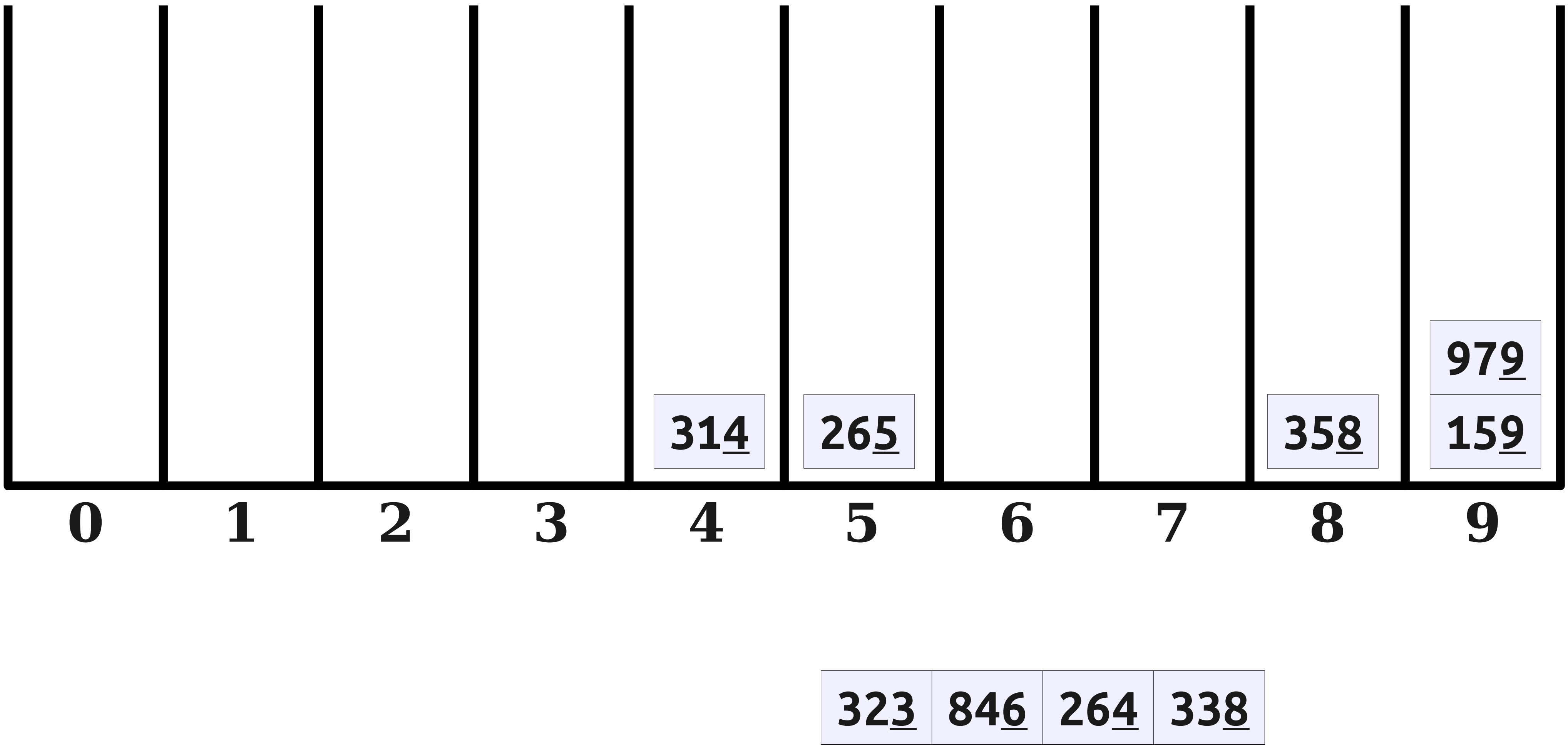
Radix Sort



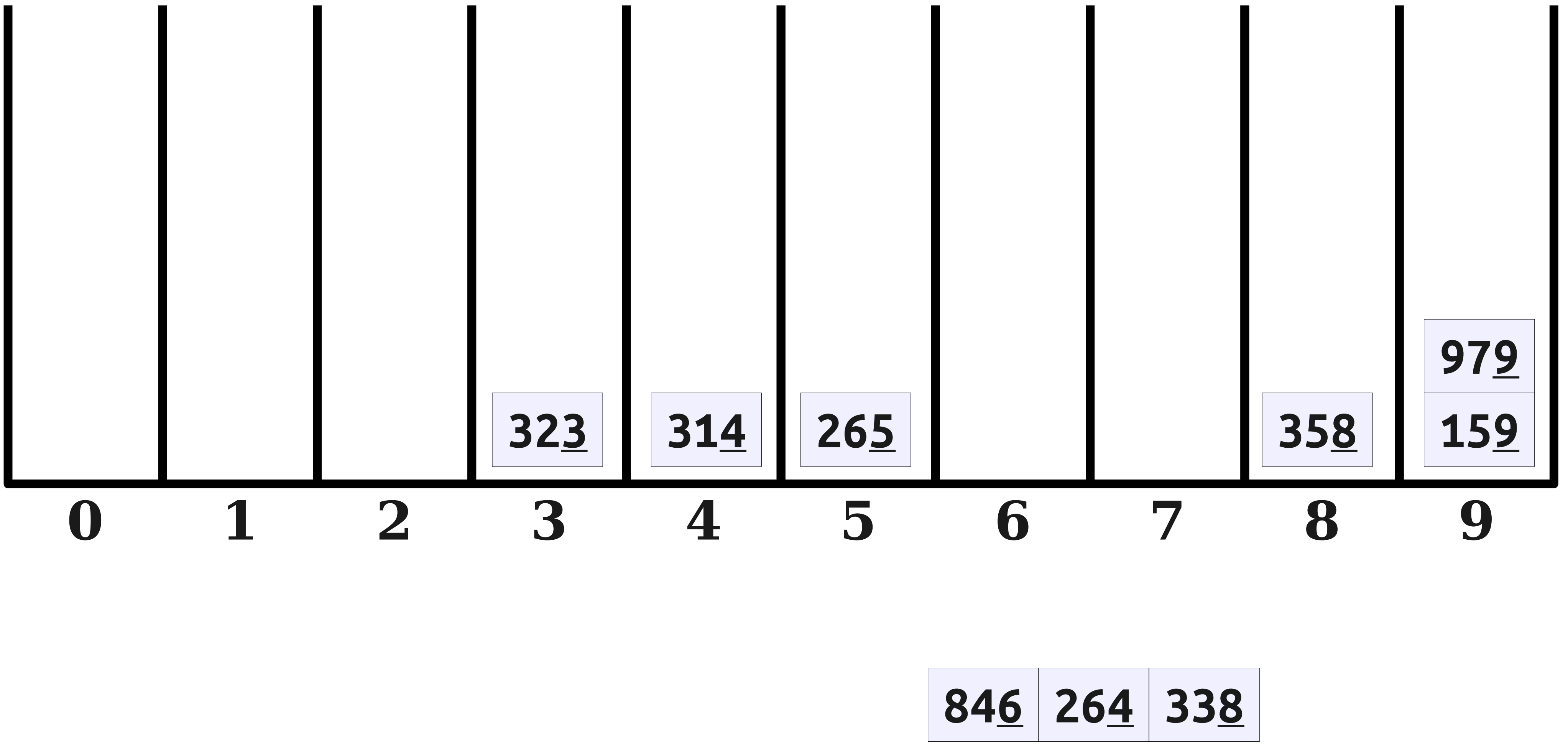
Radix Sort



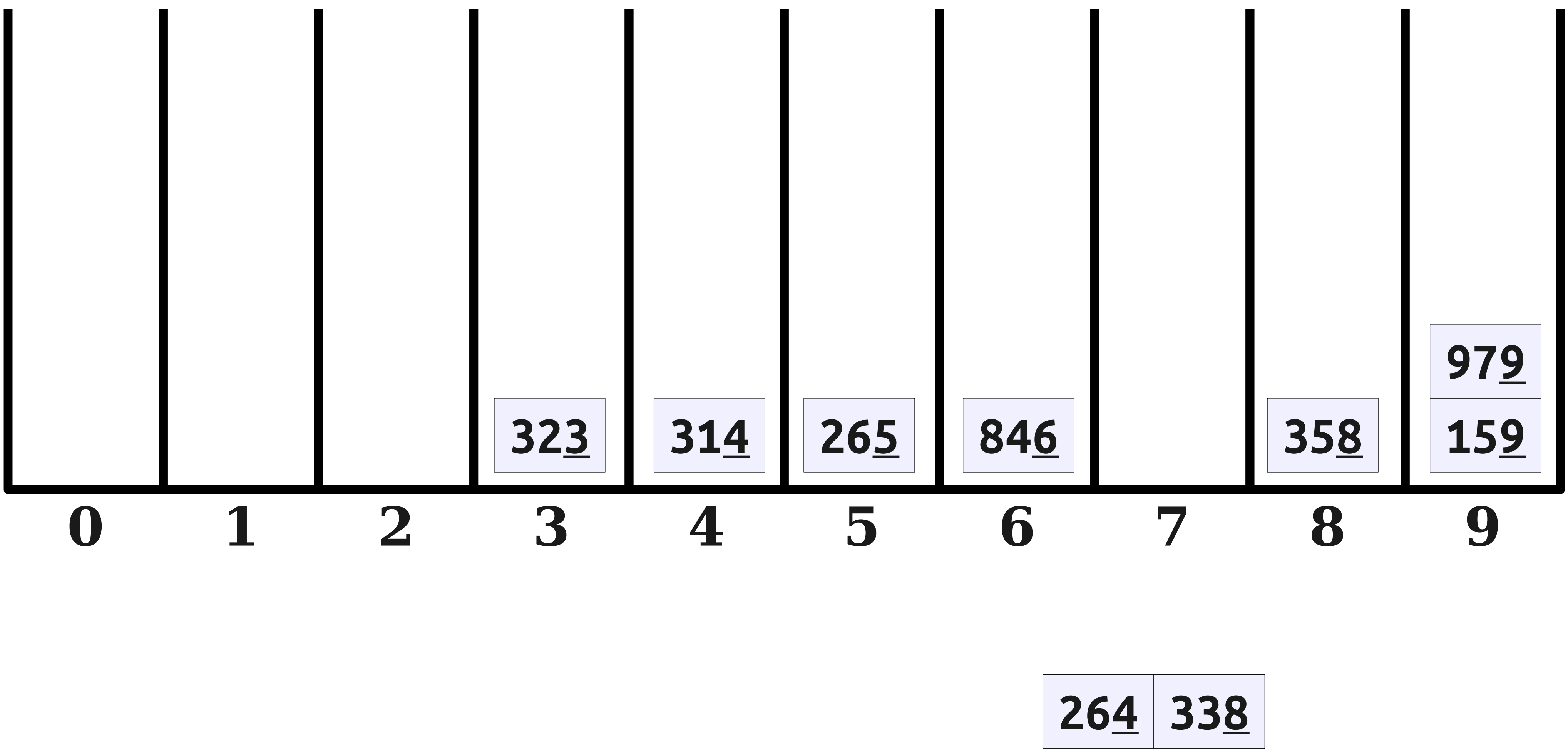
Radix Sort



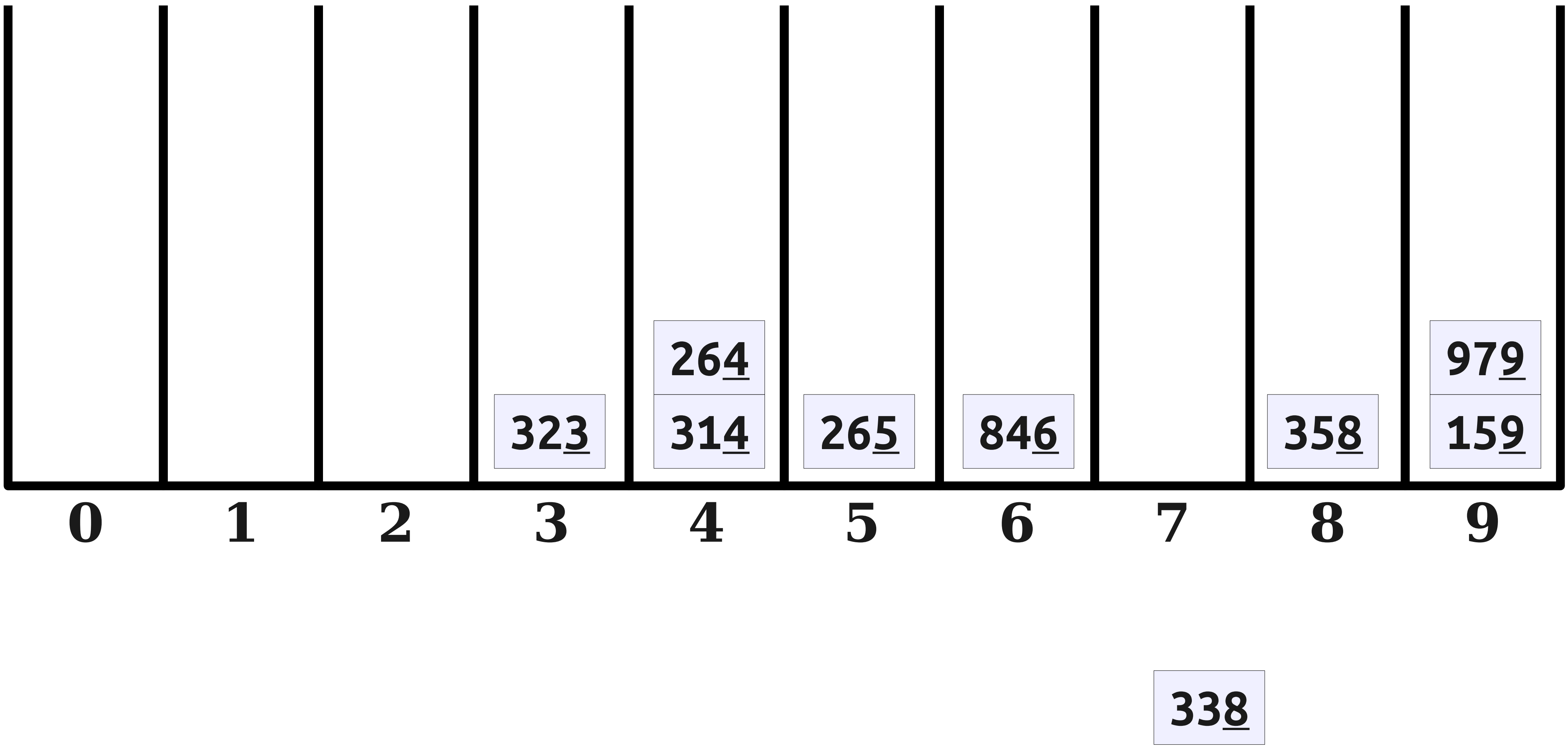
Radix Sort



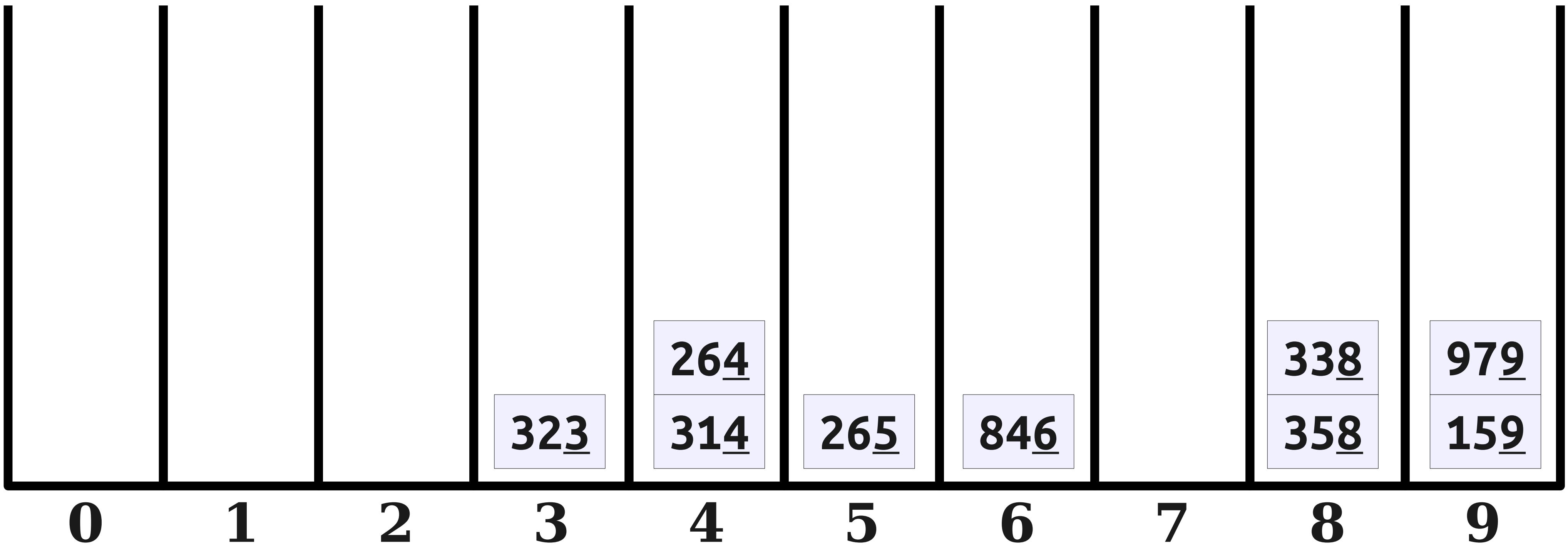
Radix Sort



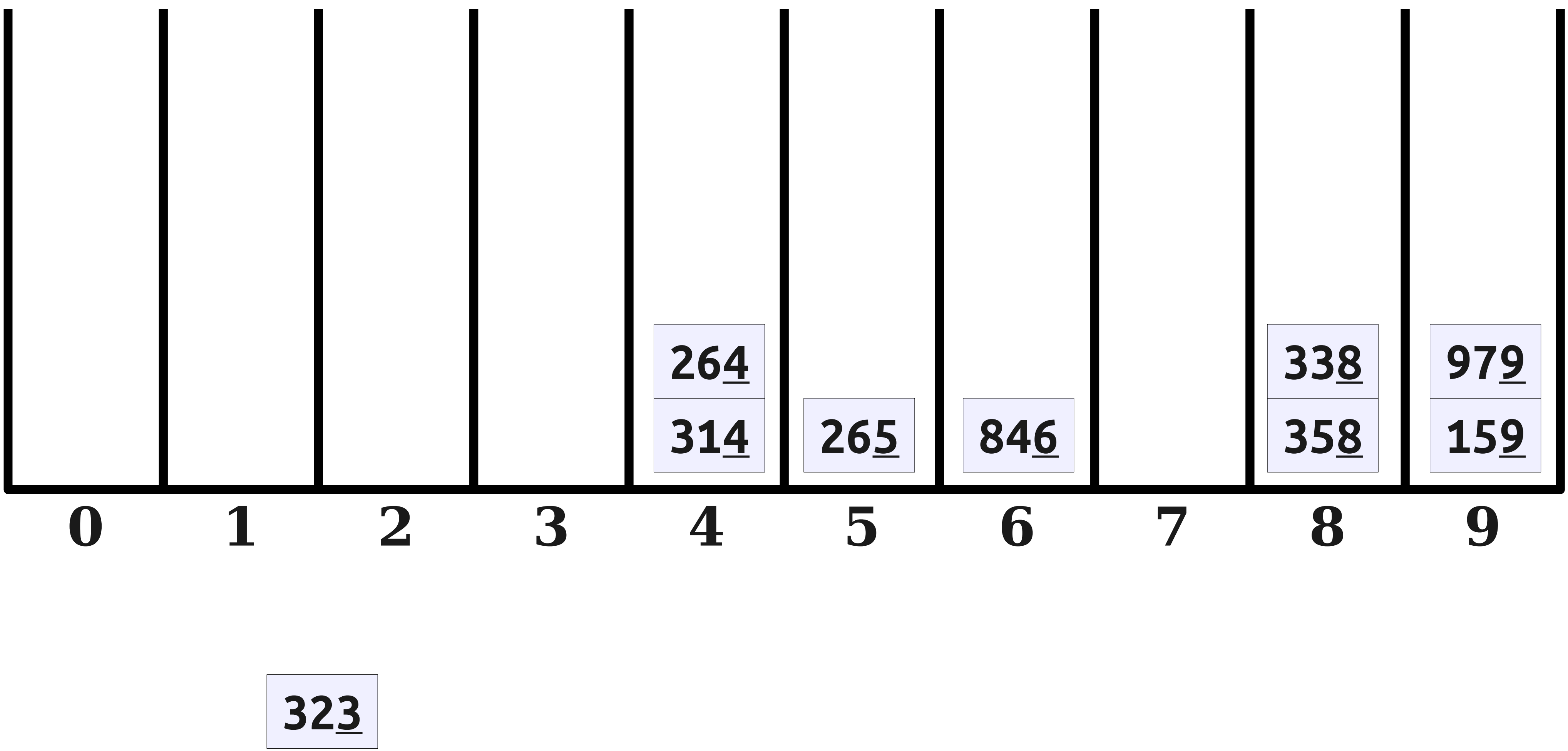
Radix Sort



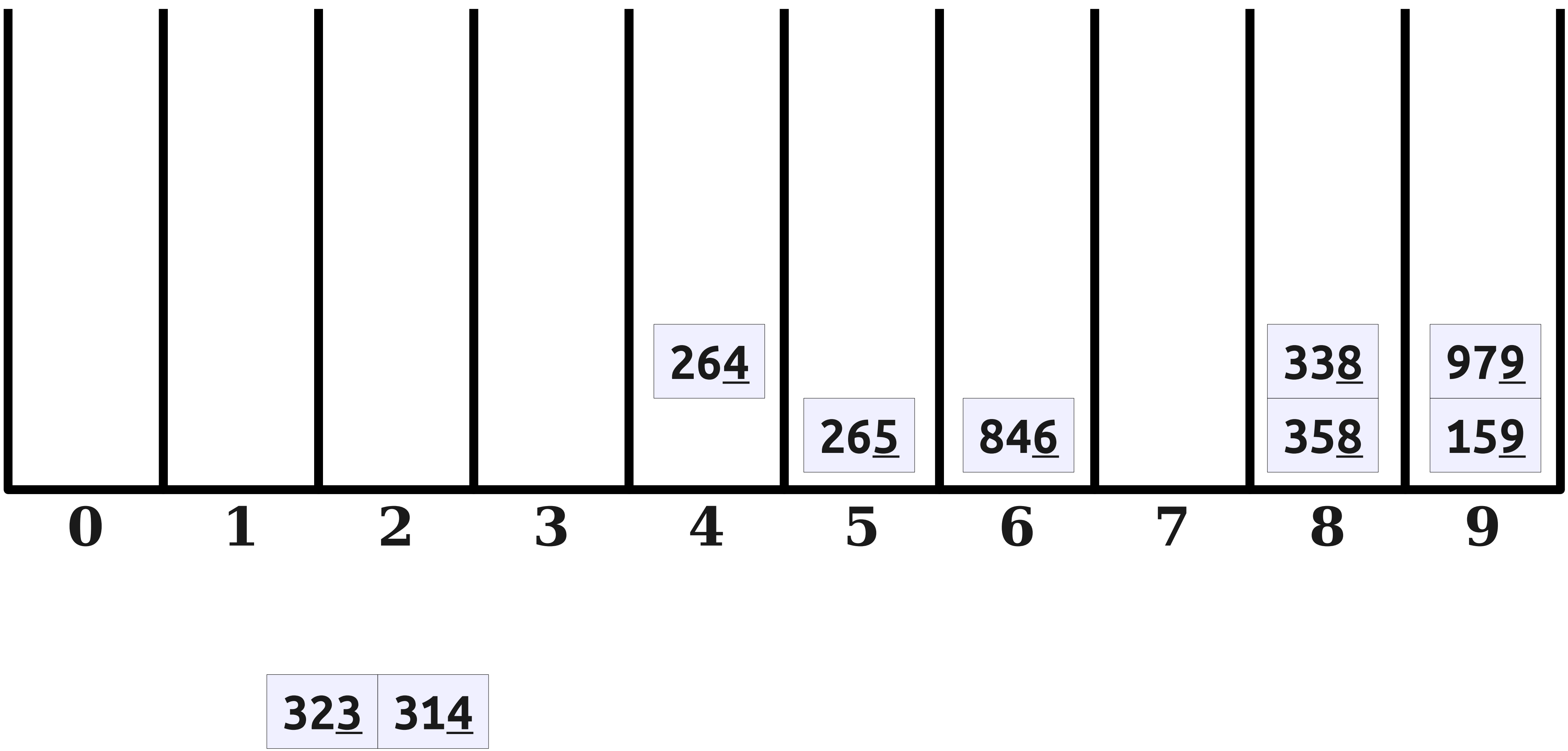
Radix Sort



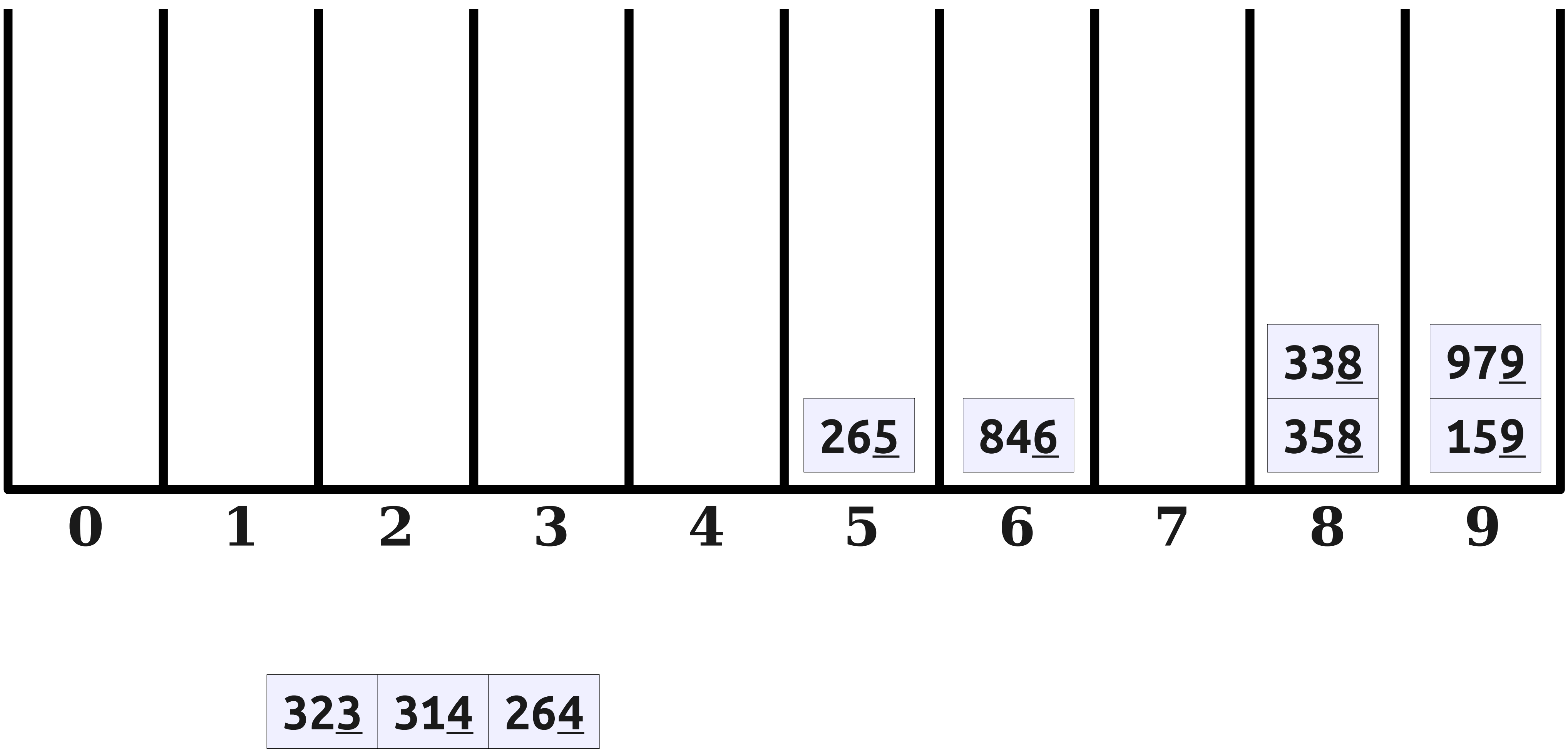
Radix Sort



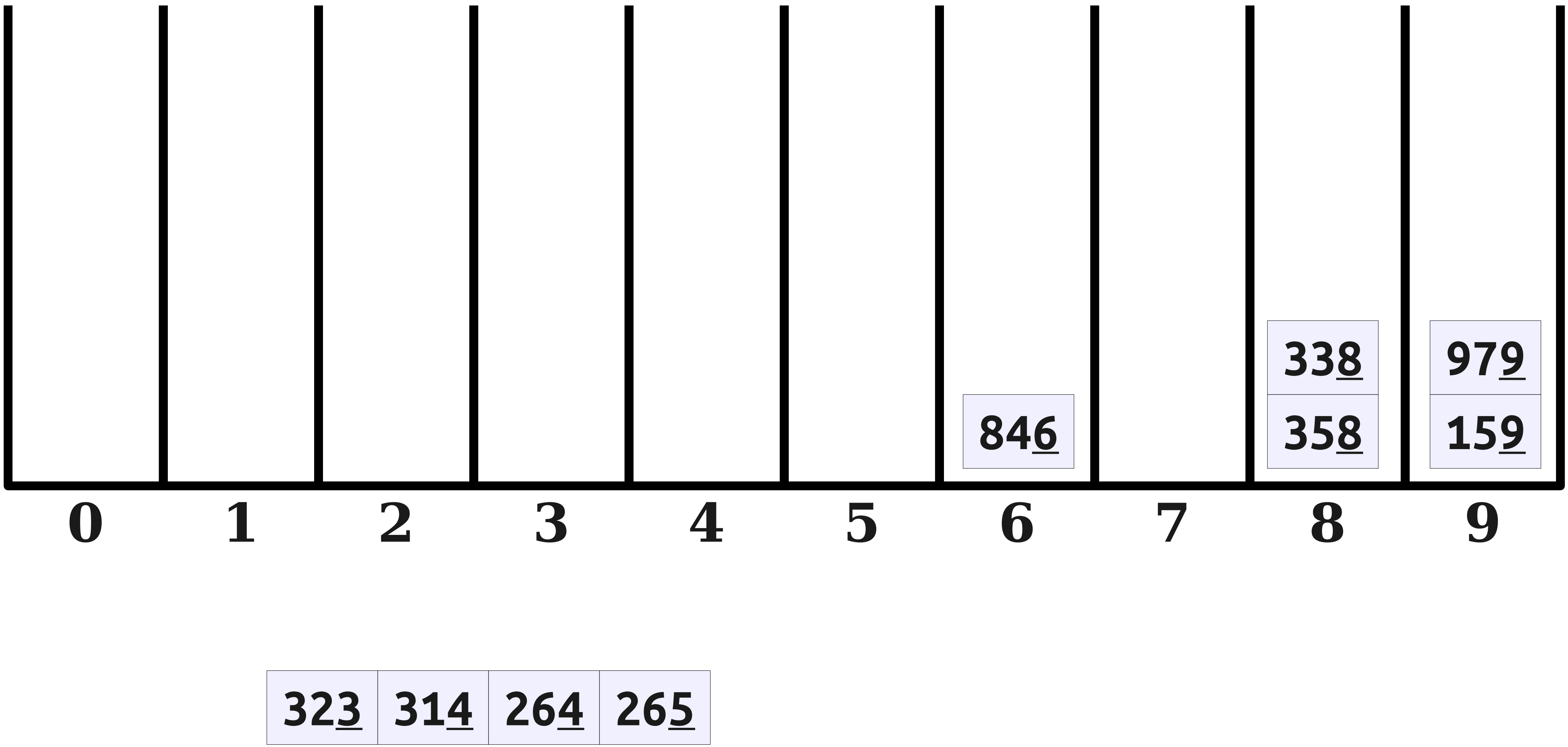
Radix Sort



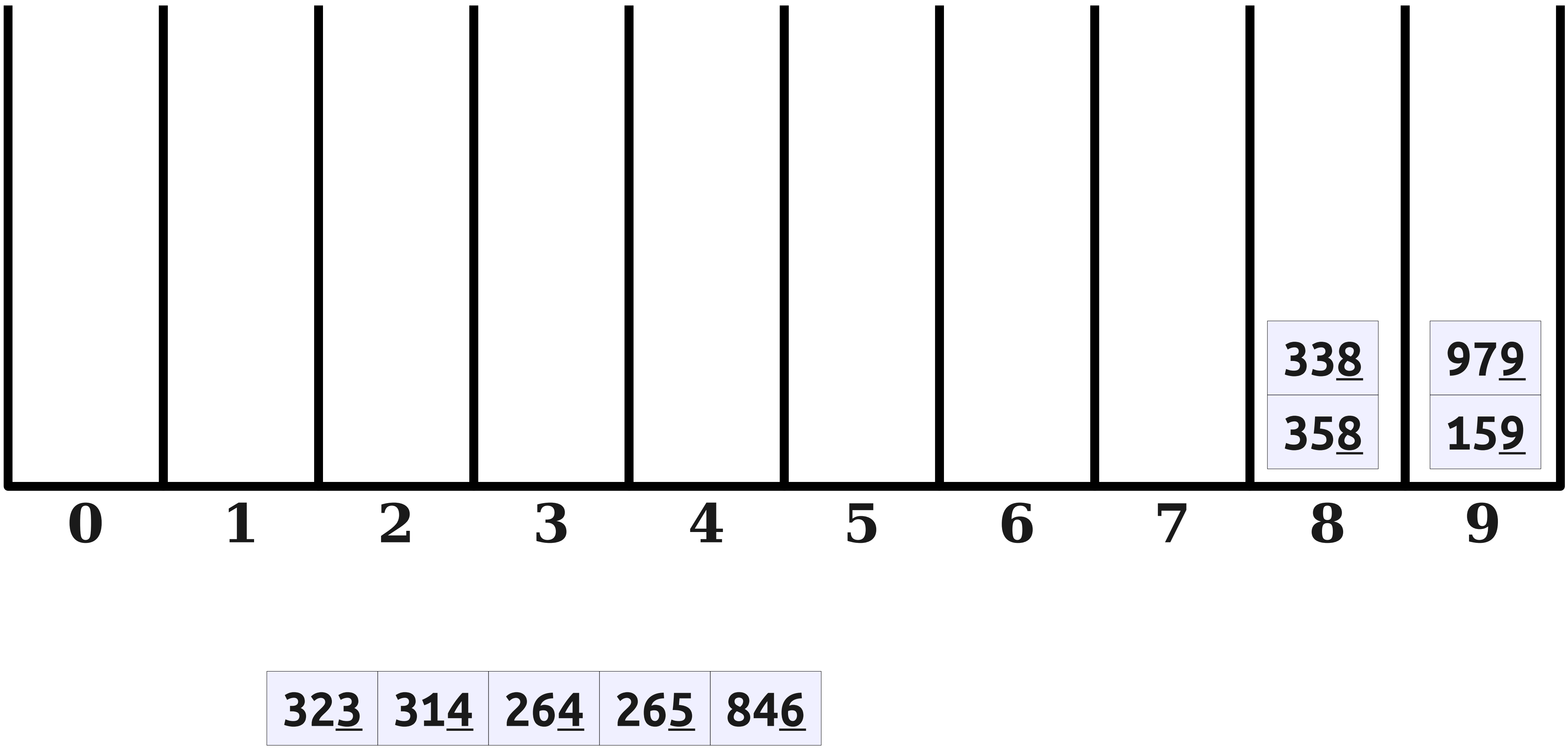
Radix Sort



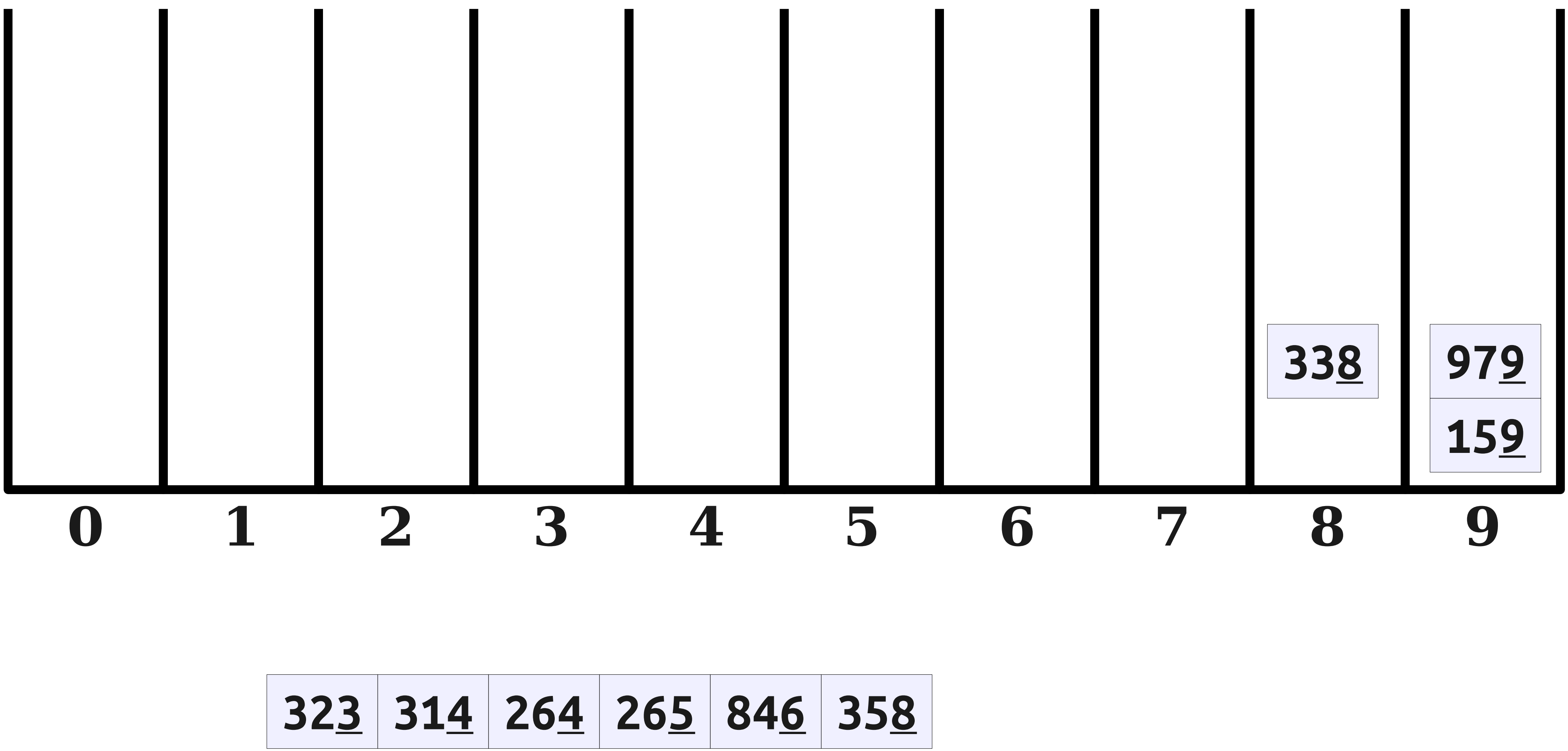
Radix Sort



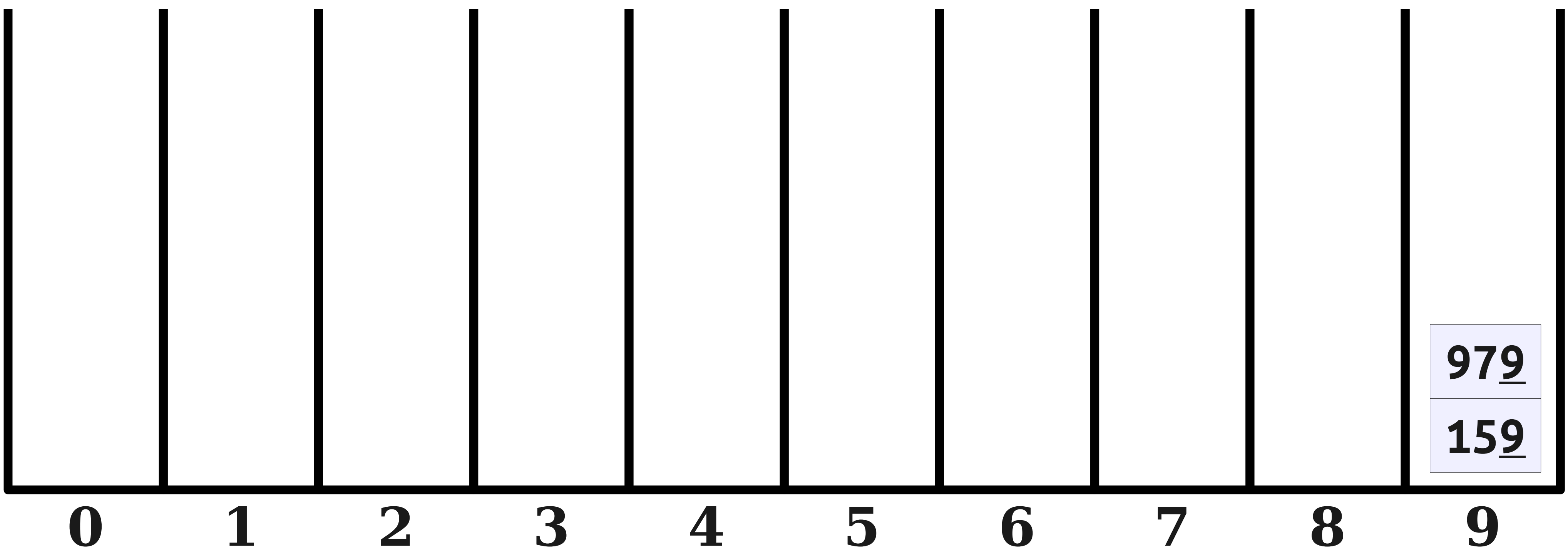
Radix Sort



Radix Sort

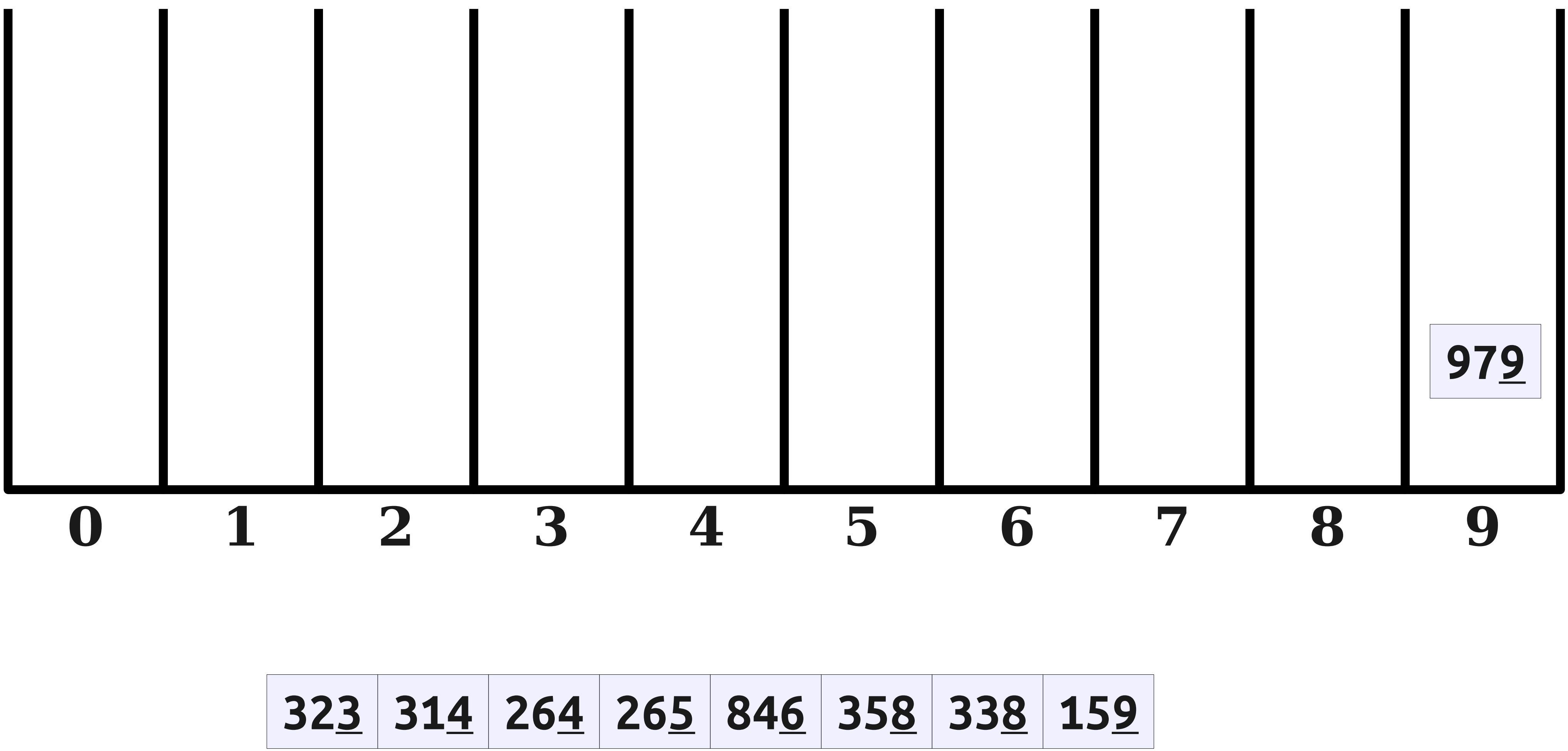


Radix Sort

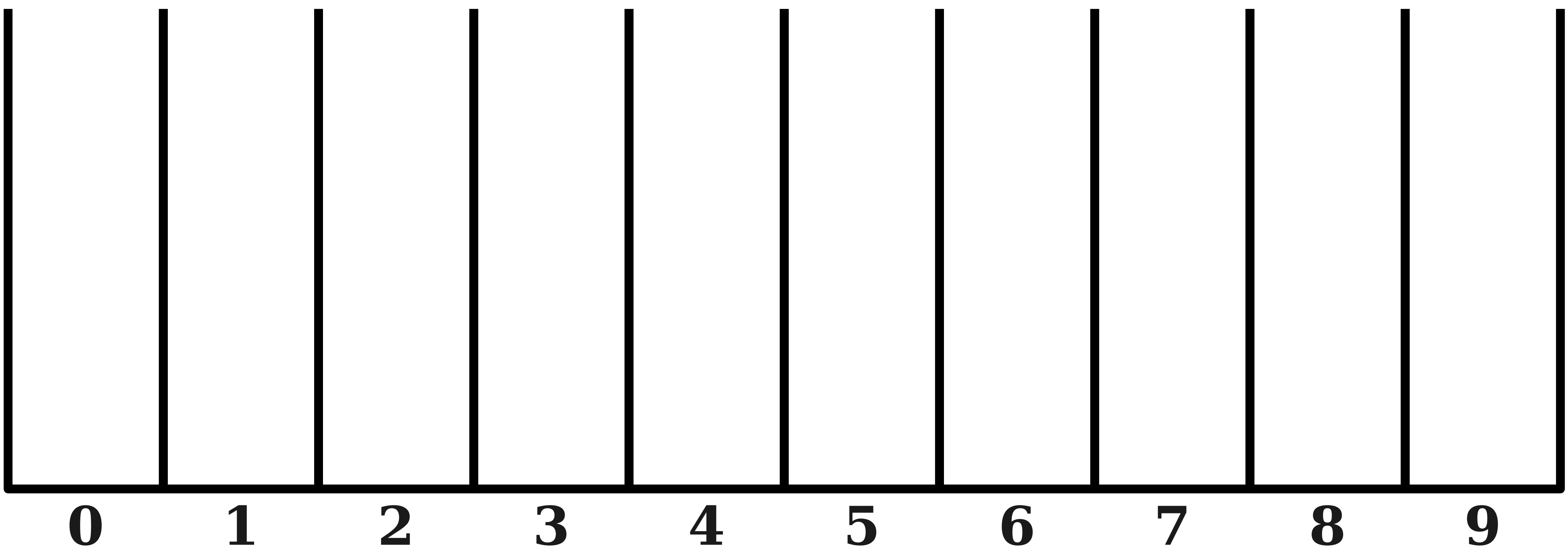


323	314	264	265	846	358	338

Radix Sort

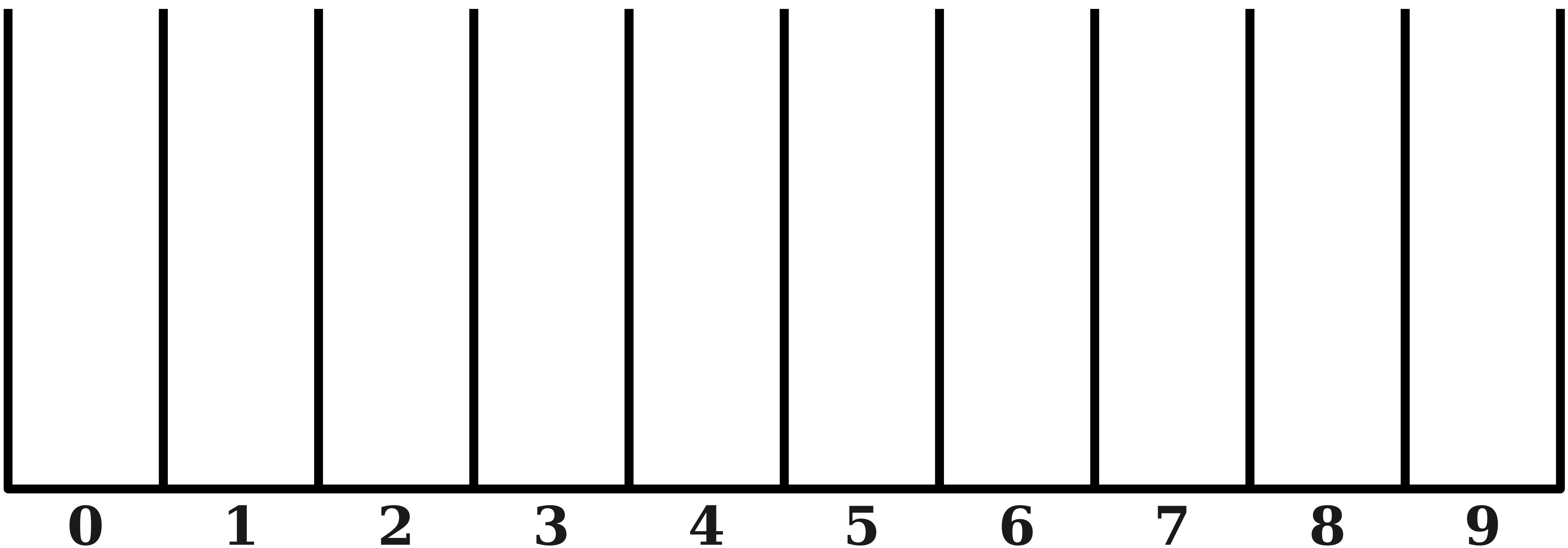


Radix Sort



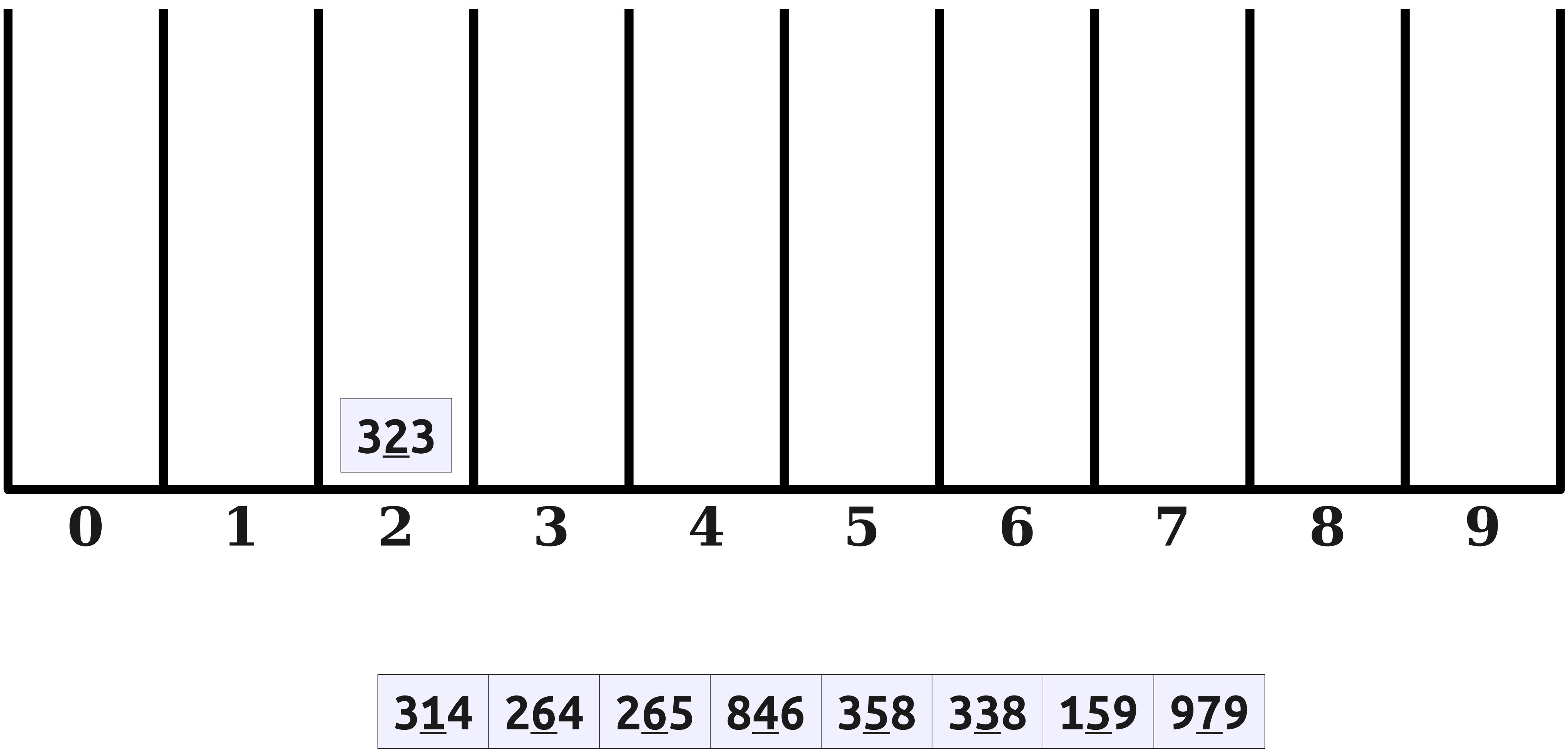
323	314	264	265	846	358	338	159	979

Radix Sort

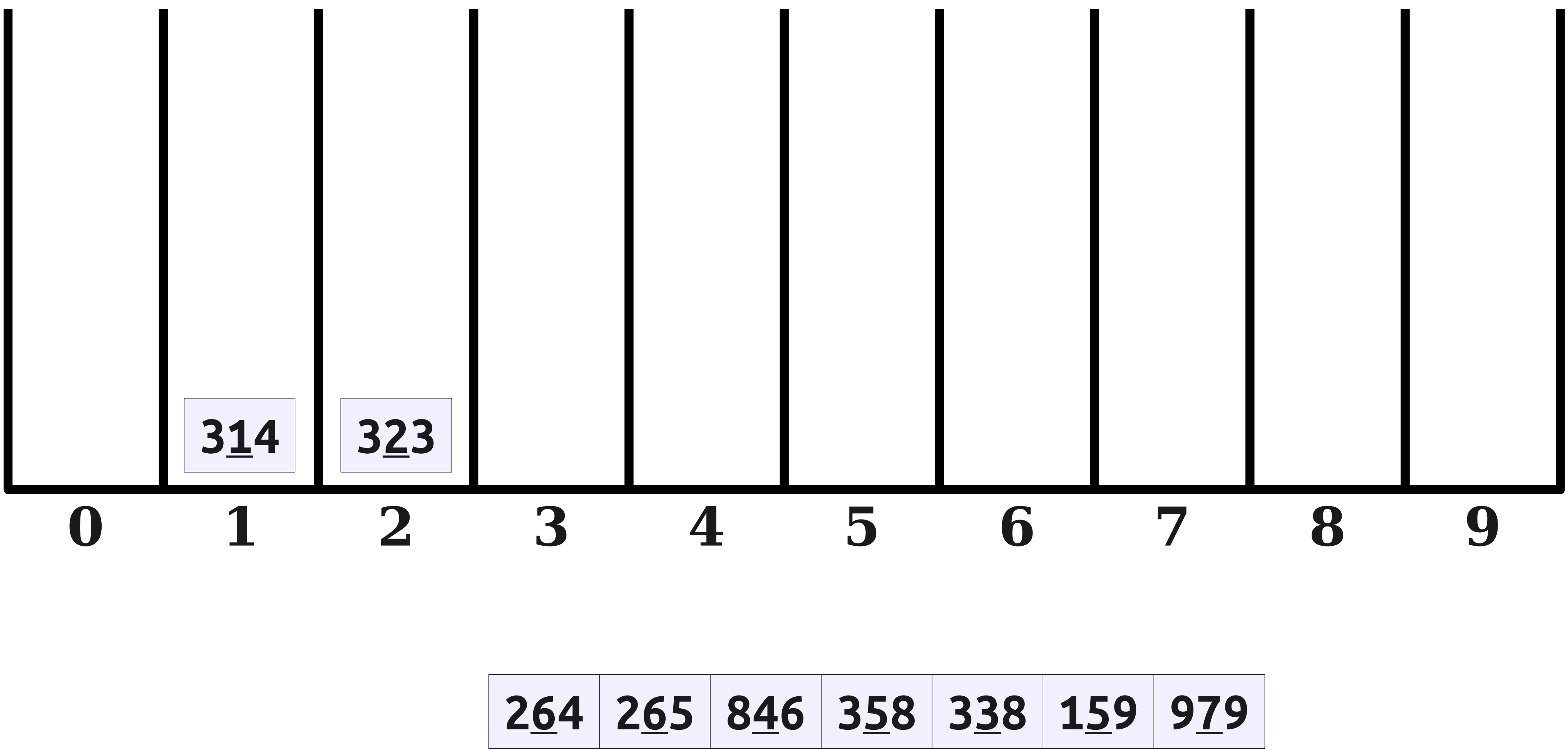


323	314	264	265	846	358	338	159	979

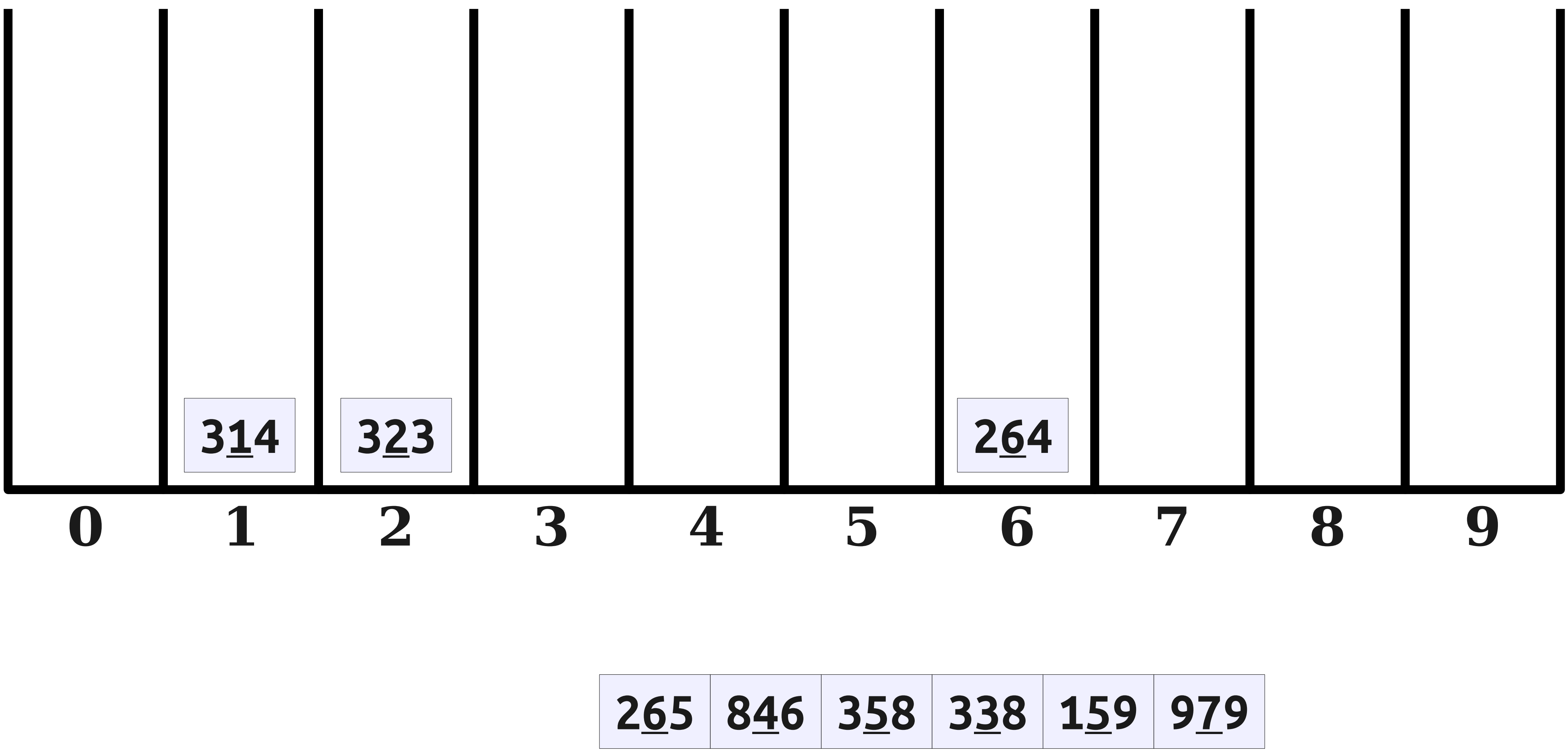
Radix Sort



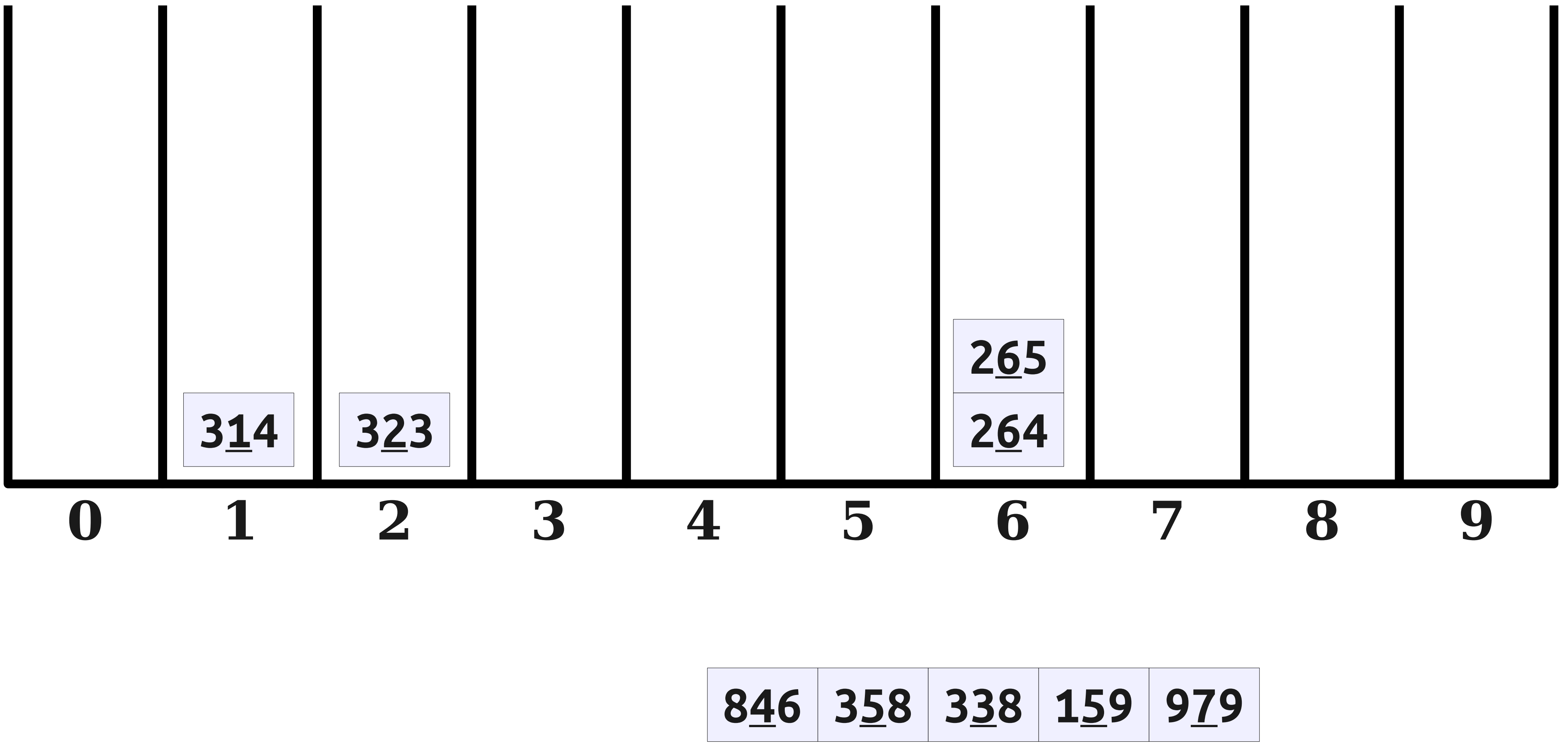
Radix Sort



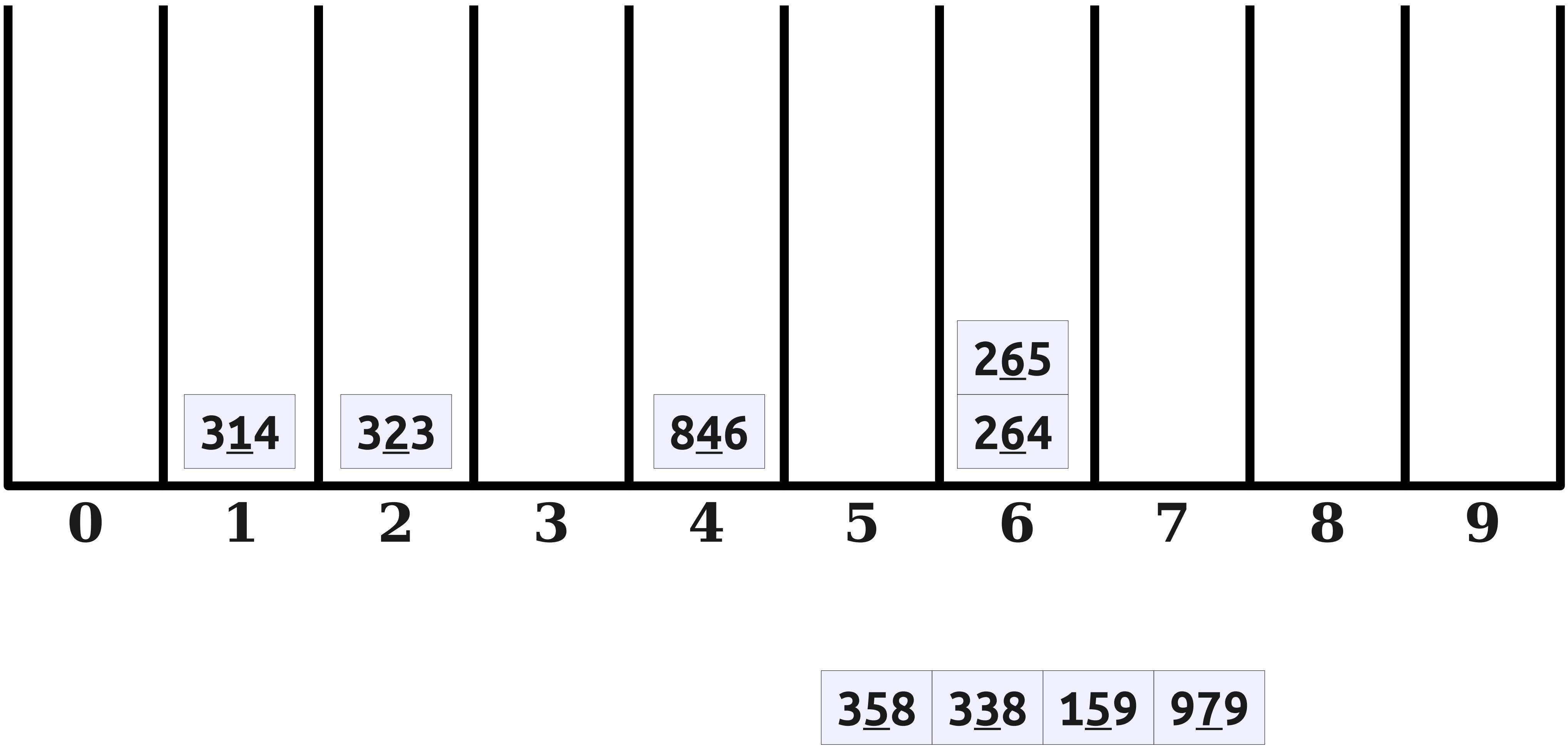
Radix Sort



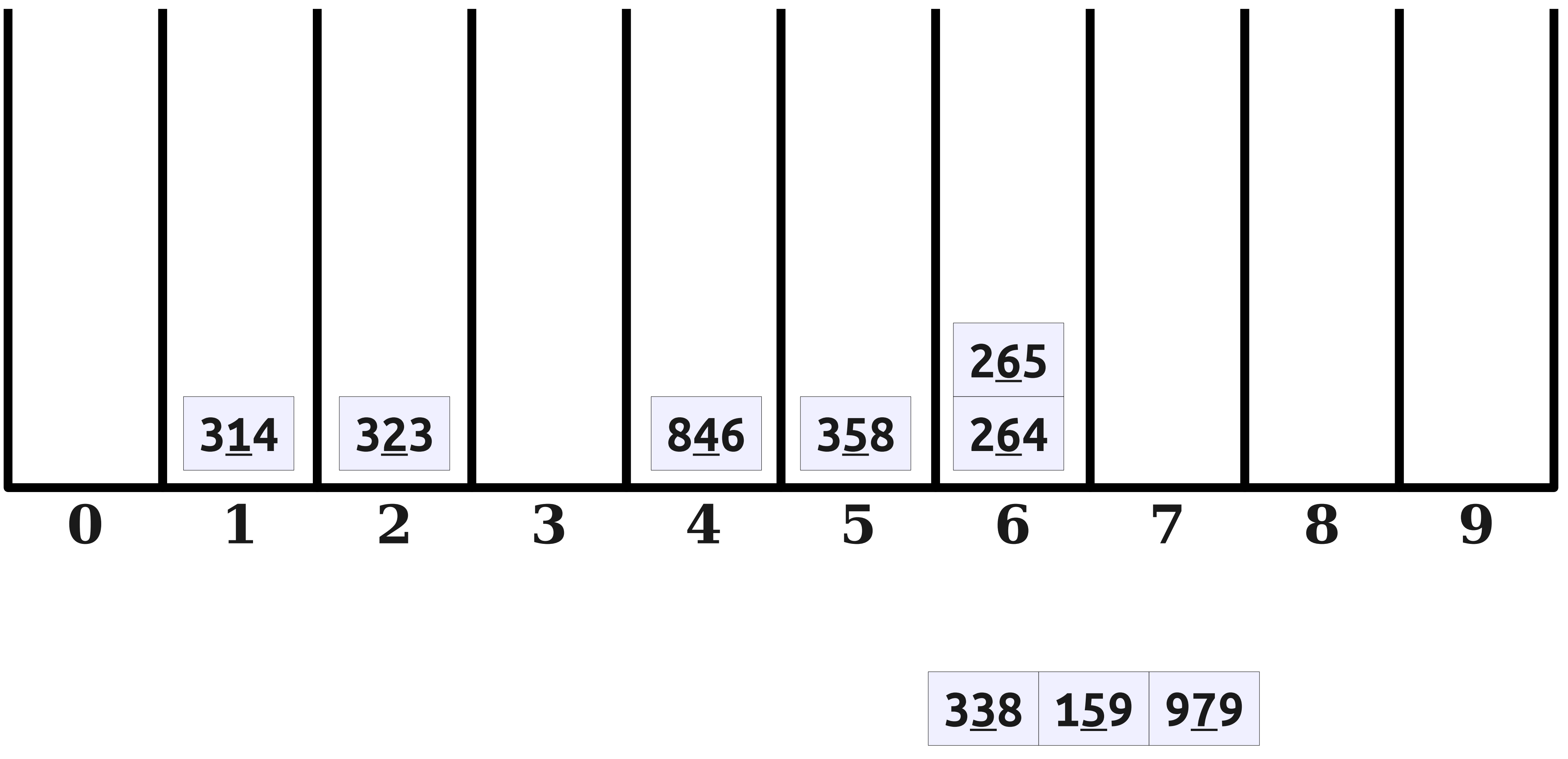
Radix Sort



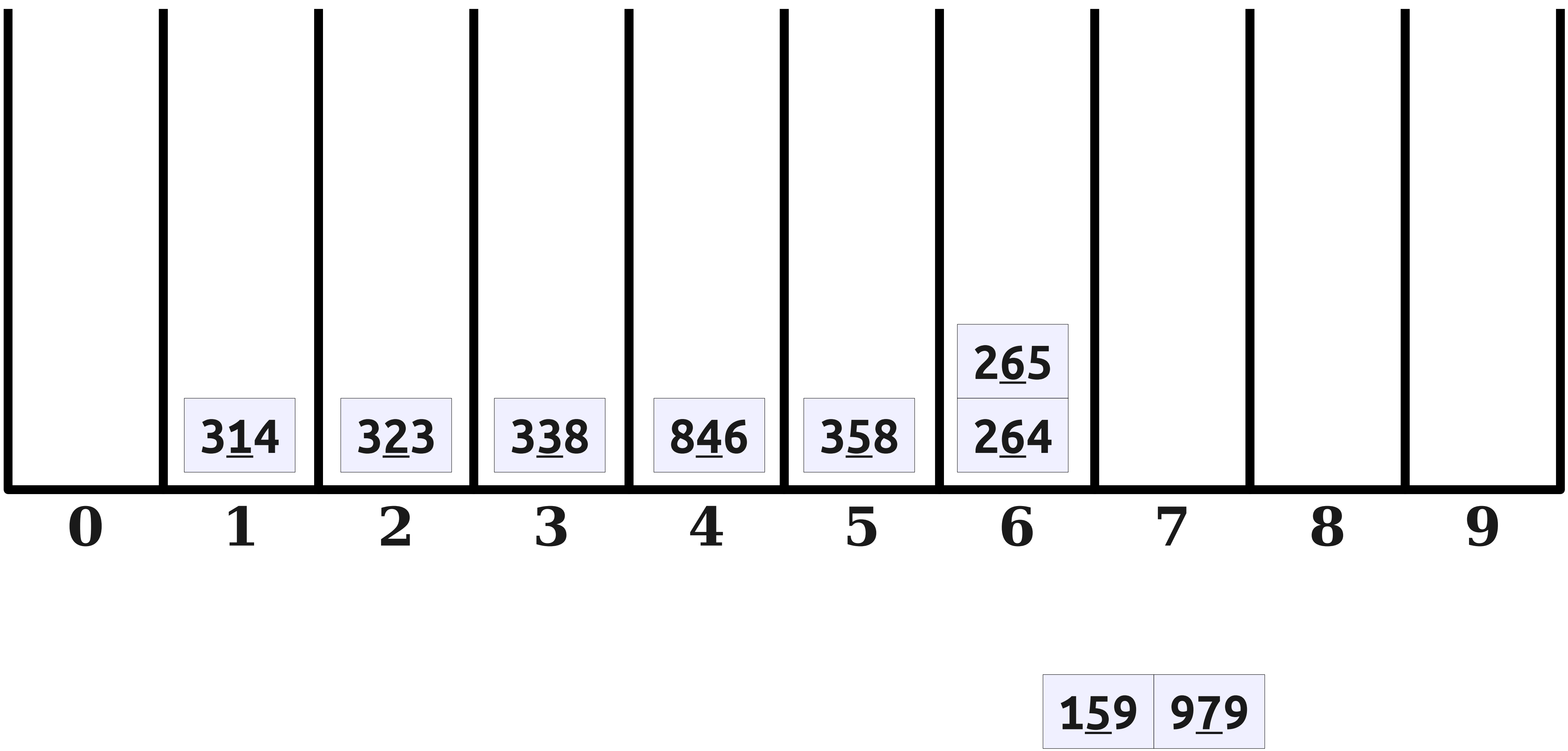
Radix Sort



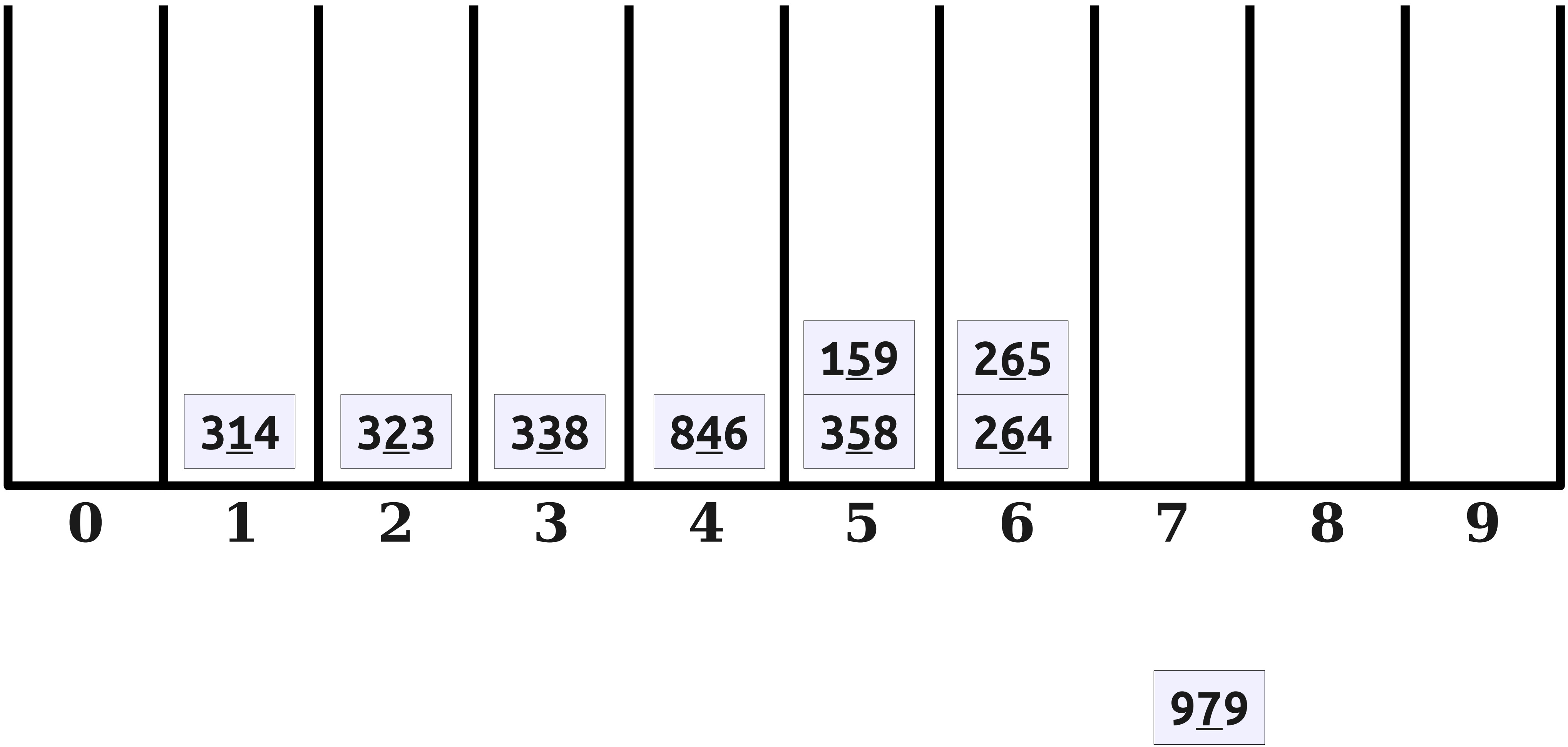
Radix Sort



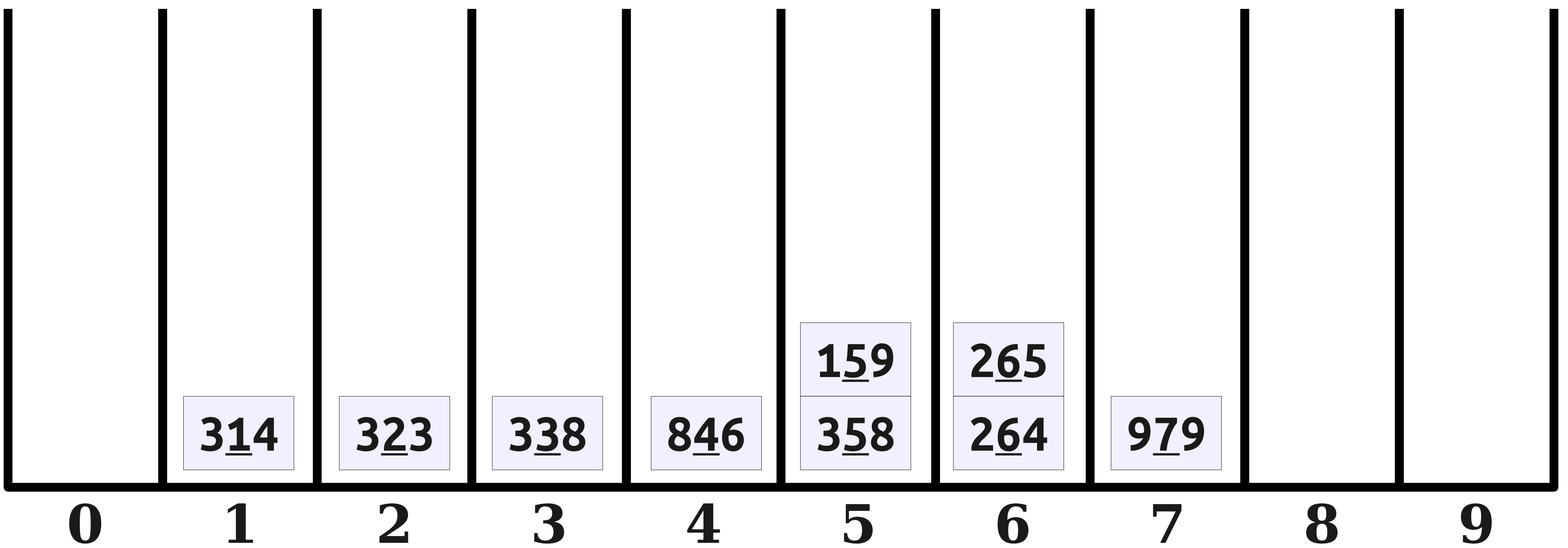
Radix Sort



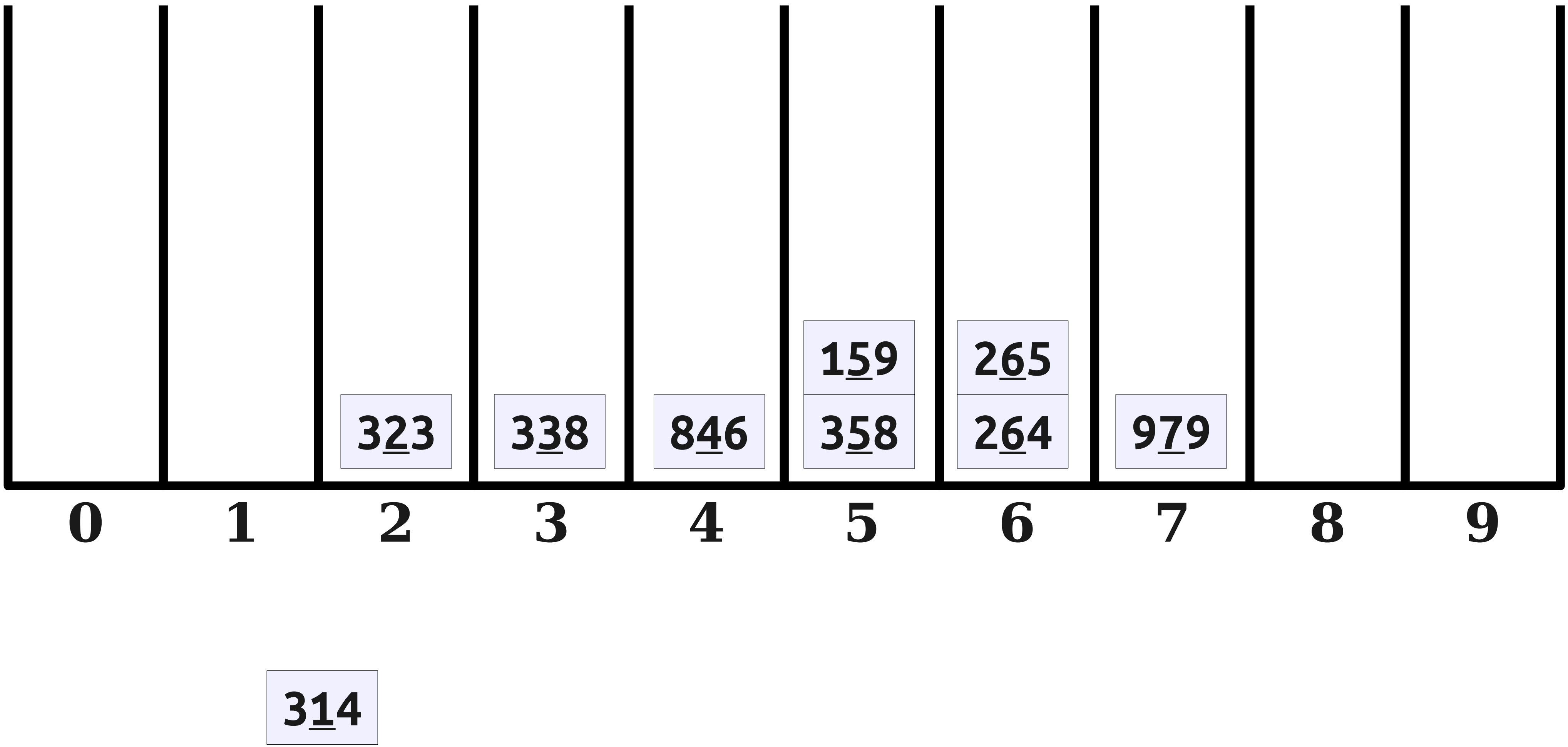
Radix Sort



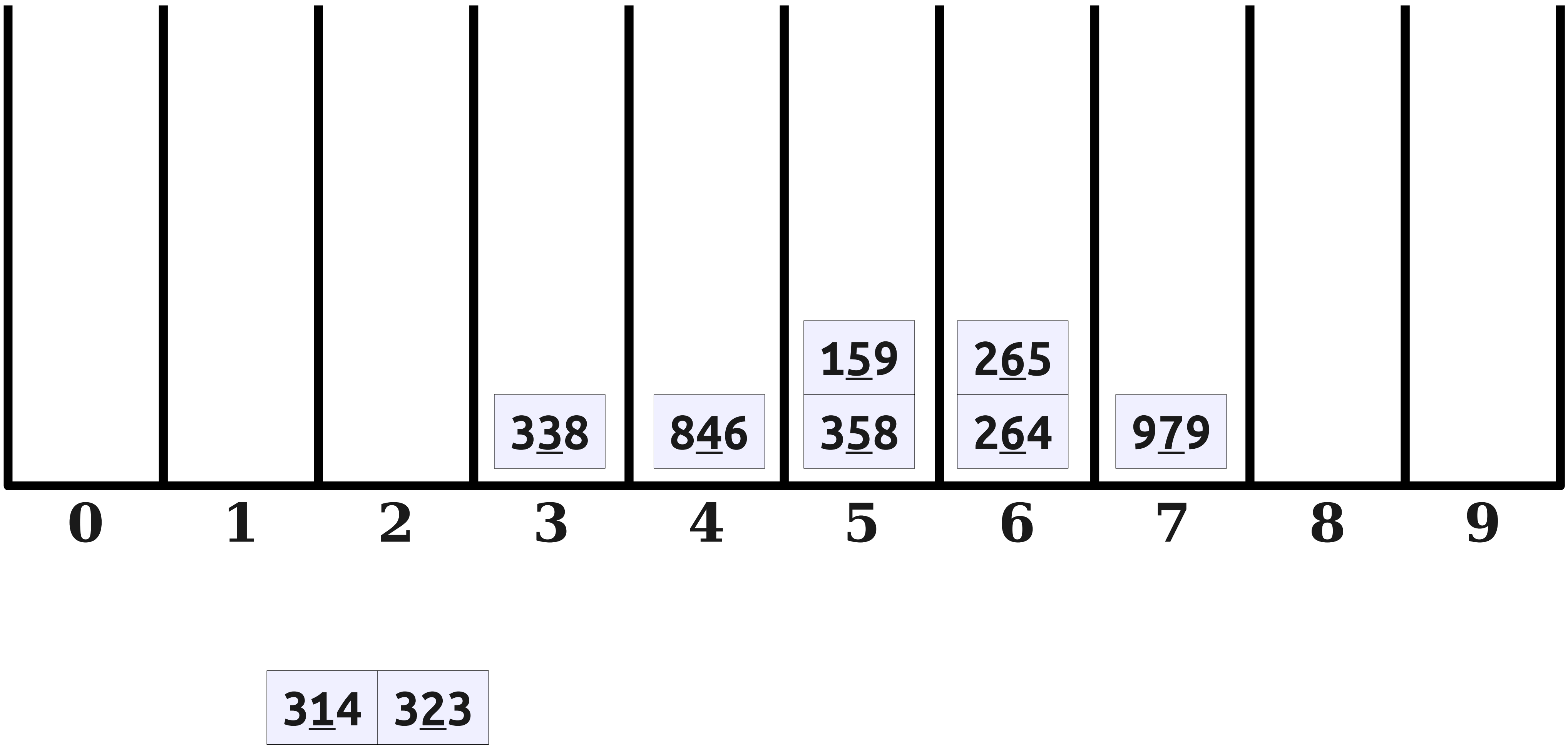
Radix Sort



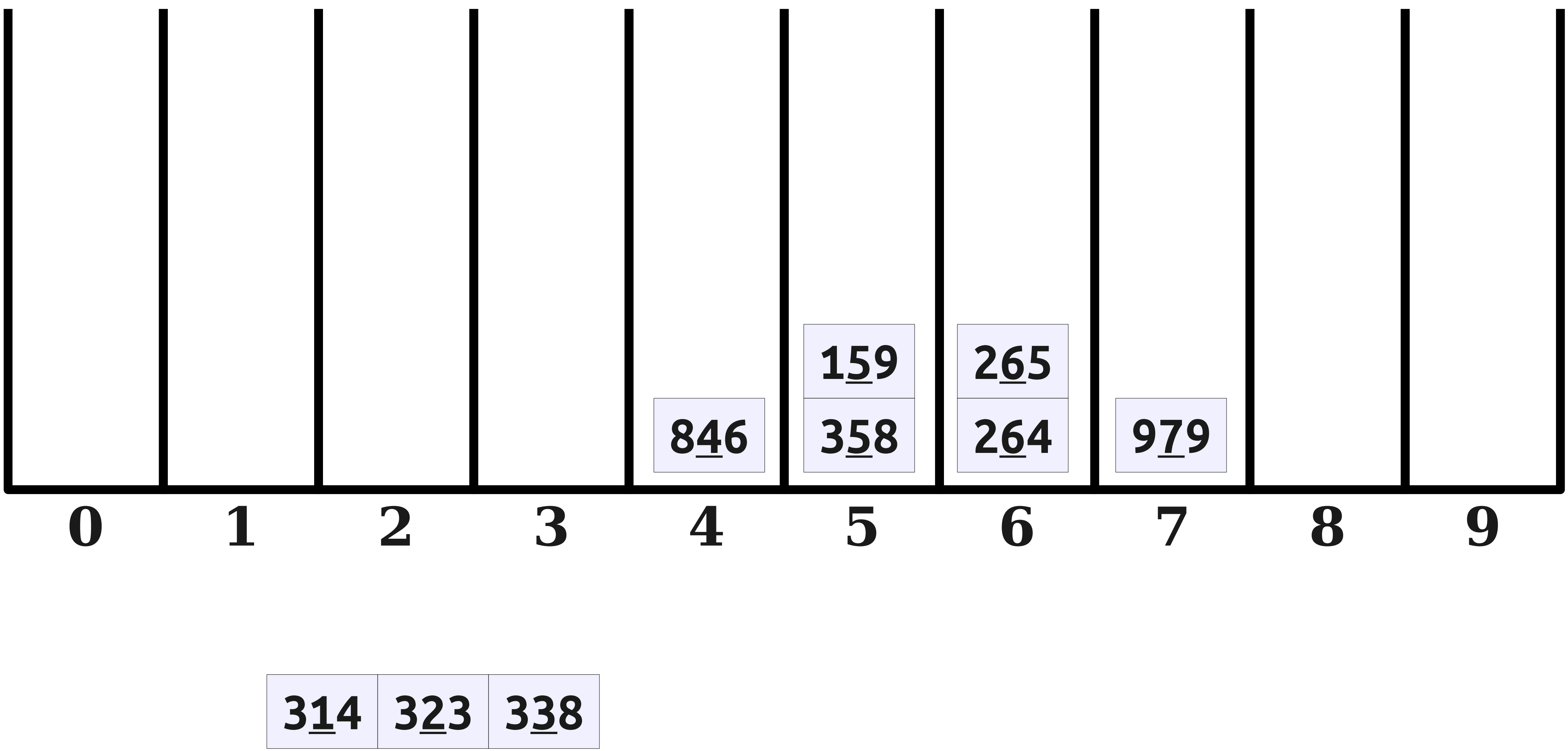
Radix Sort



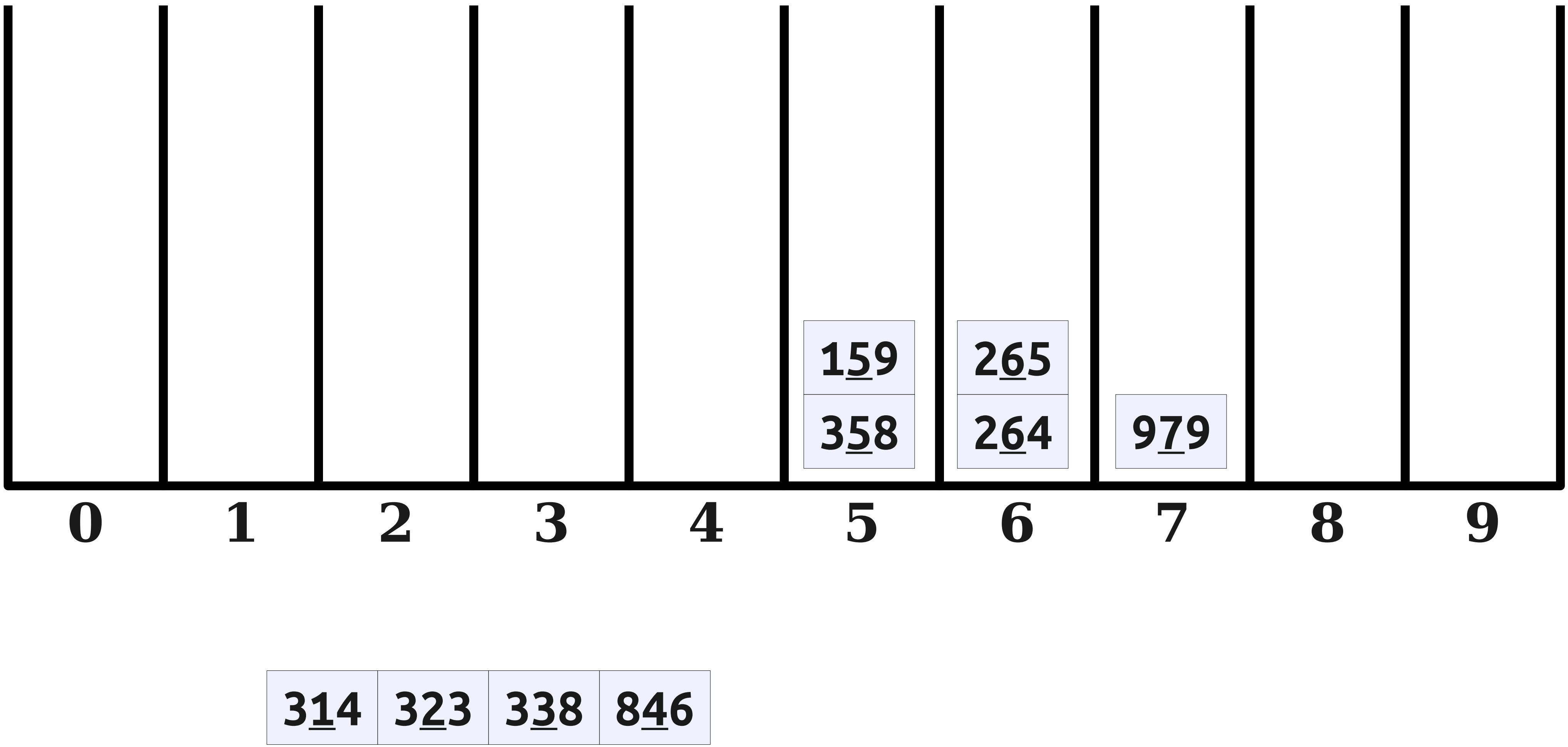
Radix Sort



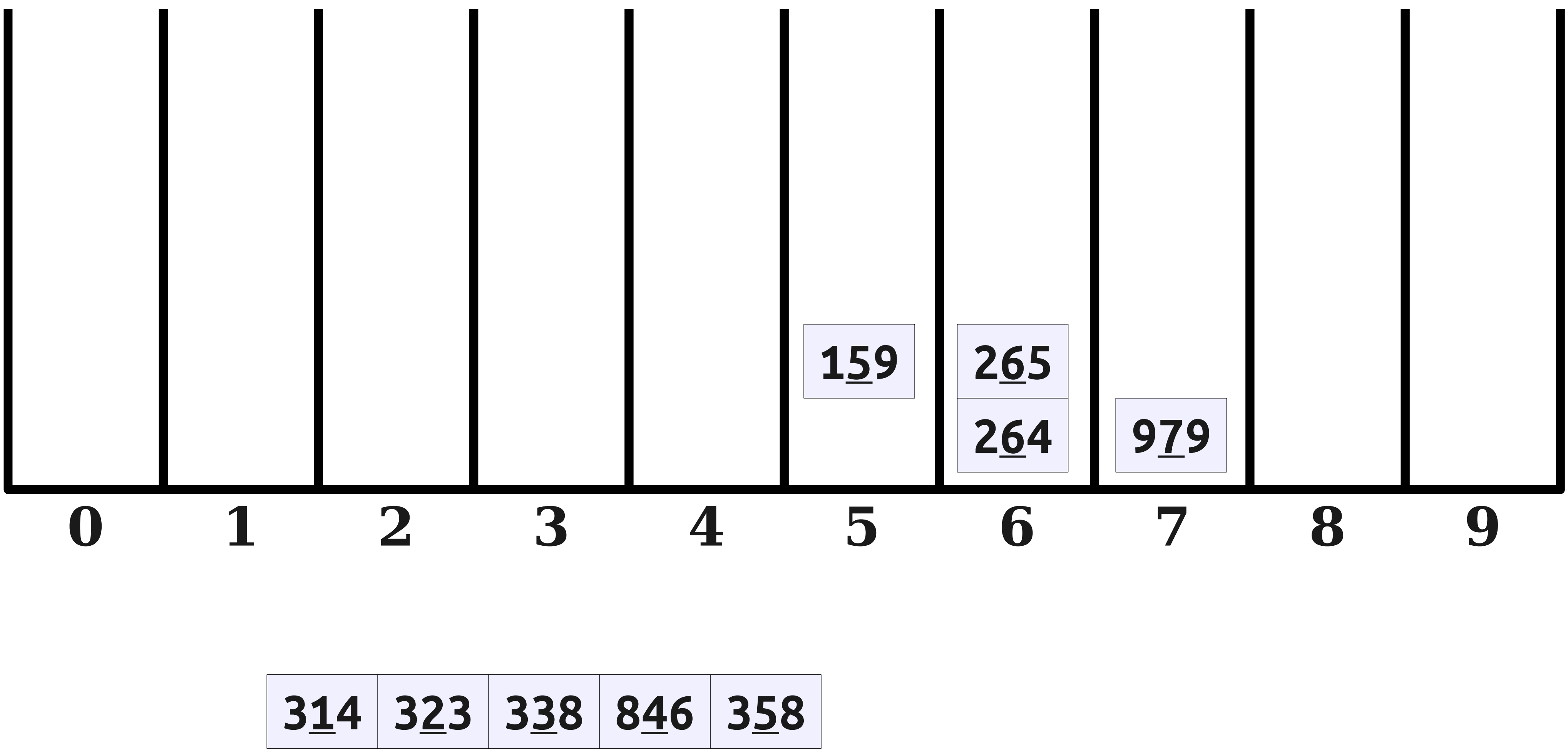
Radix Sort



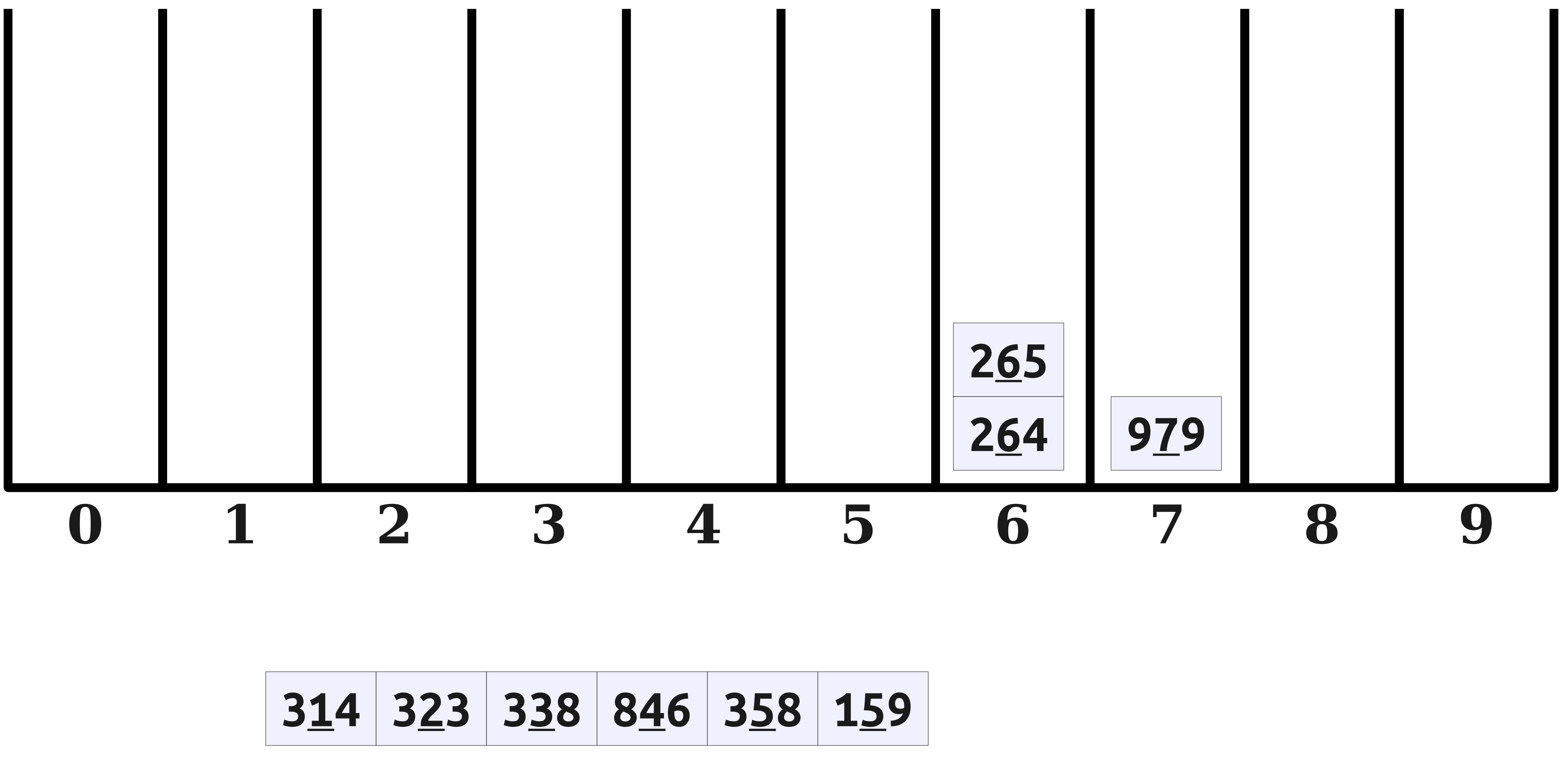
Radix Sort



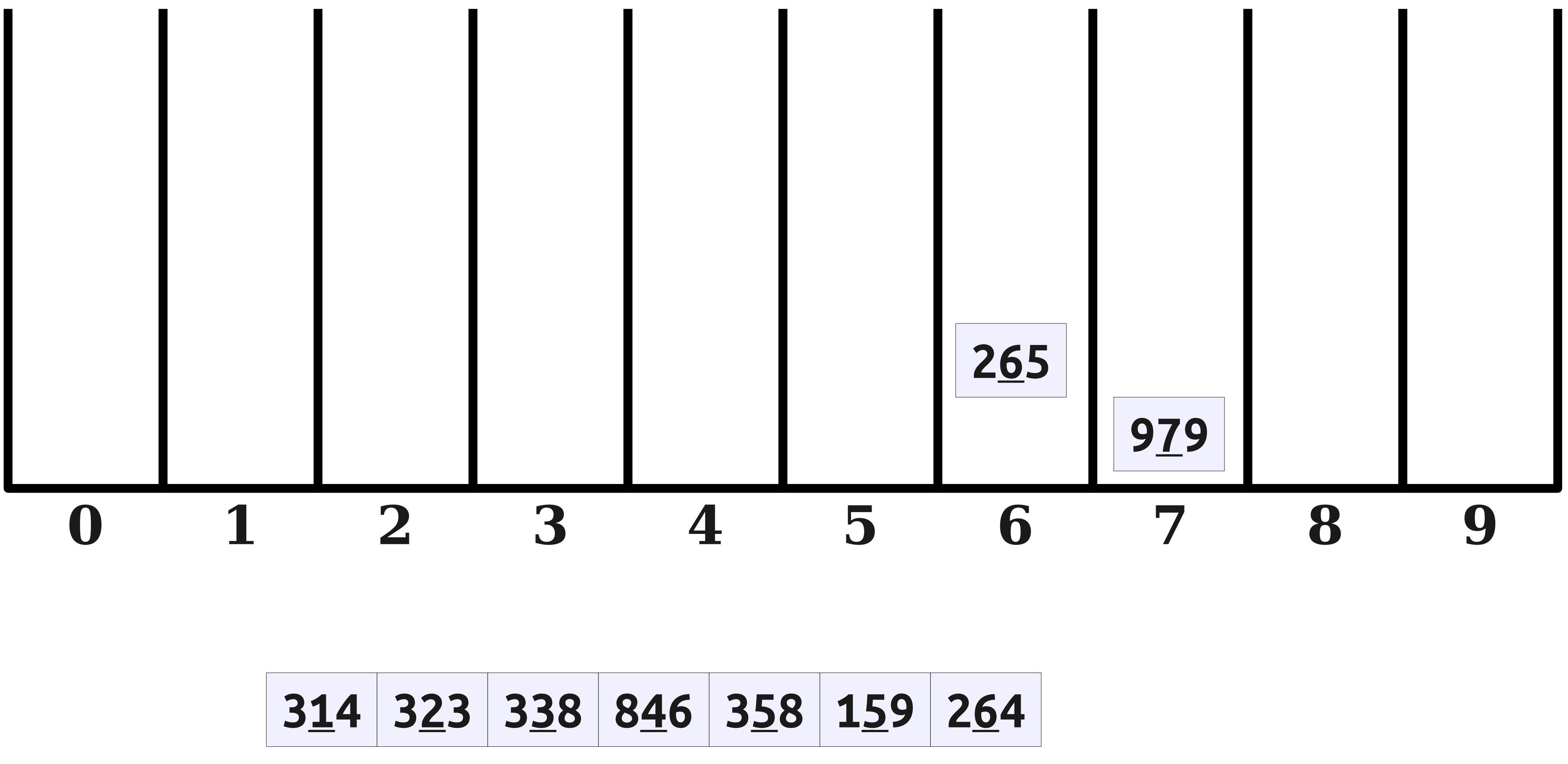
Radix Sort



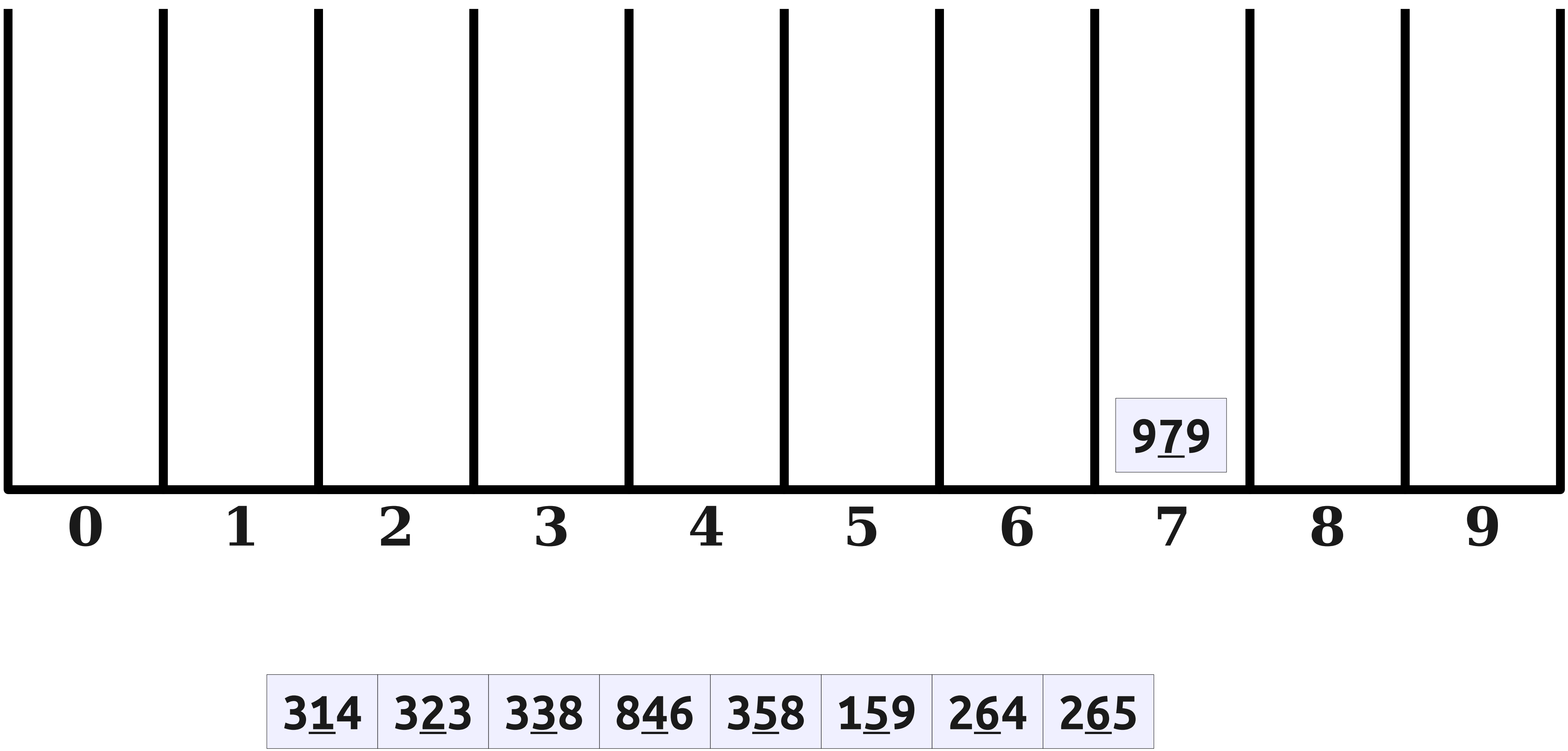
Radix Sort



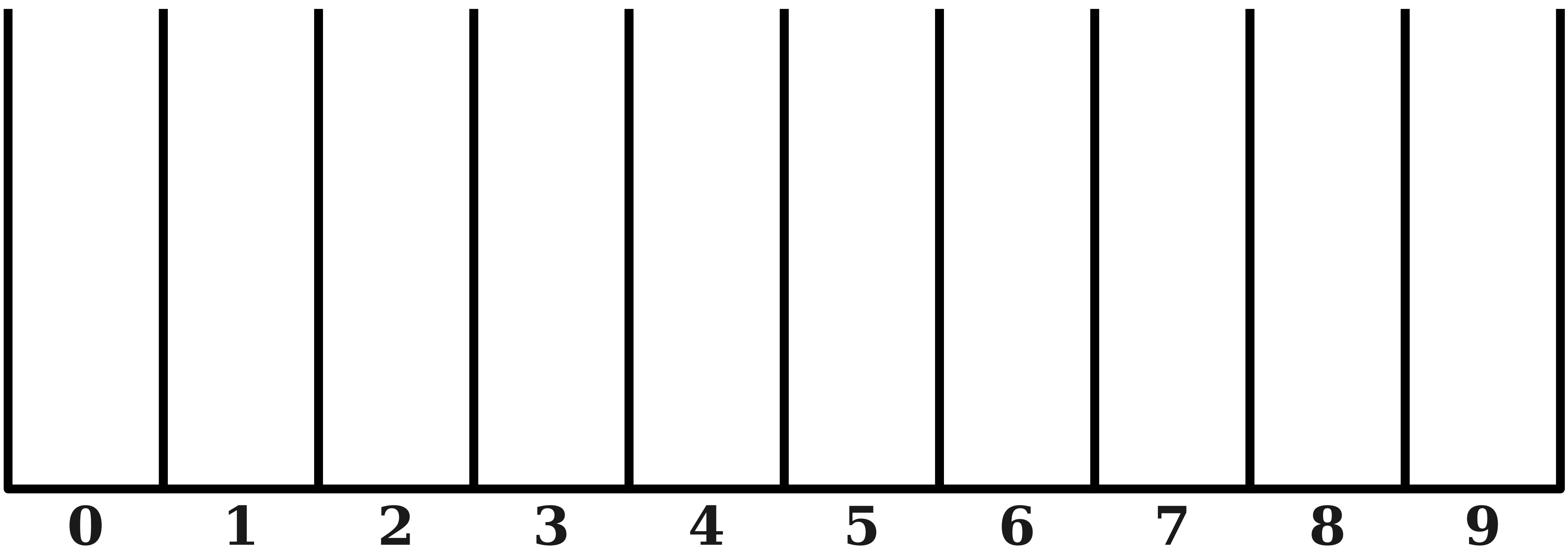
Radix Sort



Radix Sort

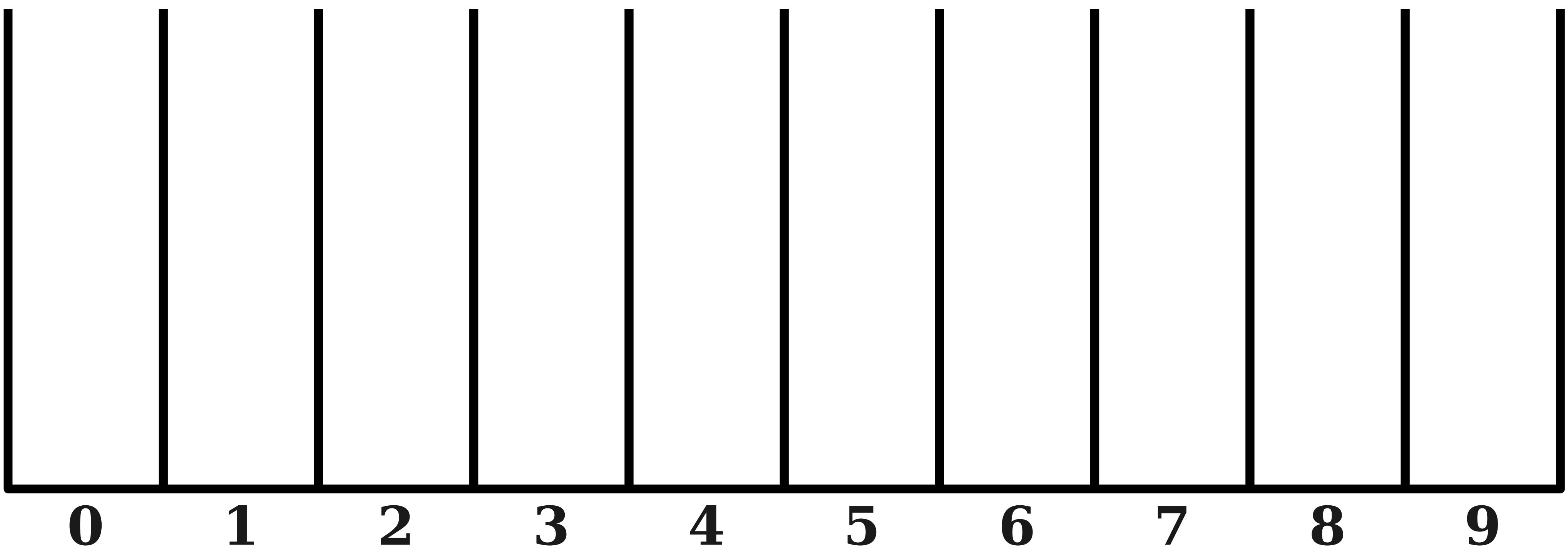


Radix Sort



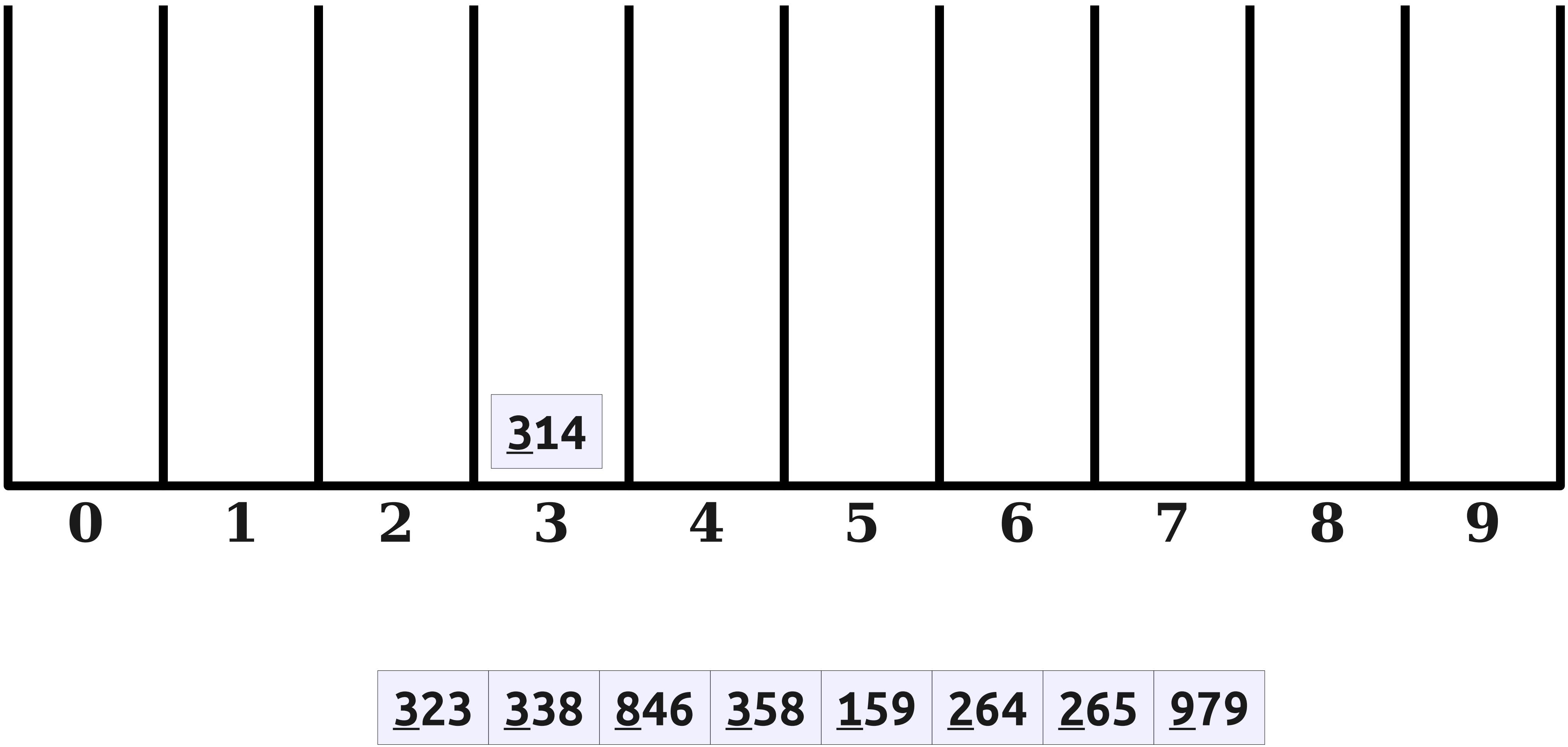
314	323	338	846	358	159	264	265	979

Radix Sort

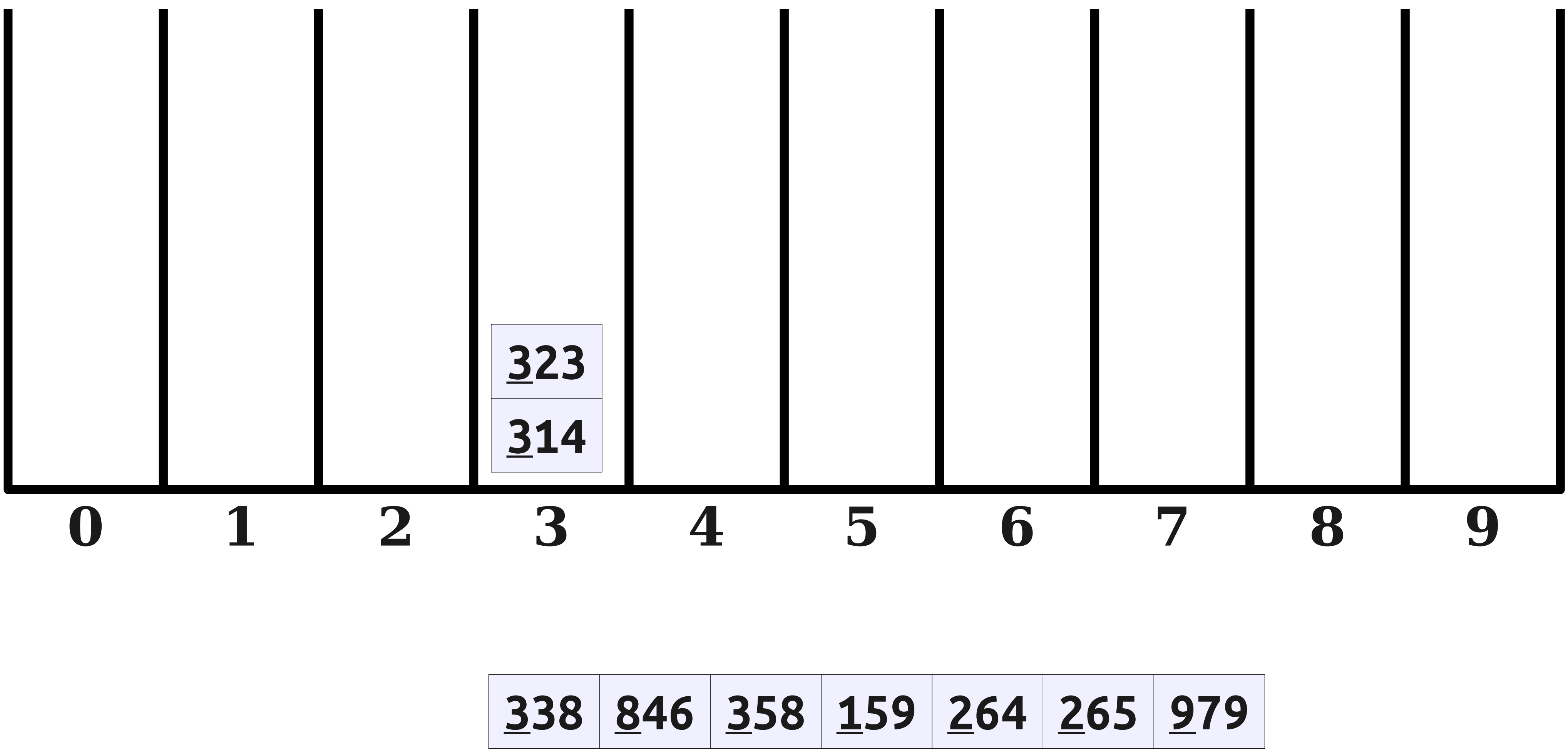


314	323	338	846	358	159	264	265	979

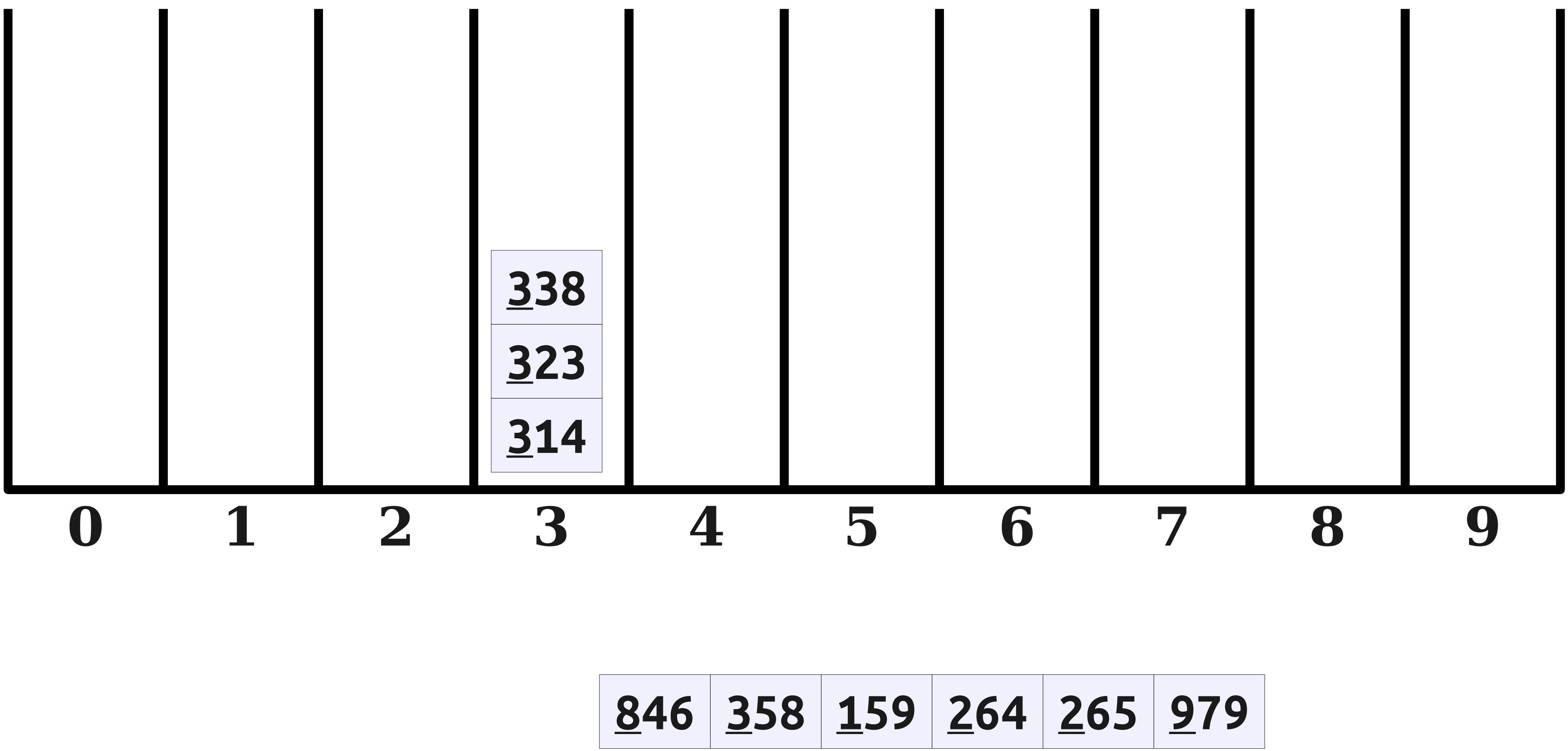
Radix Sort



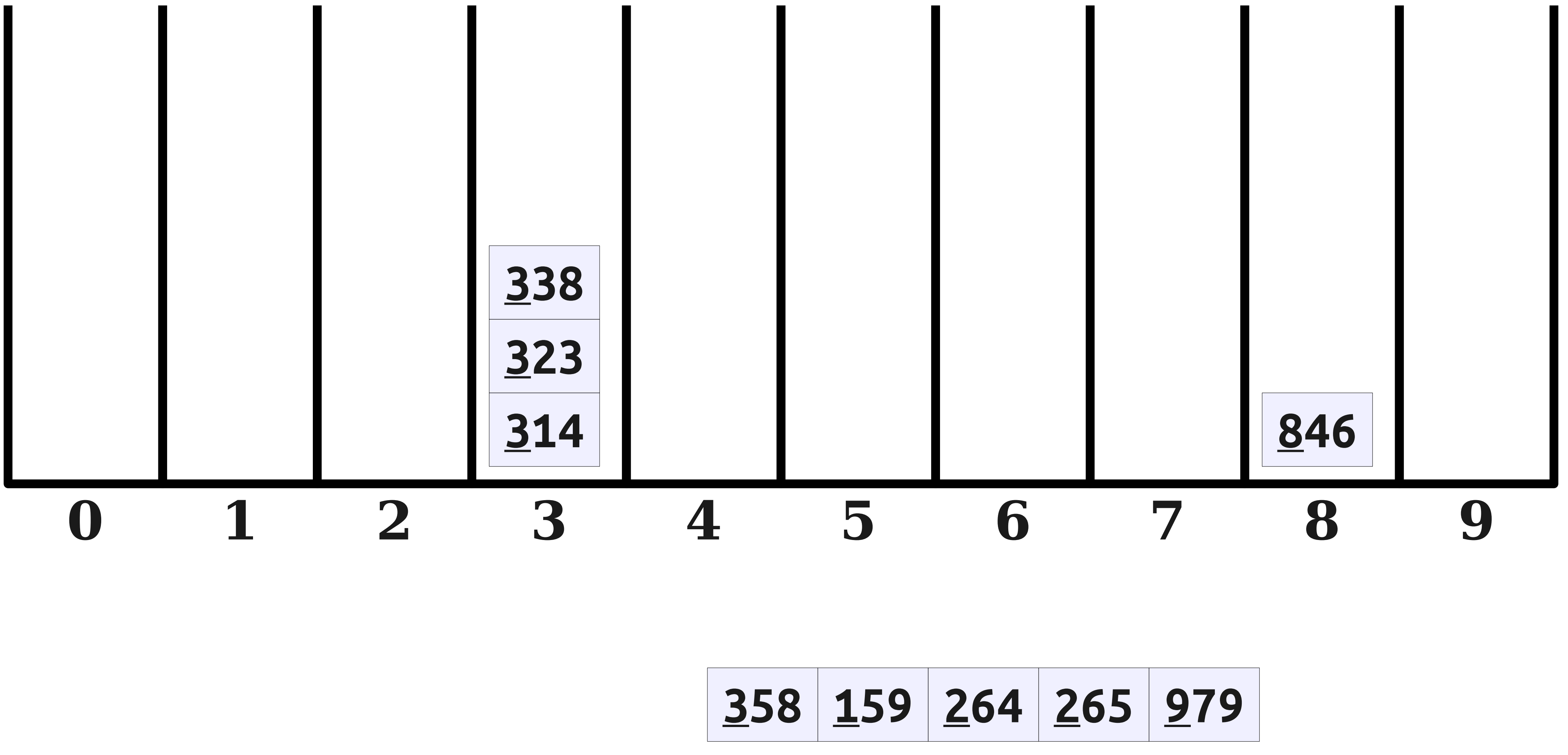
Radix Sort



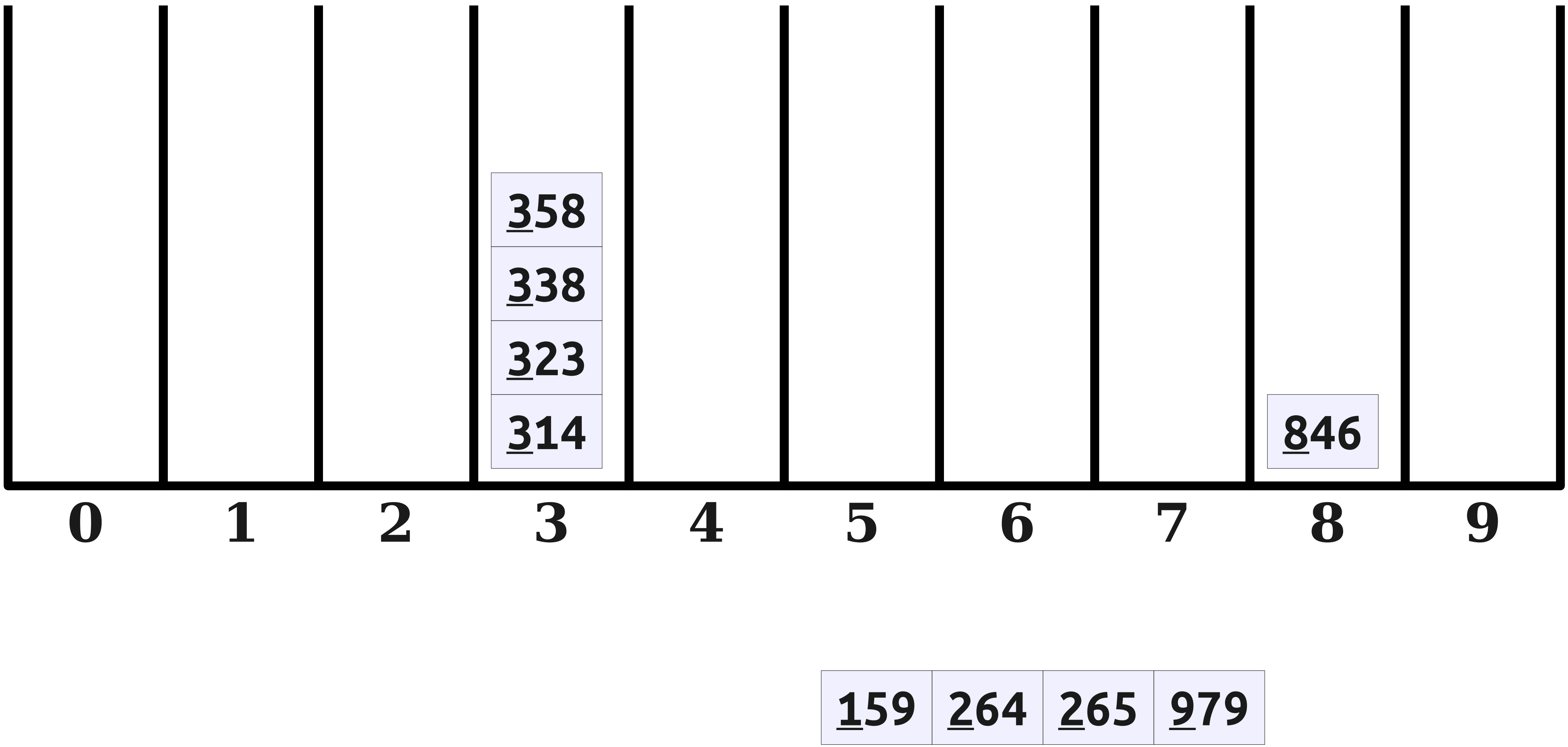
Radix Sort



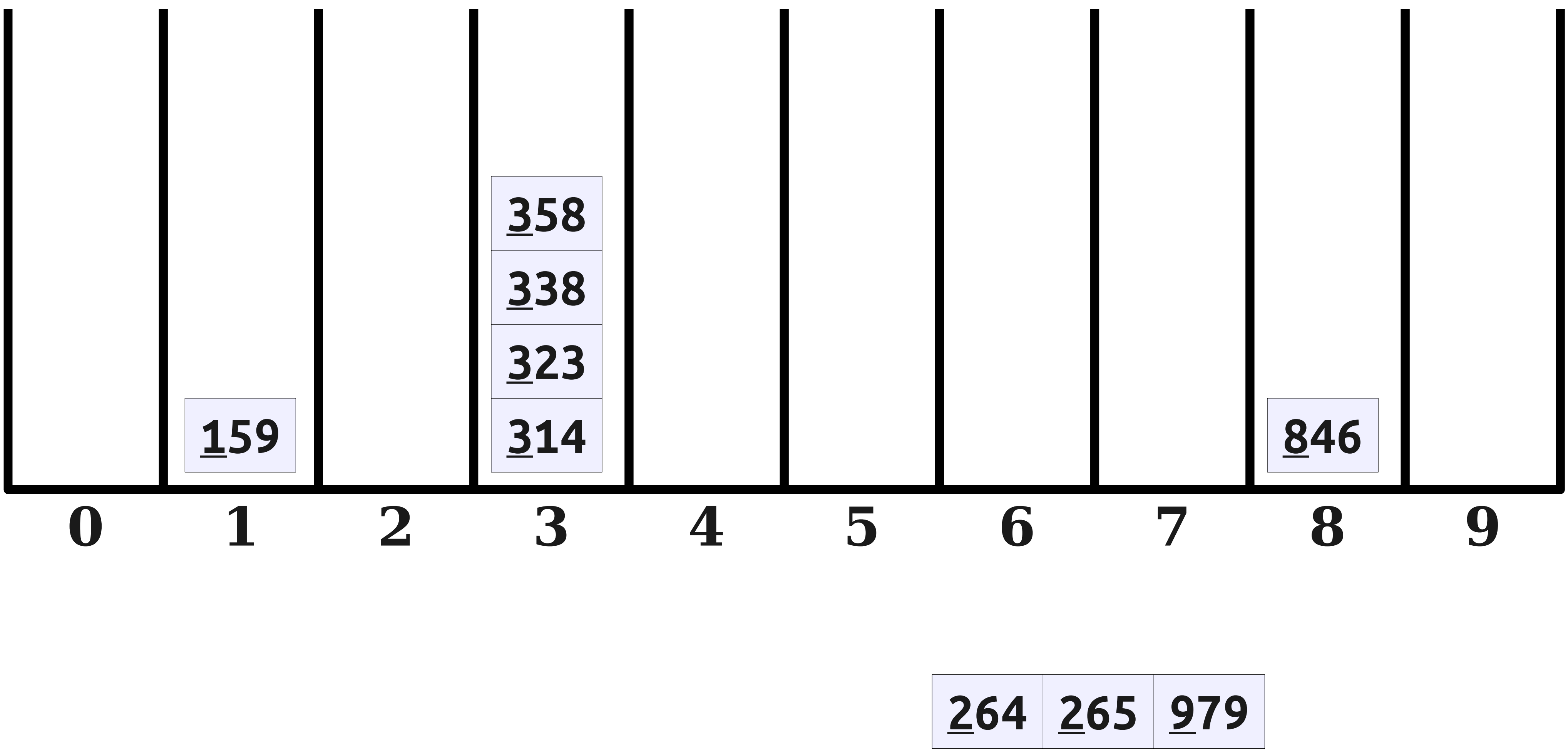
Radix Sort



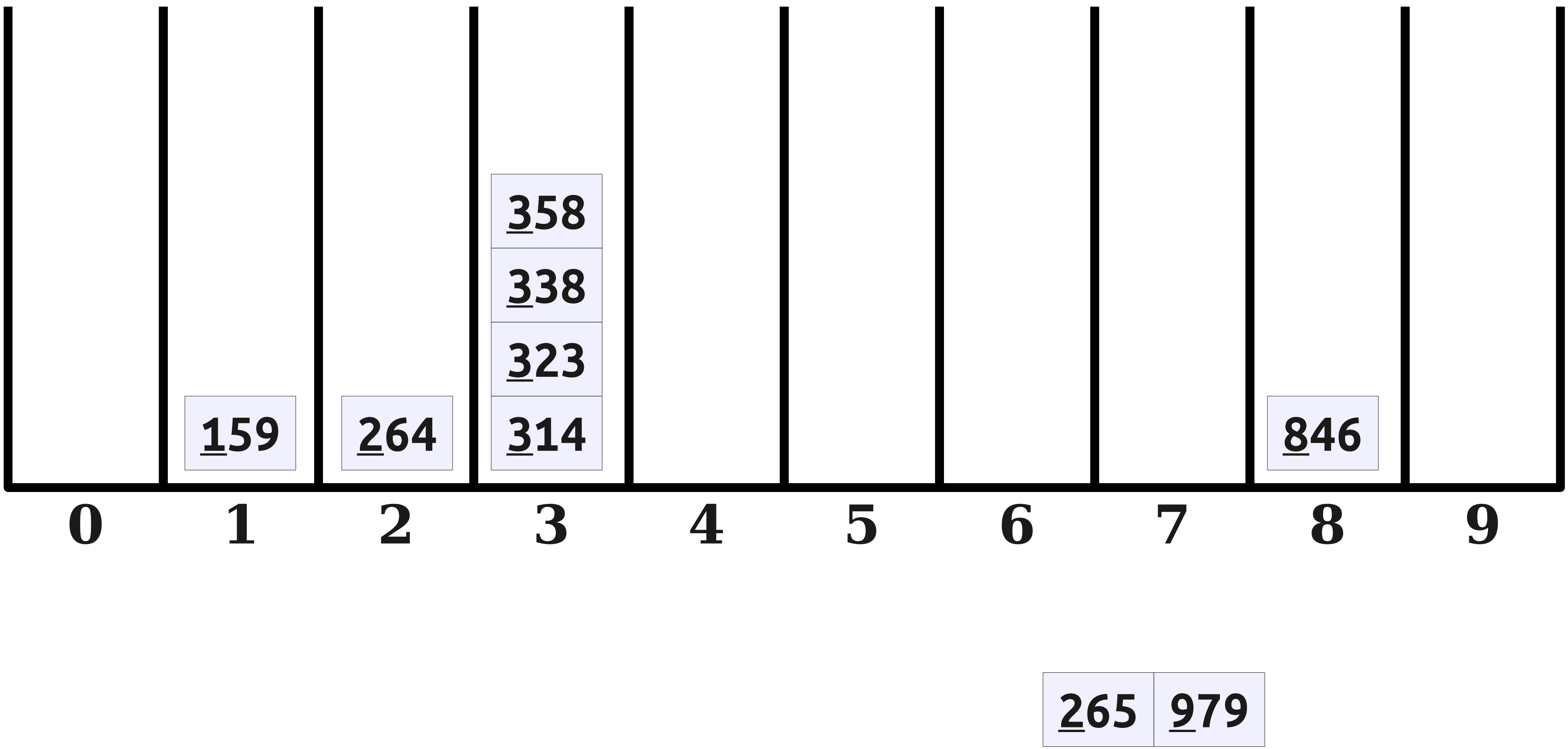
Radix Sort



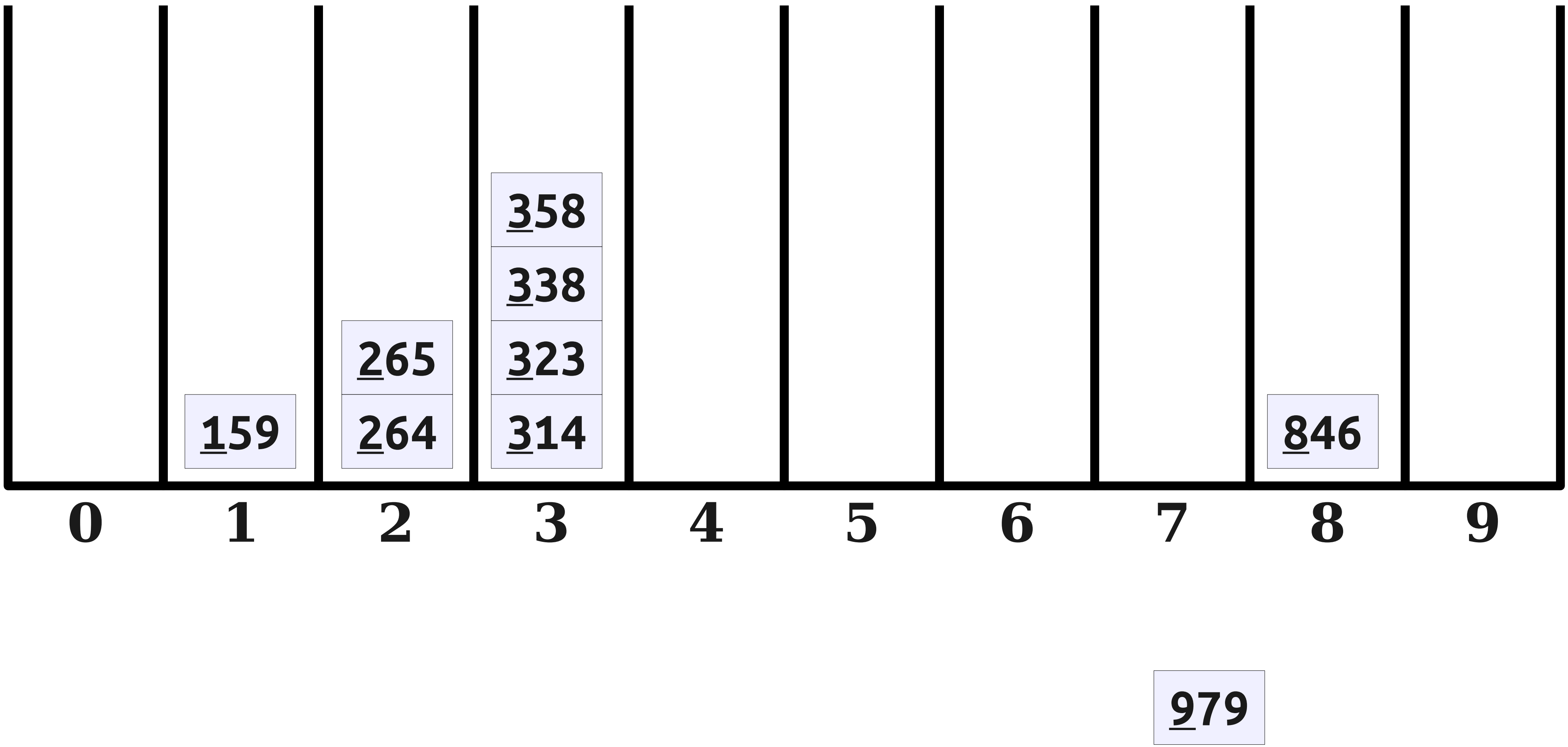
Radix Sort



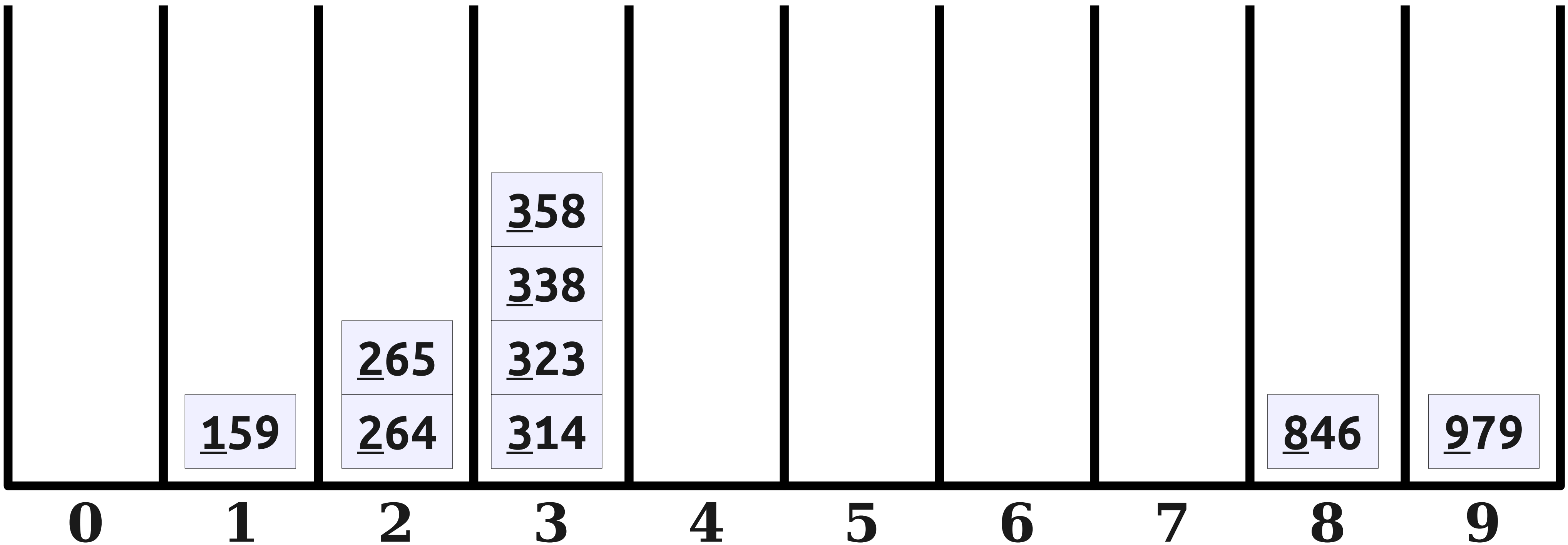
Radix Sort



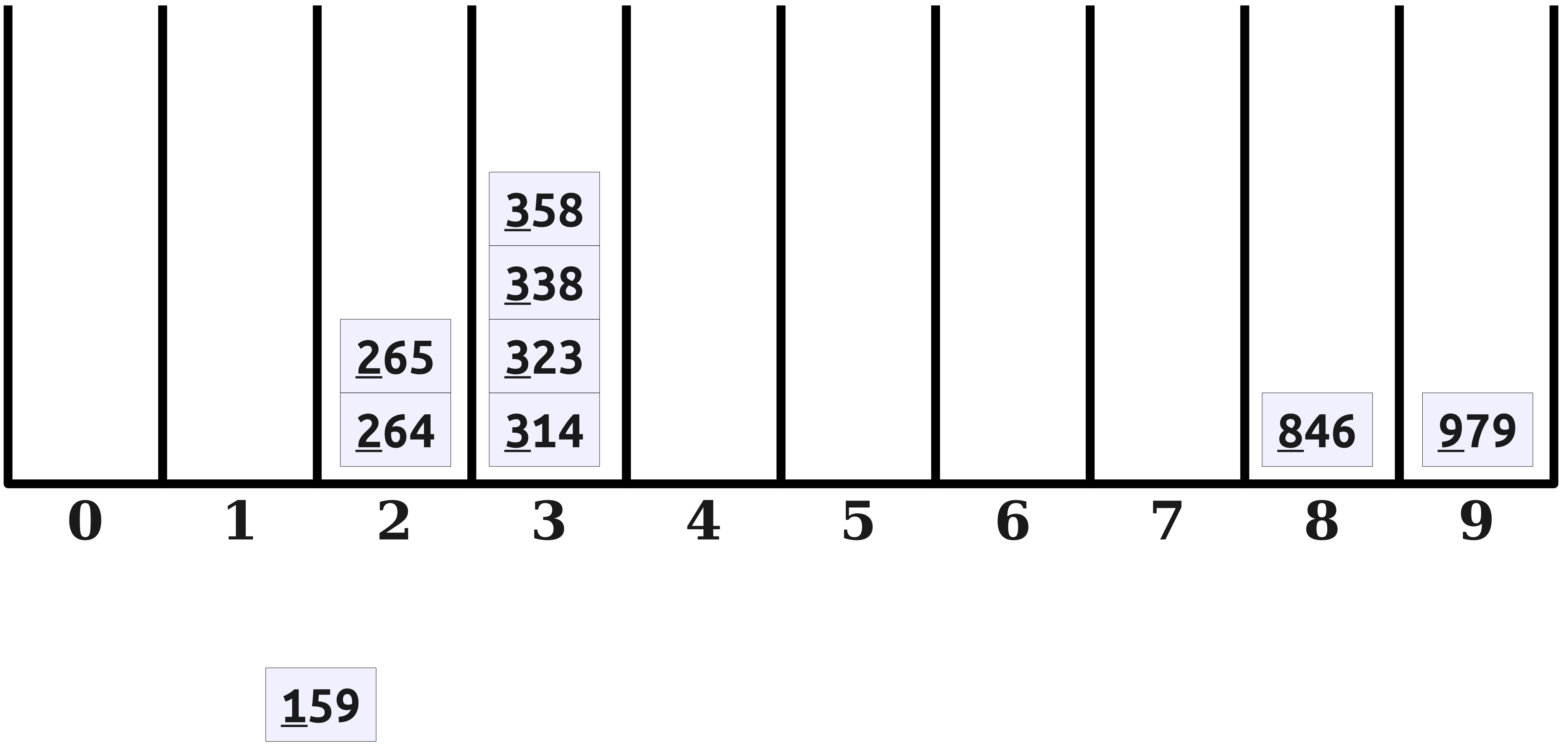
Radix Sort



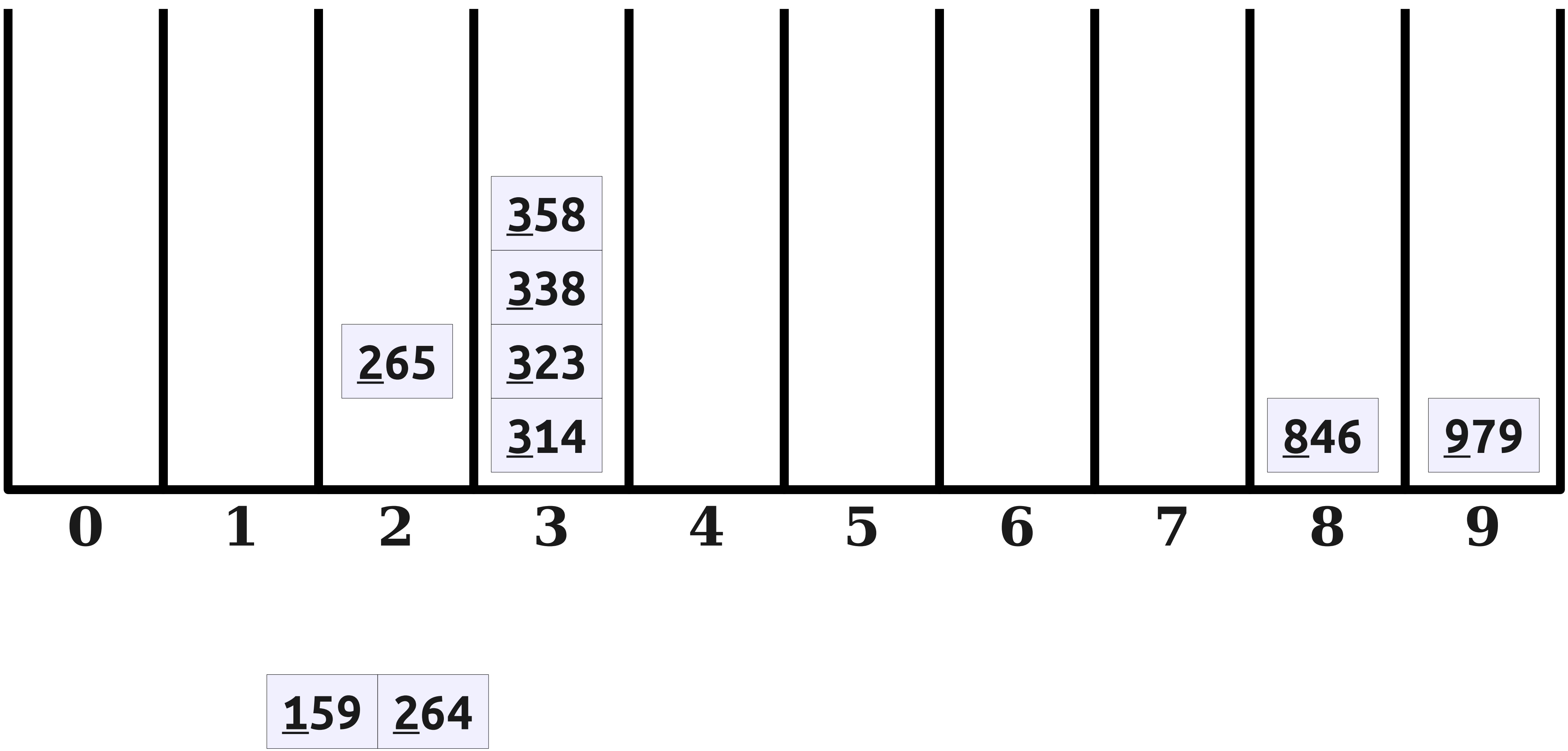
Radix Sort



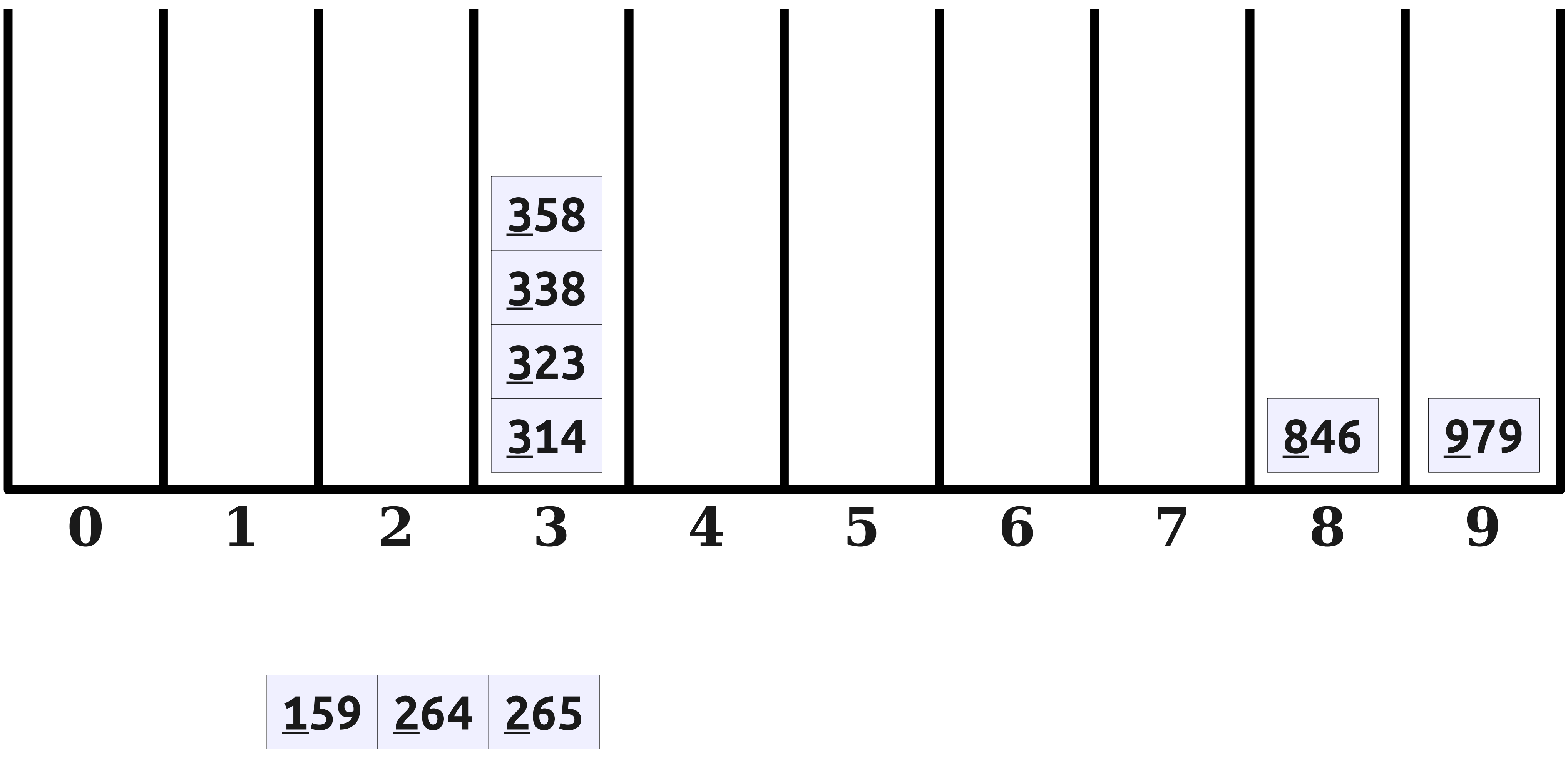
Radix Sort



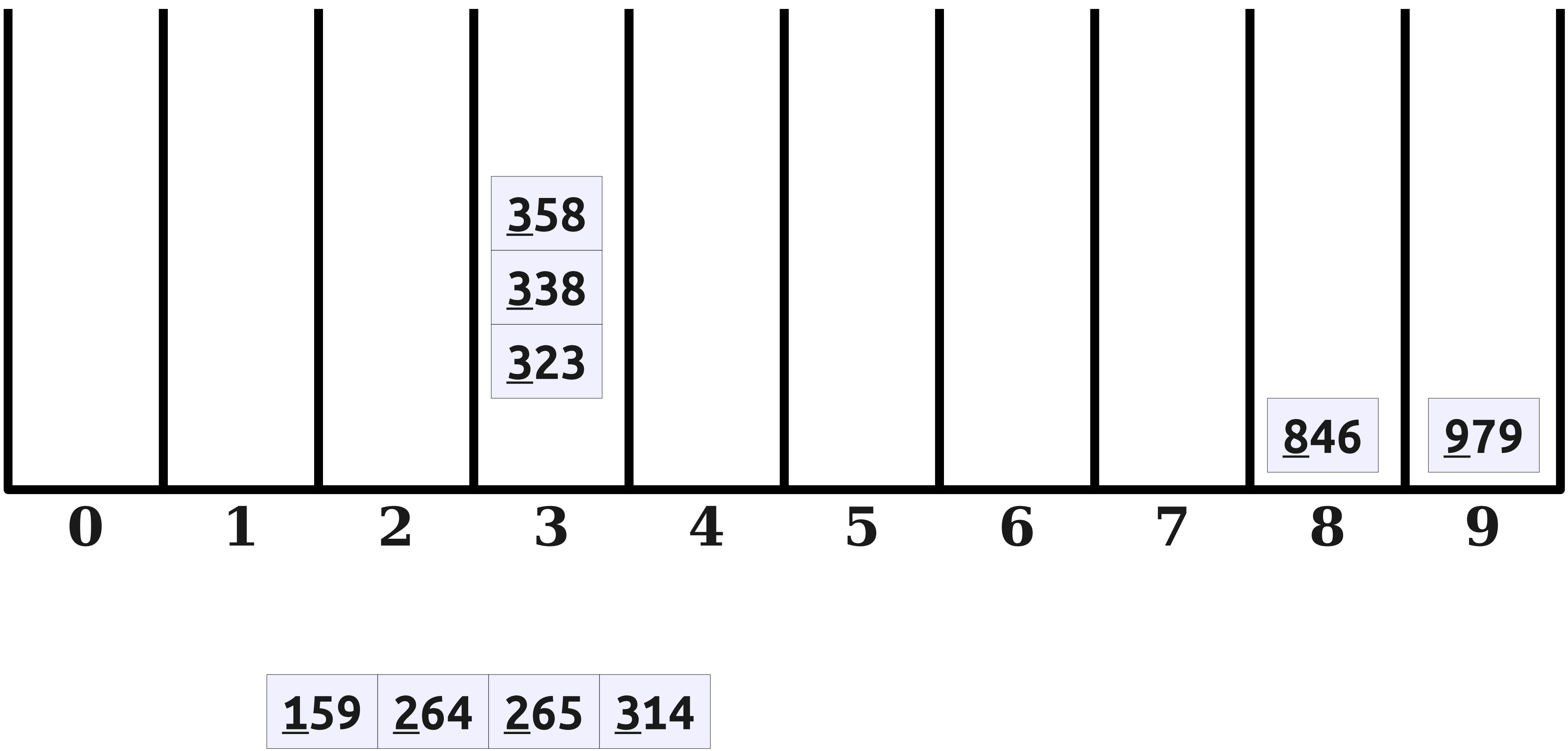
Radix Sort



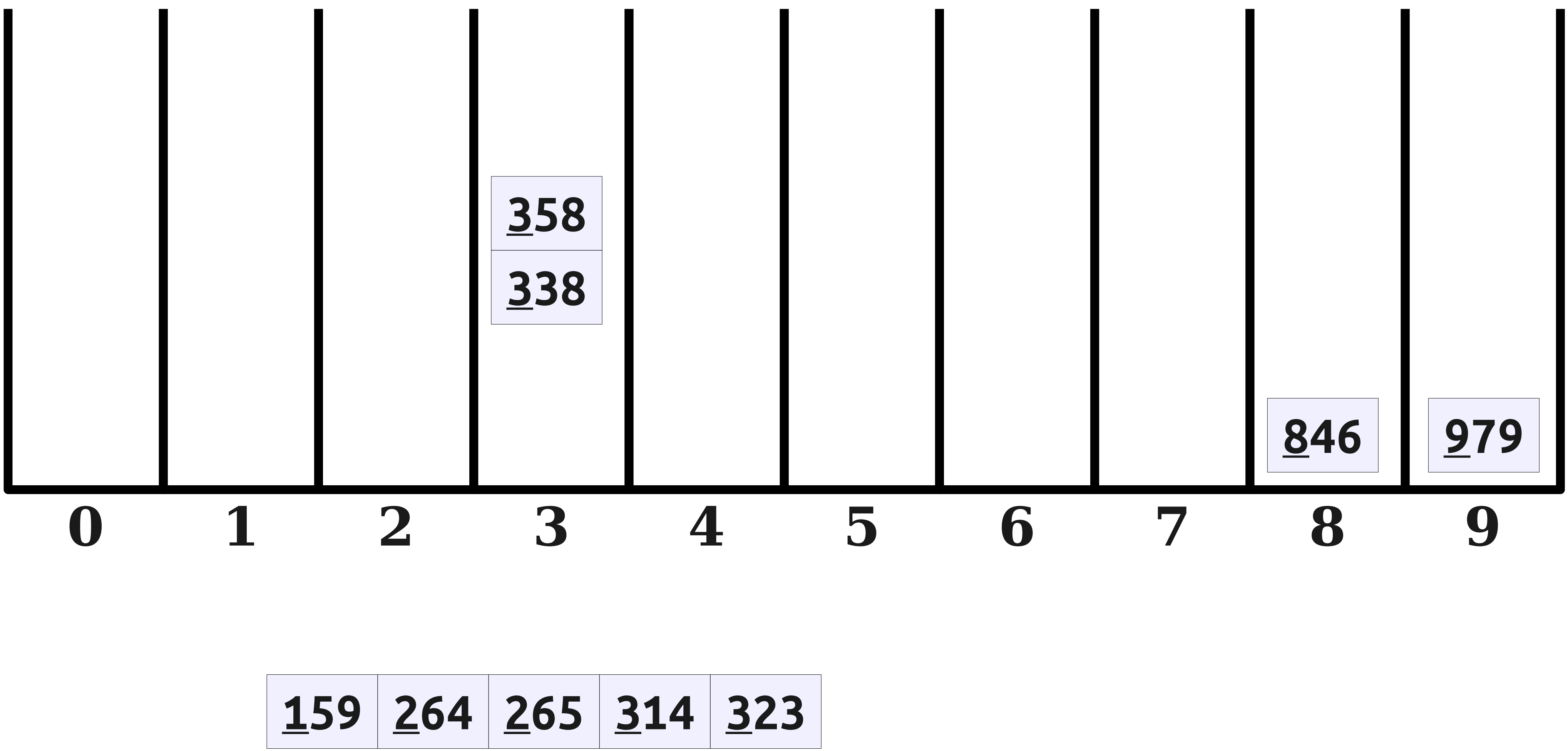
Radix Sort



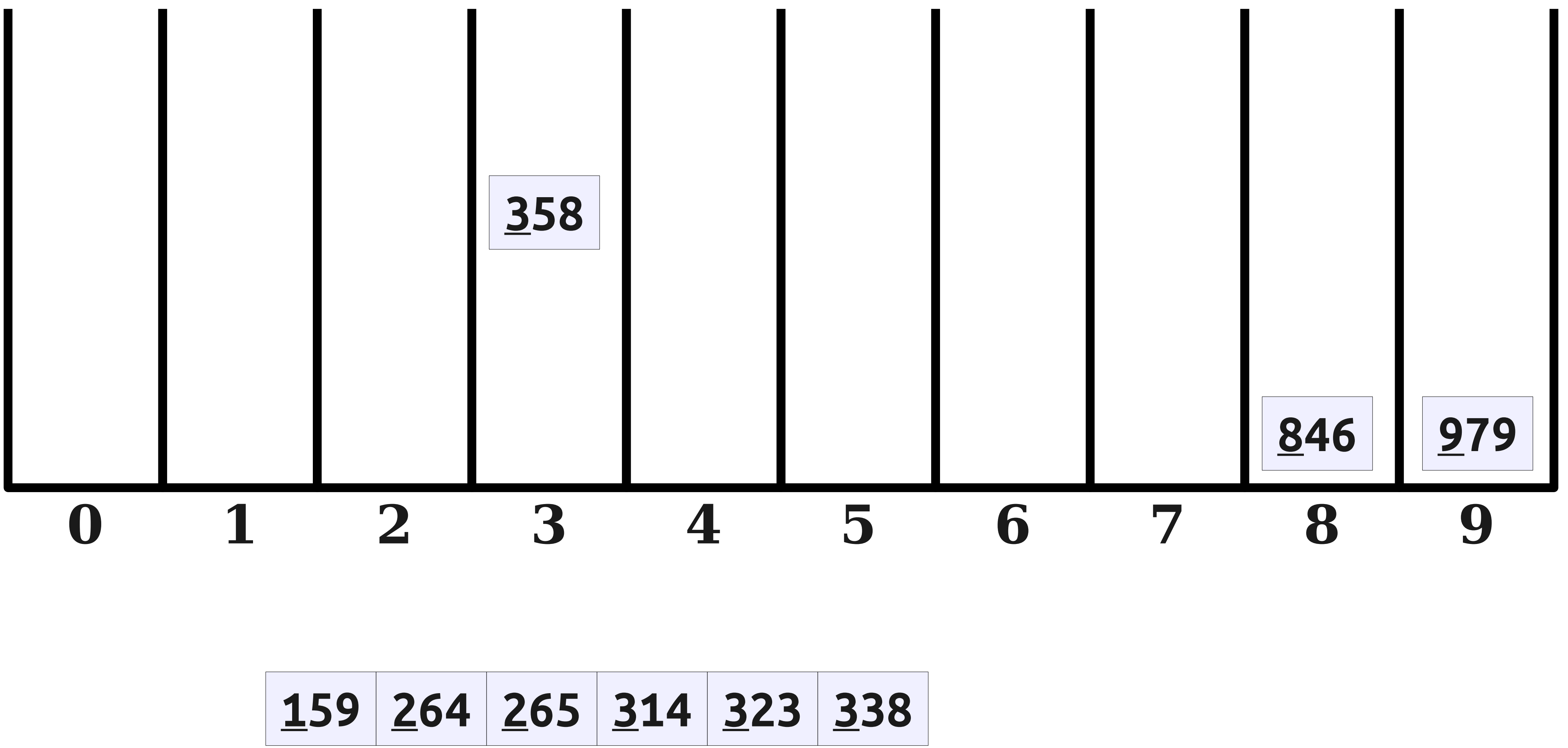
Radix Sort



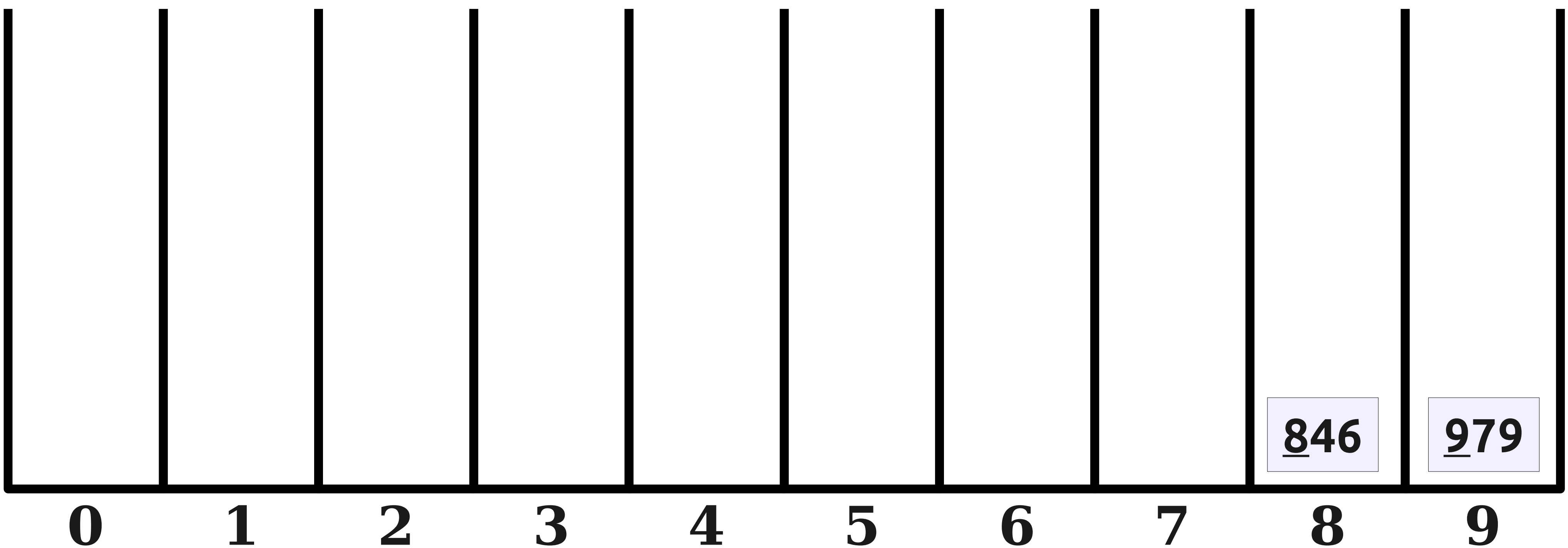
Radix Sort



Radix Sort

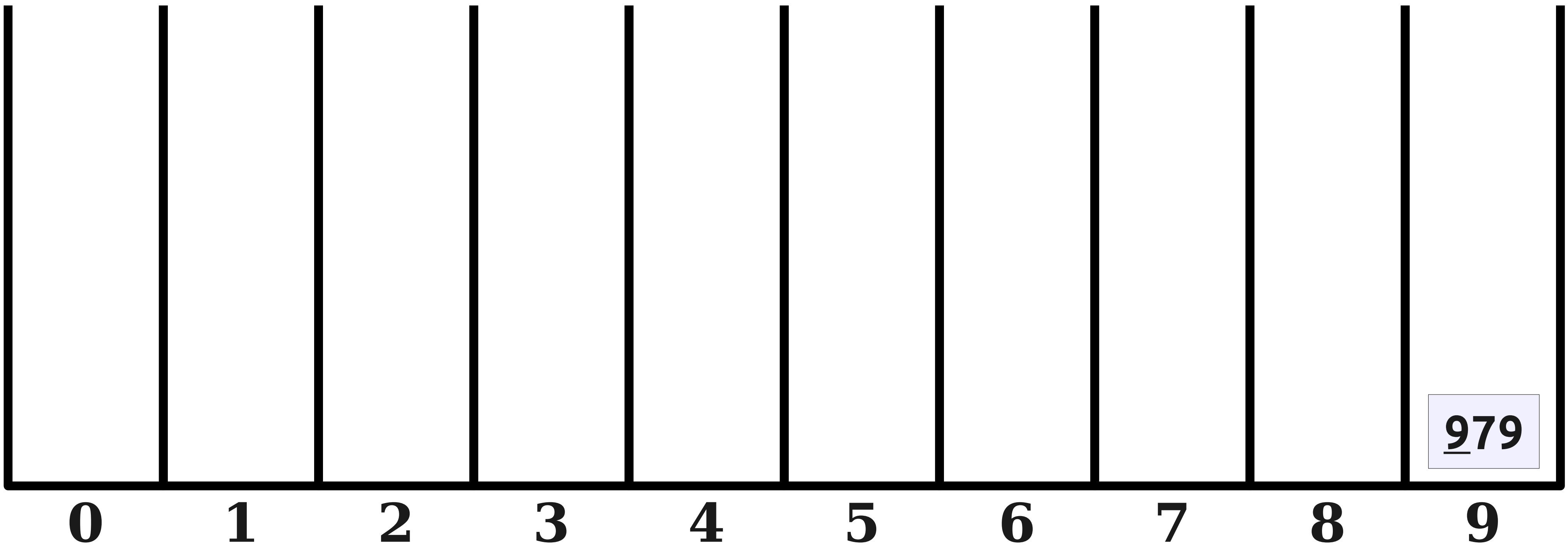


Radix Sort



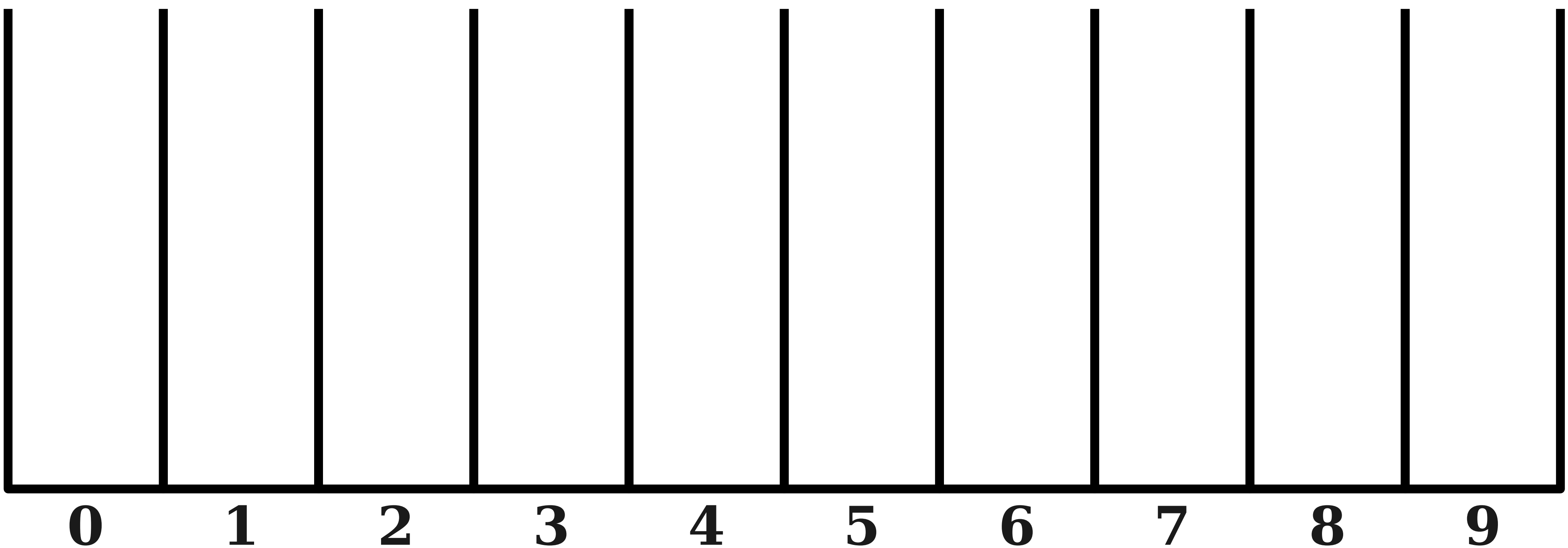
159	264	265	314	323	338	358

Radix Sort



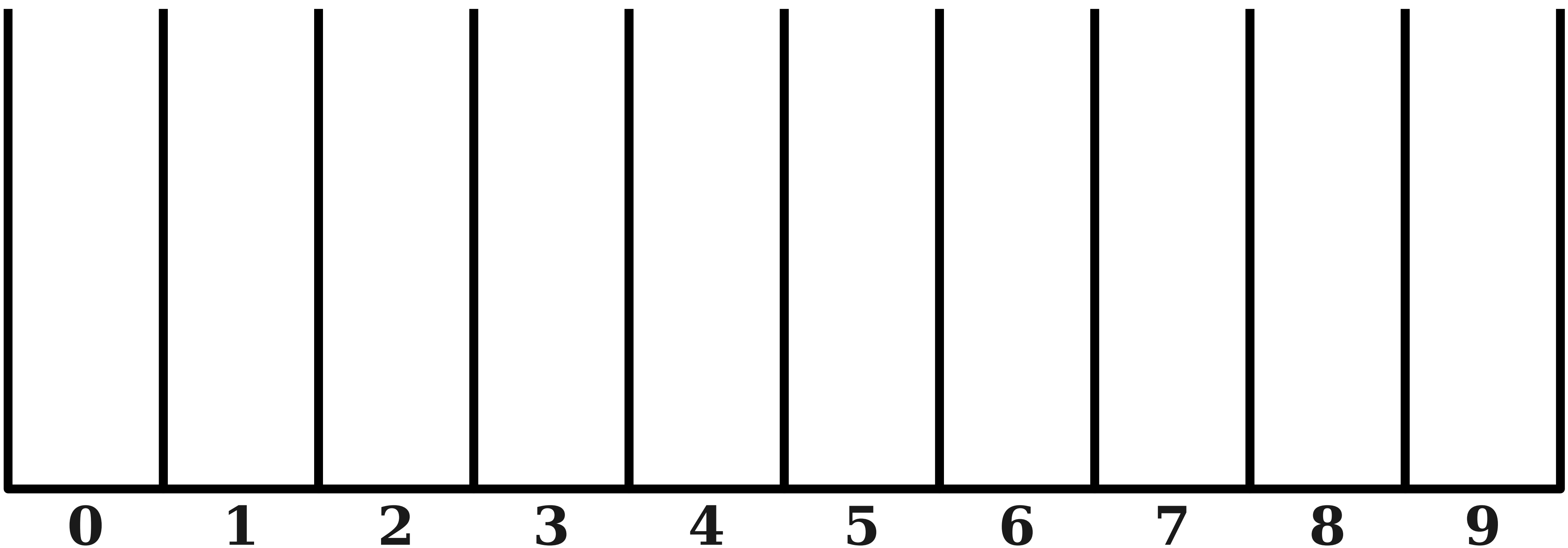
159	264	265	314	323	338	358	846

Radix Sort



159	264	265	314	323	338	358	846	979

Radix Sort



159	264	265	314	323	338	358	846	979

Radix Sort Runtime

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

Speeding Up with Radix Sort

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

The DC3 Algorithm

DC3, Intuitively

- At a high-level, DC3 works as follows:
  - Recursively get the sorted order of all suffixes starting at positions that aren't multiples of three.
  - Using this information, sort the suffixes at positions that *are* at multiples of three.
  - Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall 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 starting at positions that aren't multiples of three.
- Using this information, sort the suffixes at positions that *are* at multiples of three.
- Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall suffix array.

The details are beautiful, but tricky.

Some Terminology

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

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

Some Terminology

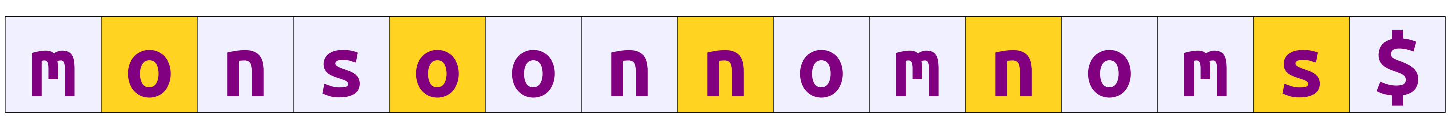
- Define T_k to be the positions in T whose indices are equal to k mod 3.
  - T_0 is the set of positions that are multiples of three.
  - T_1 is the set of positions that follow the positions in T_0.
  - T_2 is the set of positions that follow the positions in T_1.

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

T_0

Some Terminology

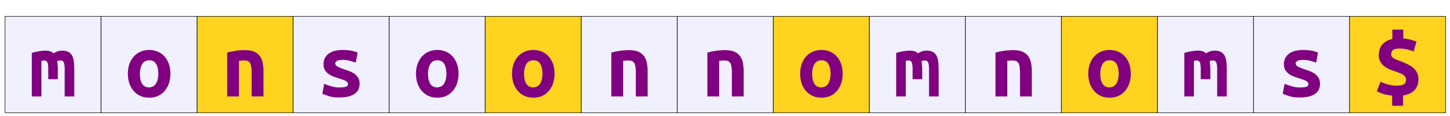
- Define T_k to be the positions in T whose indices are equal to k mod 3.
  - T_0 is the set of positions that are multiples of three.
  - T_1 is the set of positions that follow the positions in T_0.
  - T_2 is the set of positions that follow the positions in T_1.



T_1

Some Terminology

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



T_2

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

A Beautiful Insight

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

m	7

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

A Beautiful Insight

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

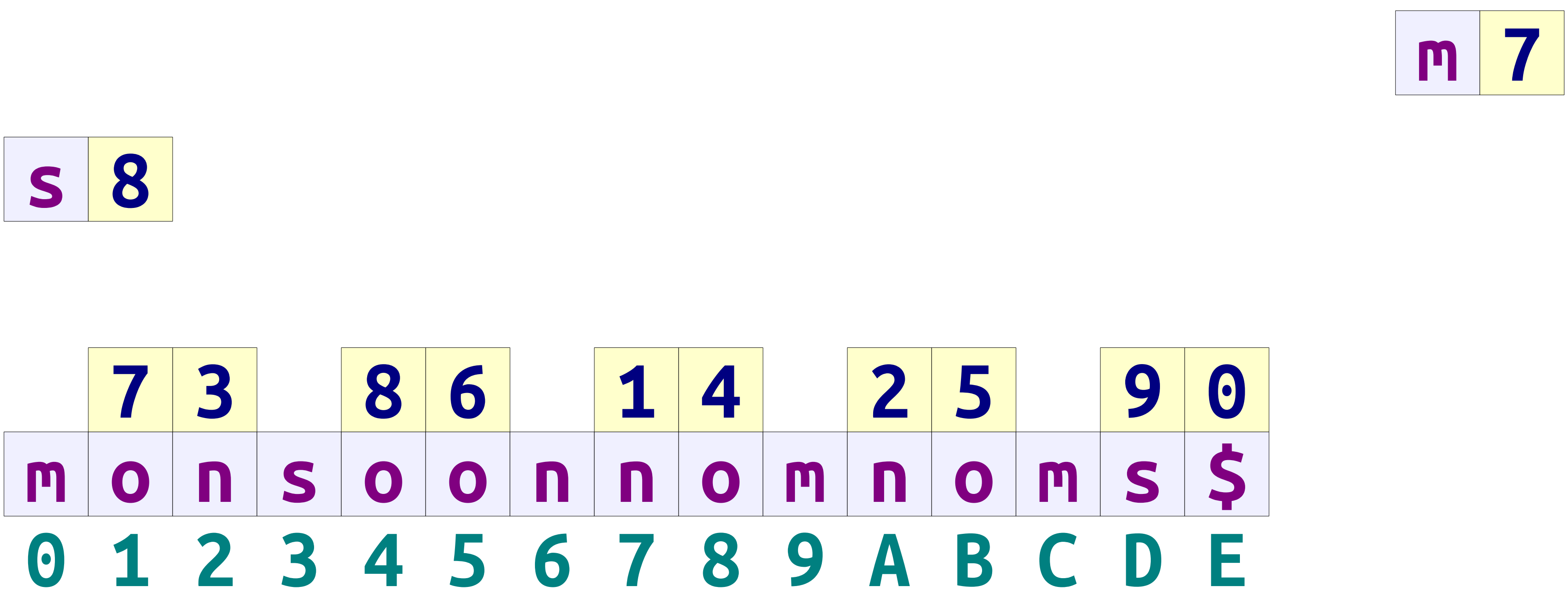
m	7

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

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

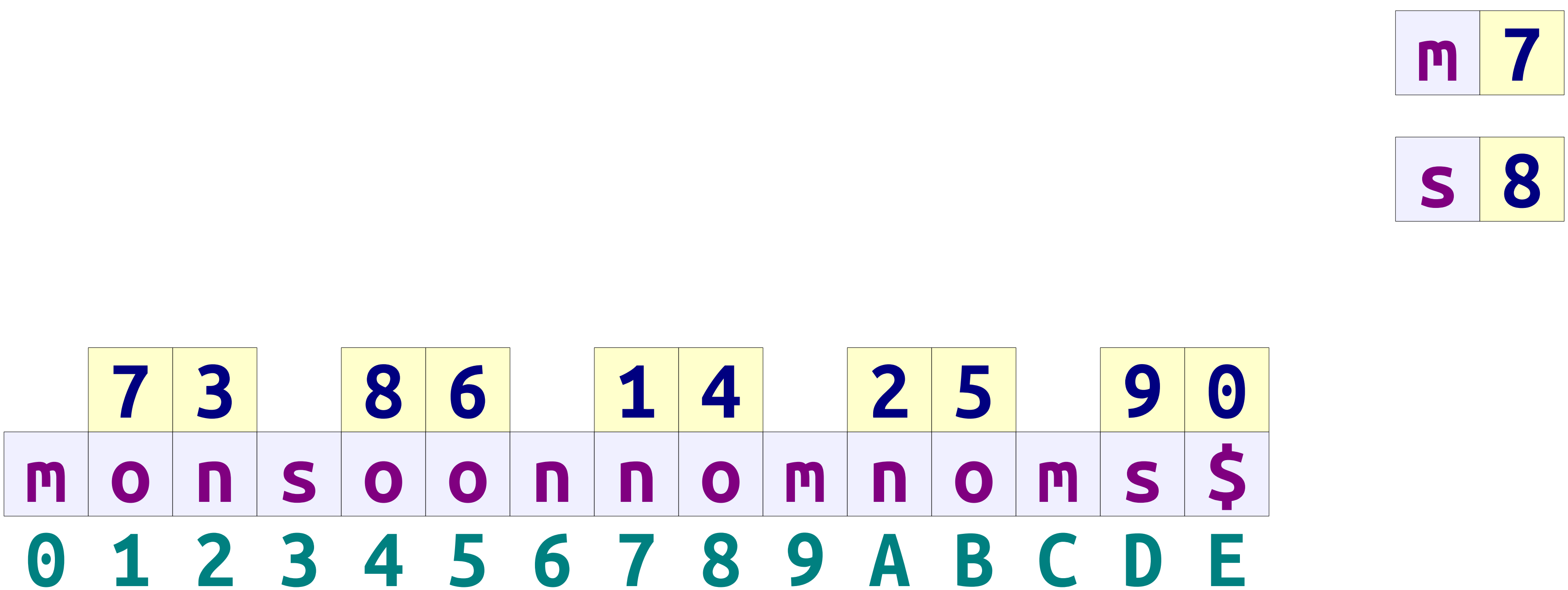
A Beautiful Insight

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



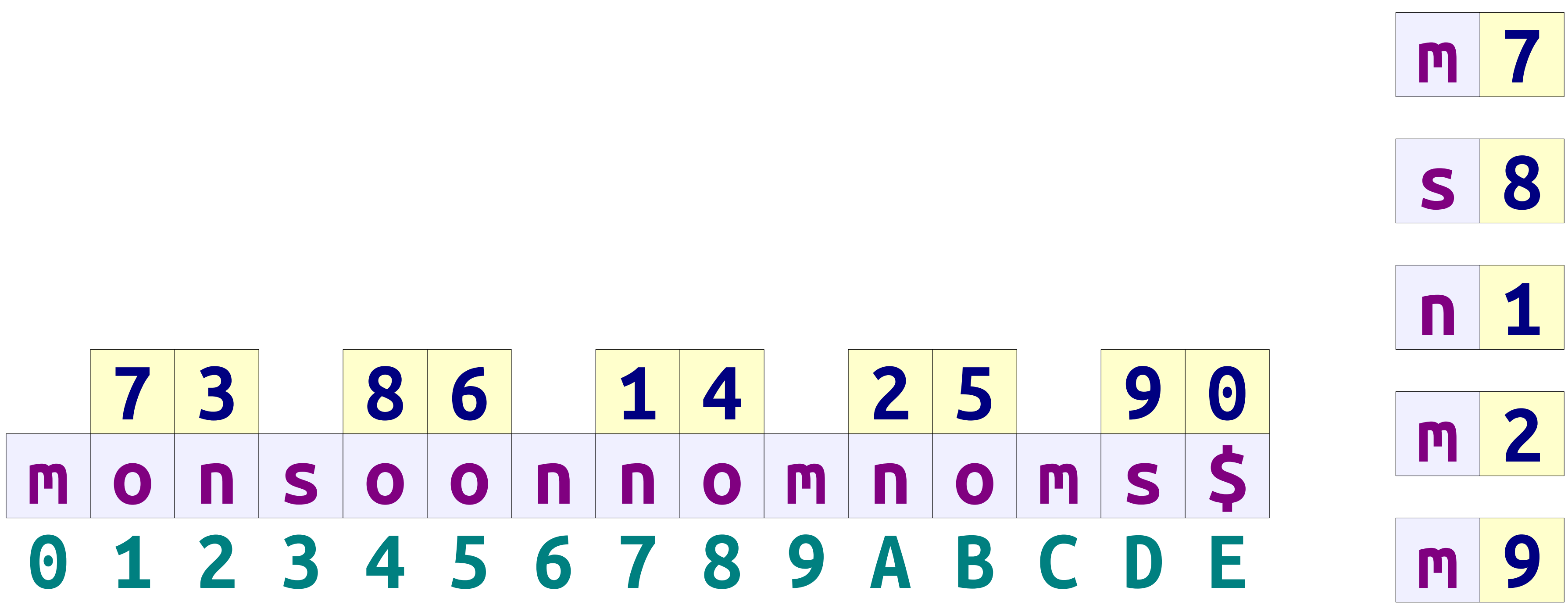
A Beautiful Insight

- **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!

		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

n	1
m	2
m	7
s	8
m	9

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

m	2
m	7
m	9
n	1
s	8

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	2
m	7
m	9
n	1
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

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 starting at positions that aren't multiples of three.
  - Using this information, sort the suffixes at positions that *are* at multiples of three.
  - Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall 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 starting at positions that aren't multiples of three.
- Using this information, sort the suffixes at positions that *are* at multiples of three.
- Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall 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

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

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

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

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

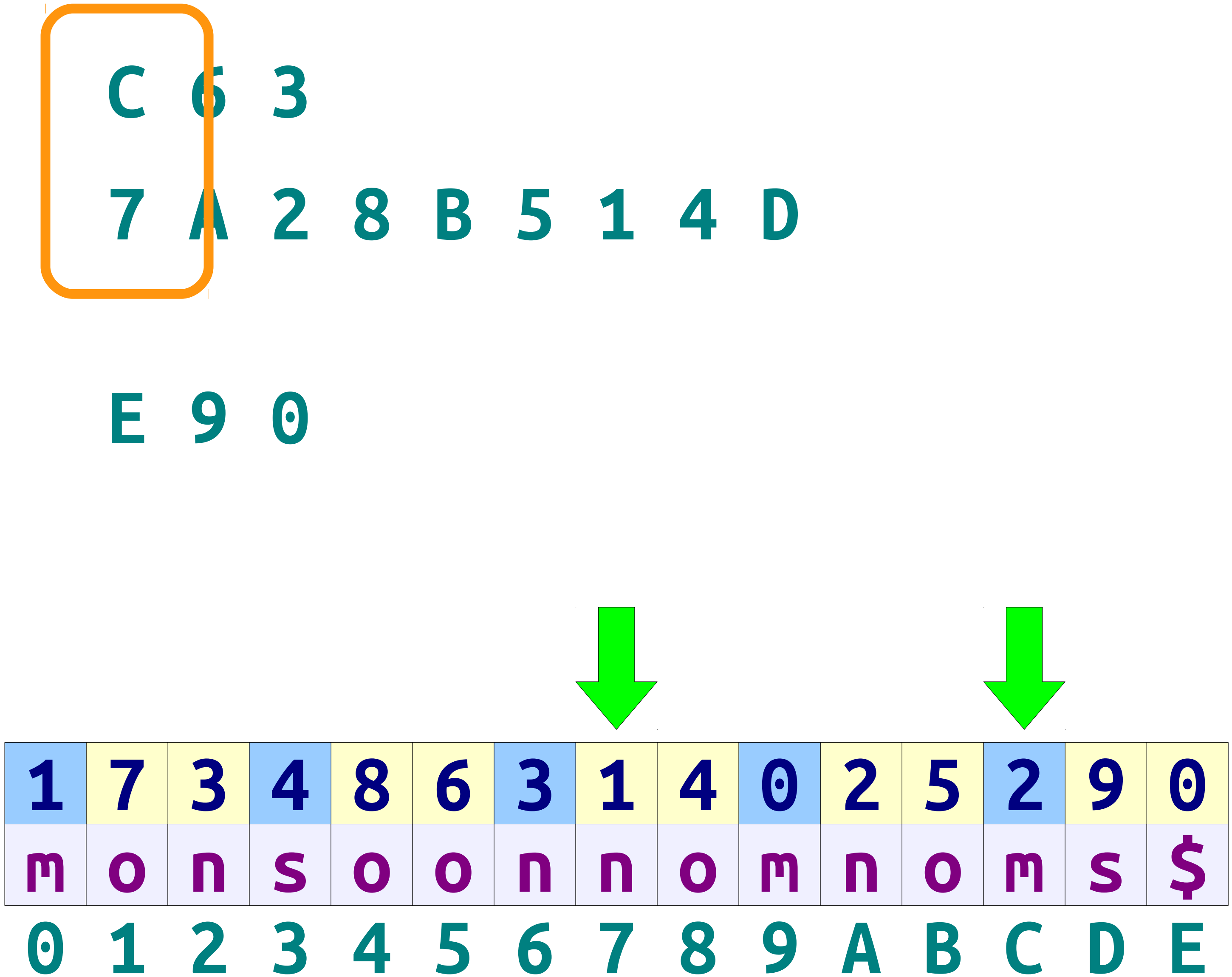
C 6 3

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

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

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	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

n	1

E 9 0 C 6

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

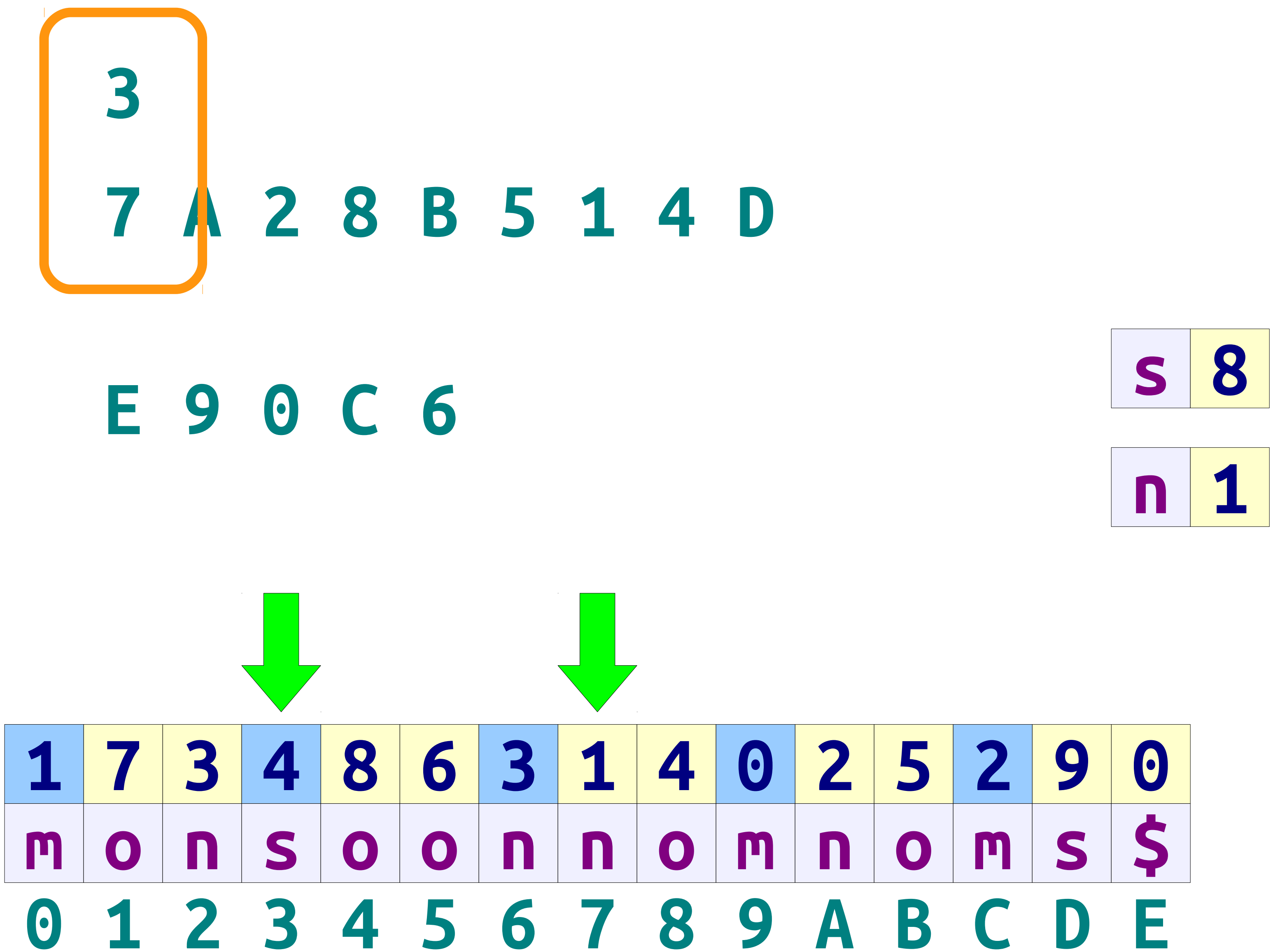
3

7 A 2 8 B 5 1 4 D

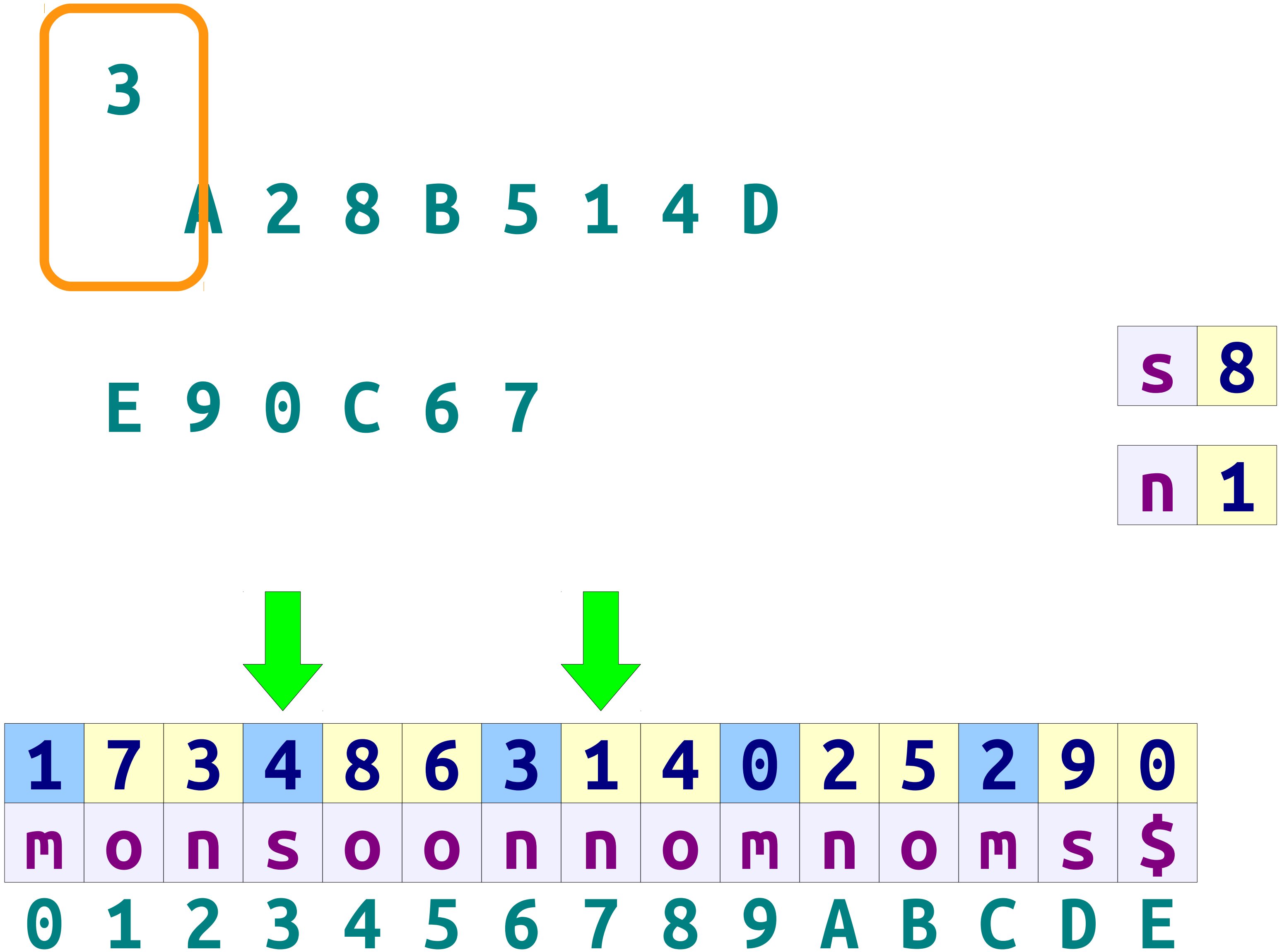
E 9 0 C 6

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


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

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

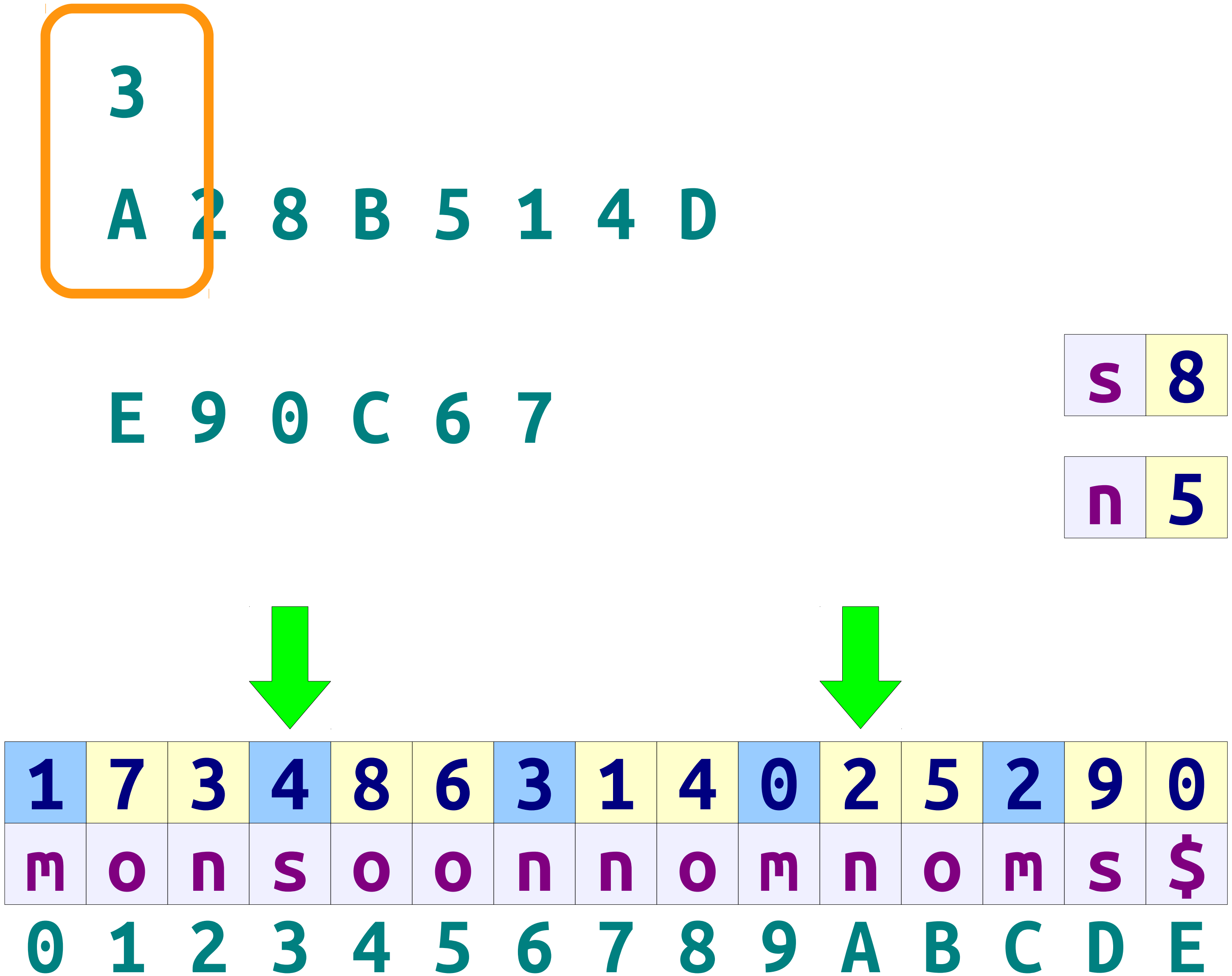
3

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

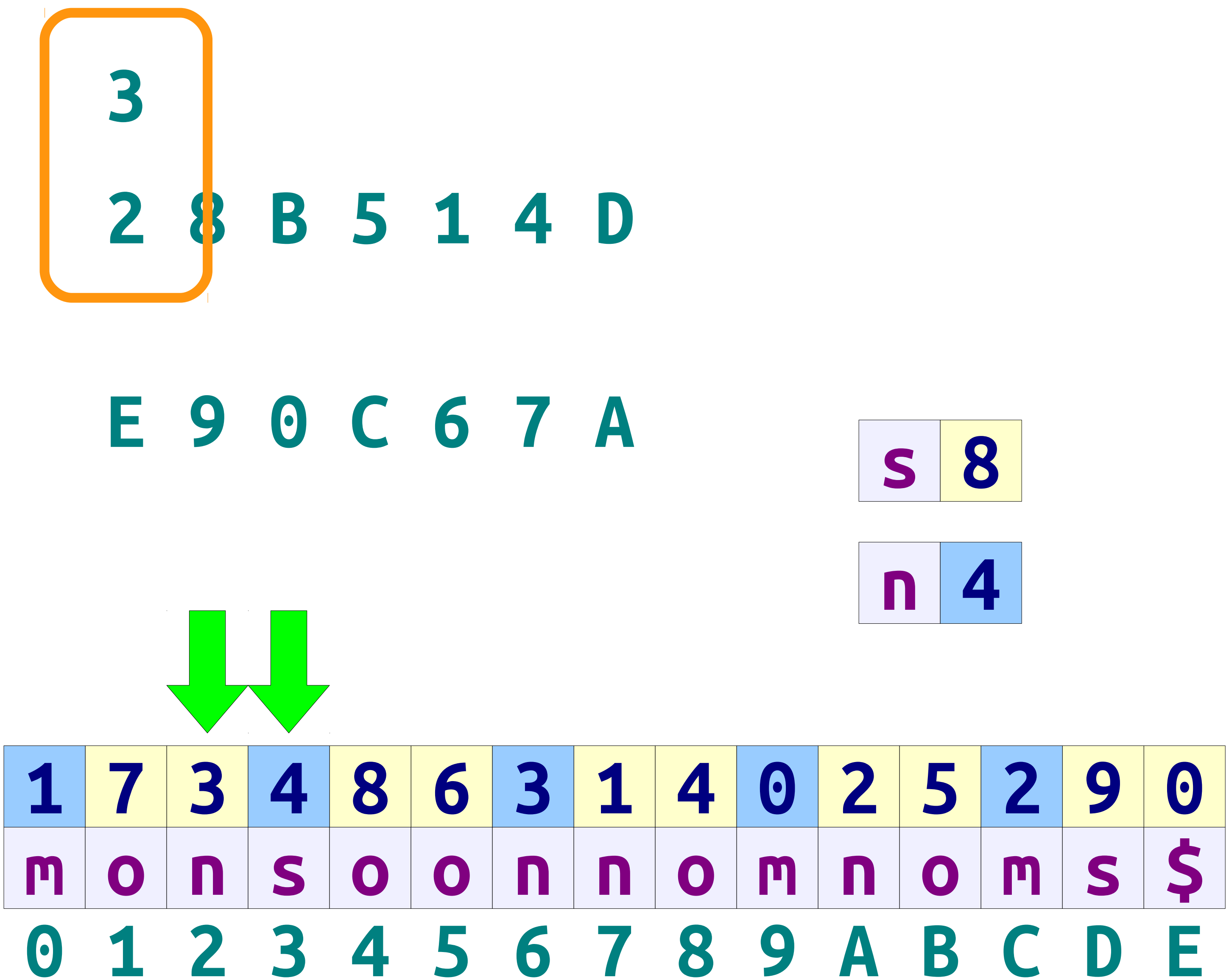
3

2 8 B 5 1 4 D

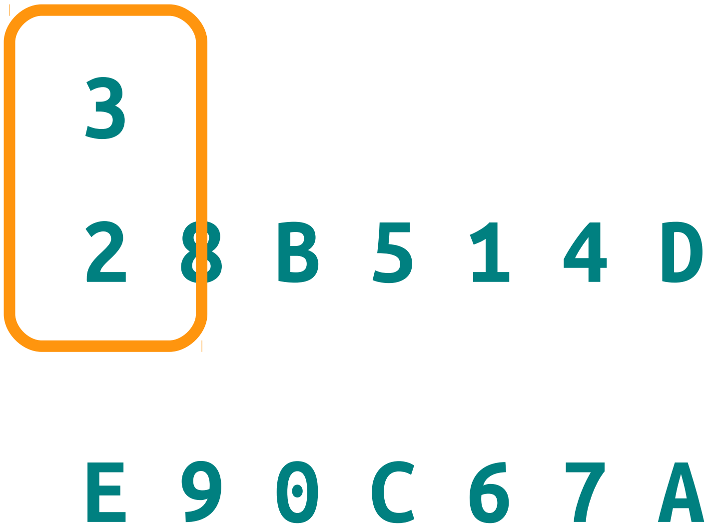
E 9 0 C 6 7 A

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

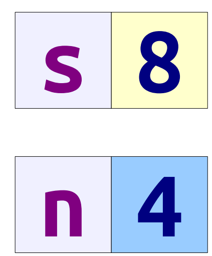
The Merging

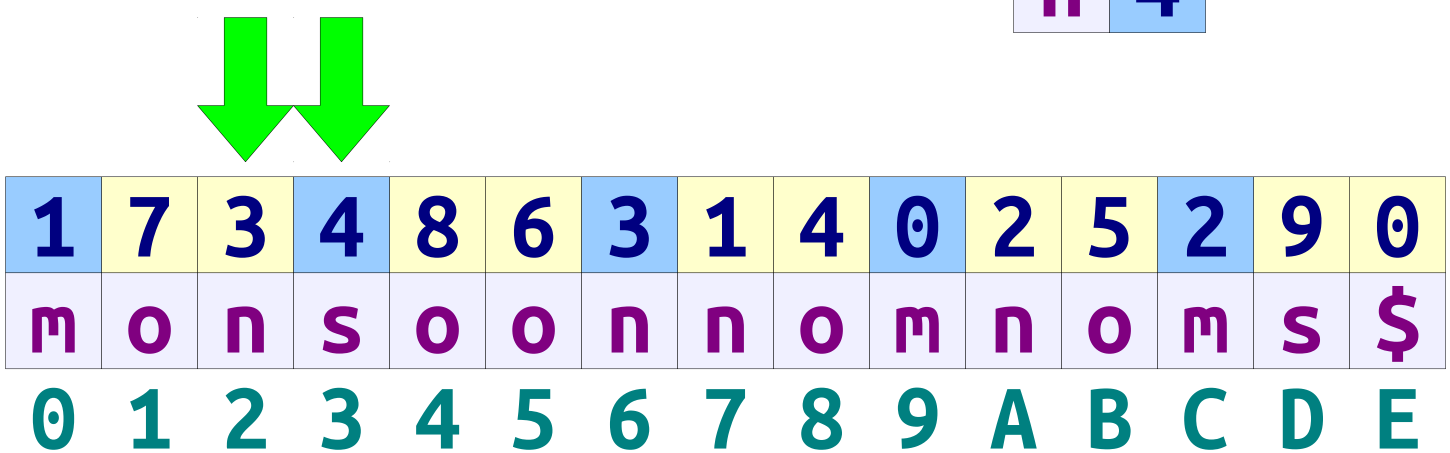

3
2 8 B 5 1 4 D
E 9 0 C 6 7 A
s 8
n 4
1 7 3 4 8 6 3 1 4 0 2 5 2 9 0
m o n s o o n n o m n o m s $
0 1 2 3 4 5 6 7 8 9 A B C D E

The Merging

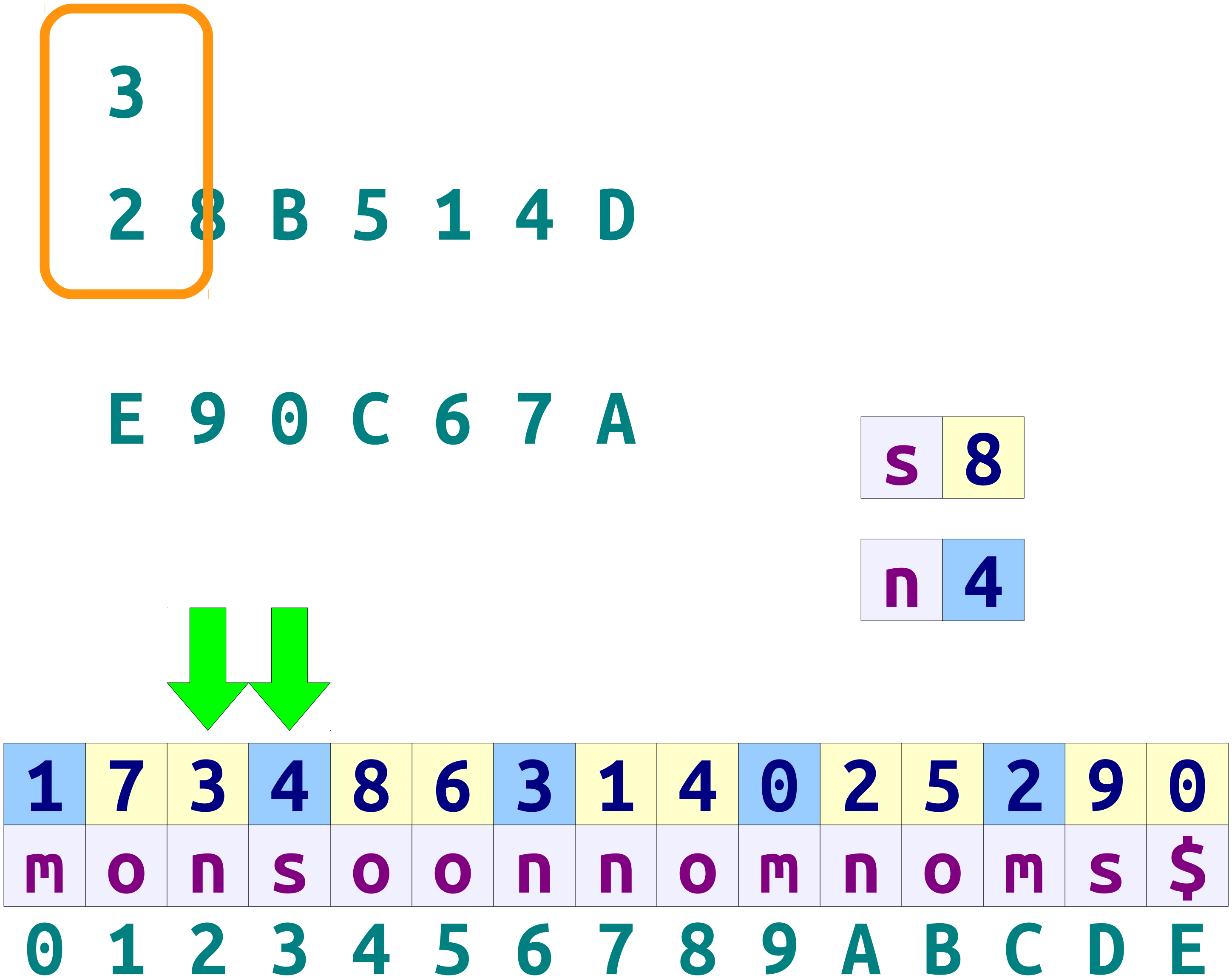


In this case it doesn't matter, but what would happen if the first letters were the same? We don't know the relative ordering of the suffixes.

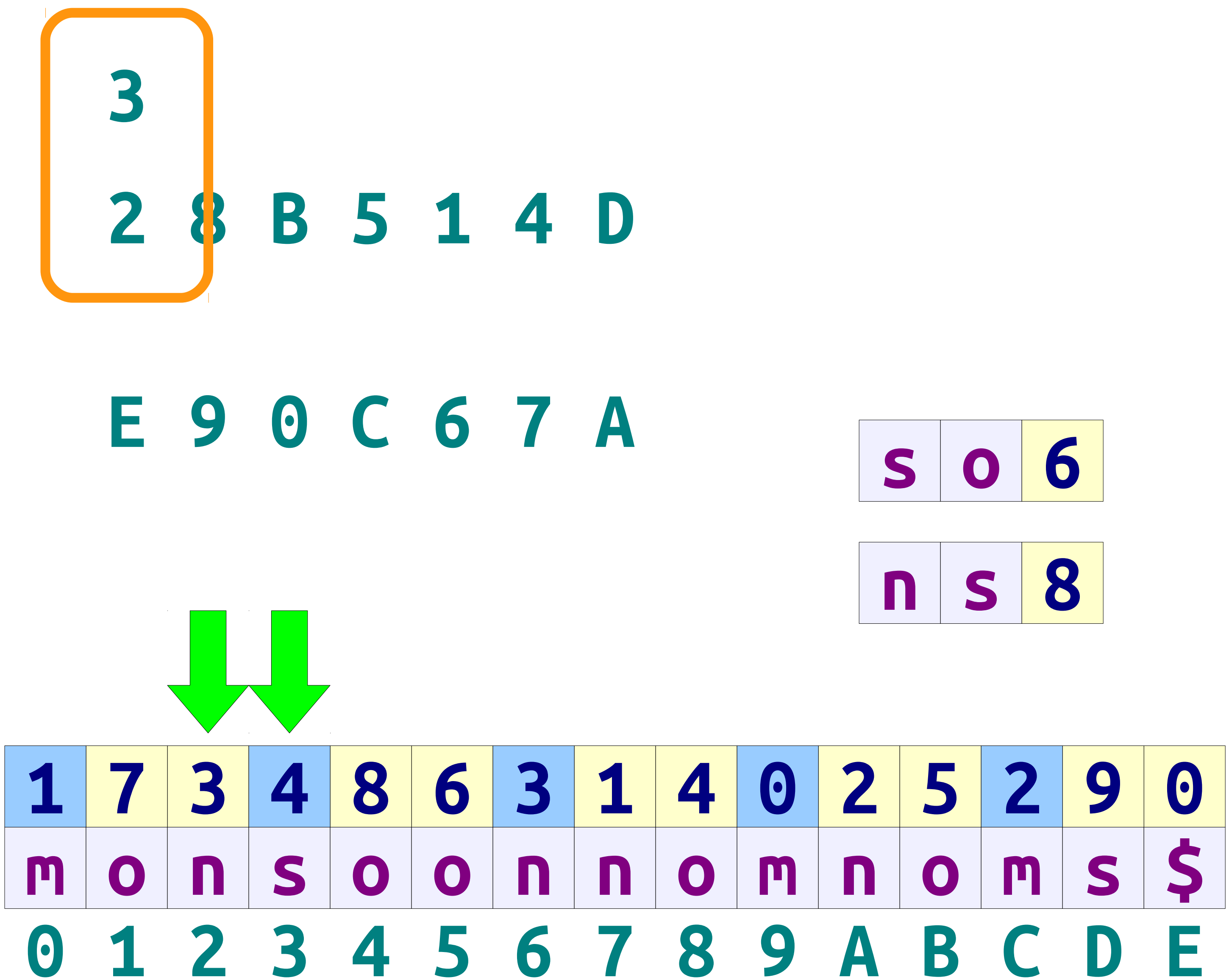




The Merging

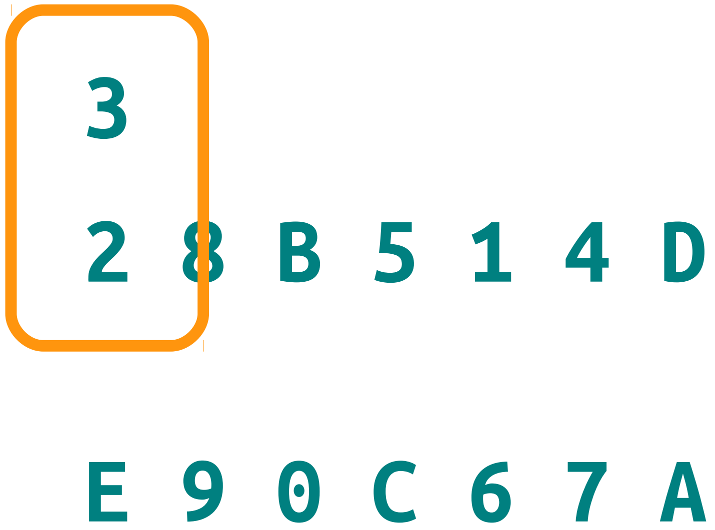

3
2 8 B 5 1 4 D
E 9 0 C 6 7 A
s 8
n 4
1 7 3 4 8 6 3 1 4 0 2 5 2 9 0
m o n s o o n n o m n o m s $
0 1 2 3 4 5 6 7 8 9 A B C D E

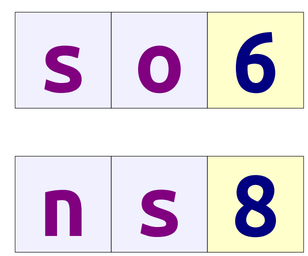
The Merging

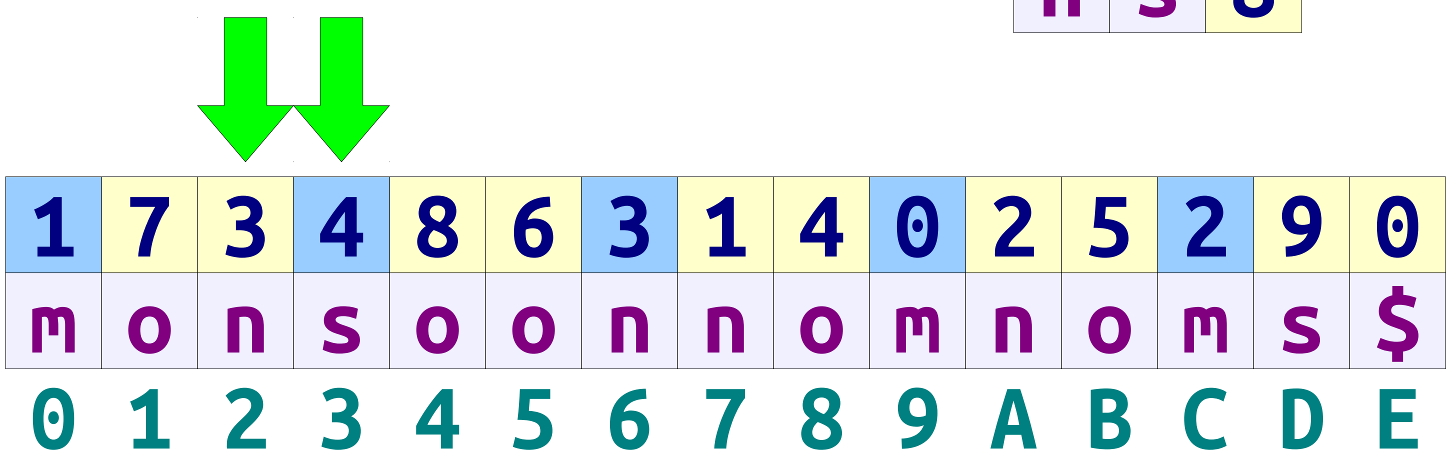

3
2 8 B 5 1 4 D
E 9 0 C 6 7 A
s o 6
n s 8
1 7 3 4 8 6 3 1 4 0 2 5 2 9 0
m o n s o o n n o m n o m s $
0 1 2 3 4 5 6 7 8 9 A B C D E

The Merging

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







The Merging

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

3

8 B 5 1 4 D

E 9 0 C 6 7 A 2

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

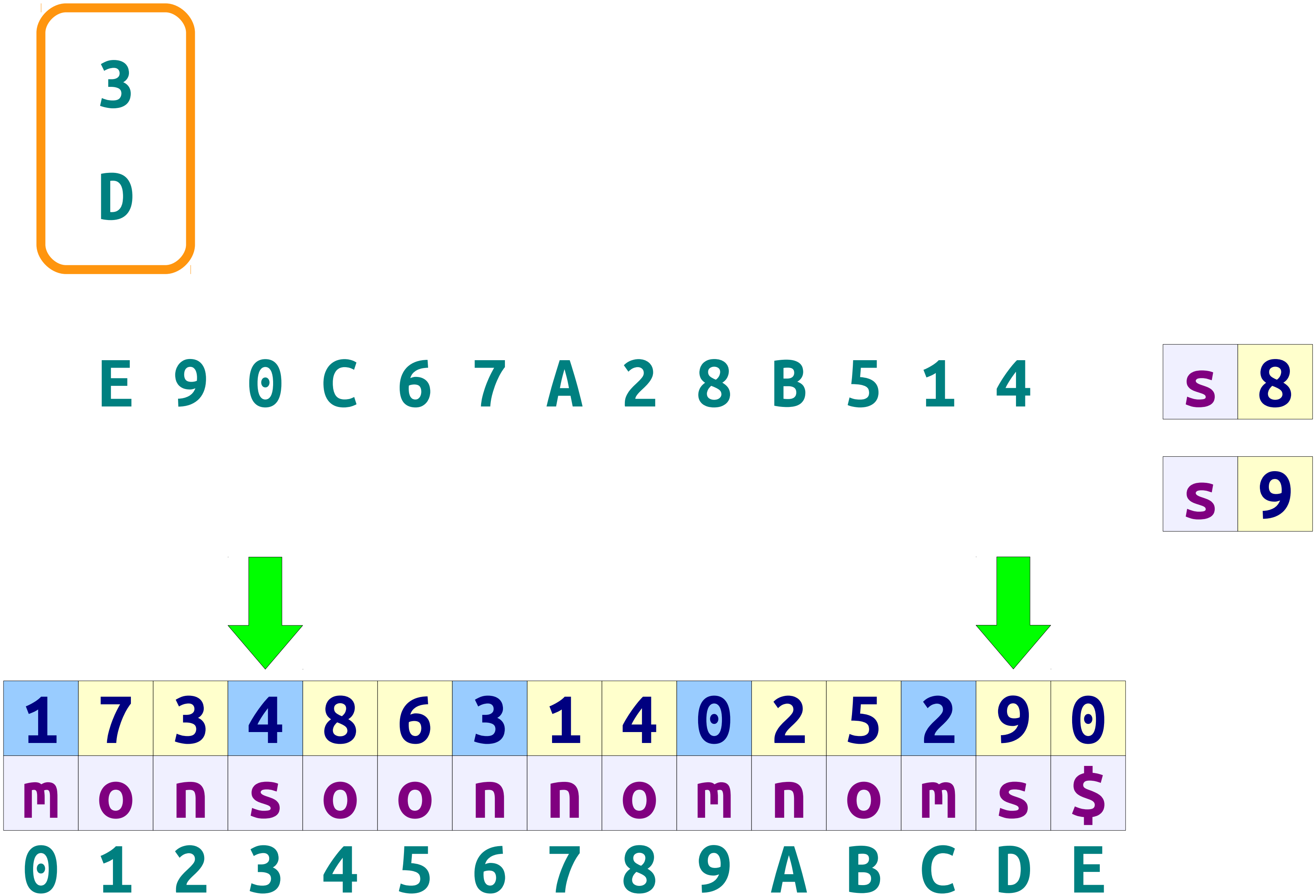
3

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 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

D

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

s	8

s	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

D

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

- Comparing any two suffixes requires at most O(1) work because we can use the existing ranking of the suffixes.
- There are a total of m suffixes to merge.
- Total runtime: **O(m)**.

DC3, Intuitively

- At a high-level, DC3 works as follows:
  - Recursively get the sorted order of all suffixes starting at positions that aren't multiples of three.
  - Using this information, sort the suffixes at positions that *are* at multiples of three.
  - Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall 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 starting at positions that aren't multiples of three.
- Using this information, sort the suffixes at positions that *are* at multiples of three.
- Using a standard merge algorithm (à la mergesort), merge the sorted lists of suffixes together into the overall suffix array.

The details are beautiful, but tricky.

The Hard Part

- 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.
- The details are a bit magical.

Some Assumptions

- We're going to assume the initial input alphabet consists of a set of integers 0, 1, 2, …, $|\Sigma| - 1$.
- 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.

The Crazy Step

- Begin by computing $T$$[1:] and $T$$[2:] and padding each with $ until the lengths are multiples of three.

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

The Crazy Step

- Begin by computing $T\$[1:]$ and $T\$[2:]$ and padding each with \$ until the lengths are multiples of three.

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	\$	\$	\$

The Crazy Step

- Begin by computing T\$[1:] and T\$[2:] and padding each with \$ until the lengths are multiples of three.
- 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	$	$	$

The Crazy Step

- Begin by computing $T\$[1:]$ and $T\$[2:]$ and padding each with \$ until the lengths are multiples of three.
- Then, concatenate those strings together.

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

Um, Why?

- **Claim:** The relative order of the suffixes in the first half of the string starting at positions in T_1 and the suffixes in the second half of the string at positions in T_2 is the same as the relative order of those suffixes in T.

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

Um, Why?

- **Claim:** The relative order of the suffixes in the first half of the string starting at positions in T_1 and the suffixes in the second half of the string at positions in T_2 is the same as the relative order of those suffixes in T.

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

Um, Why?

- **Claim:** The relative order of the suffixes in the first half of the string starting at positions in T_1 and the suffixes in the second half of the string at positions in T_2 is the same as the relative order of those suffixes in T.

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

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

Um, Why?

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

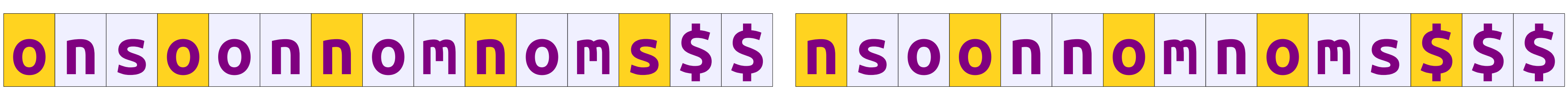
o	n	s	o	o	n	n	o	m	n	o	m	s	$	$

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

So, Um…

… we just doubled the size of our input string.

You're not supposed to do that in a divide-and-conquer algorithm.



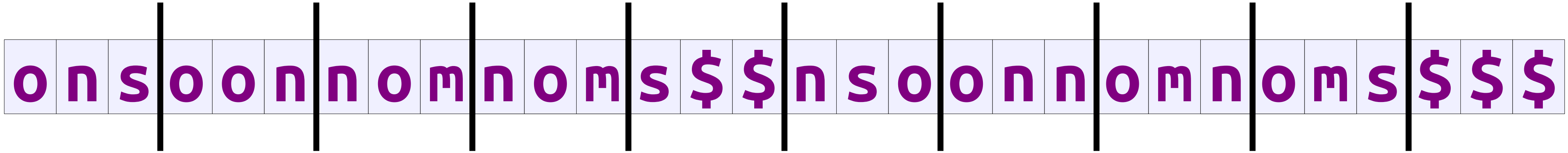
Playing with Blocks

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

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

Playing with Blocks

- **Key Insight:** Break the input apart into blocks of size three.
- Think about what happens if we compare two suffixes:
  - 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.



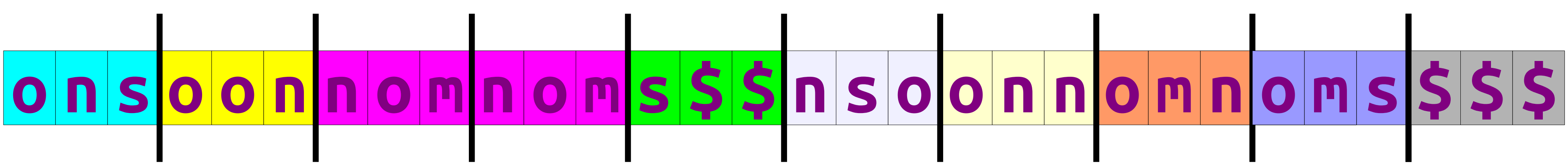
Playing with Blocks

- Notice: All suffixes from T_1 and T_2 are now at positions that zero mod 3.
- All we care about are the suffixes at those positions.
- We can treat each block of three characters as a single unit!

ons	oon	nom	nom	s$$	nso	onn	omn	oms	$$$

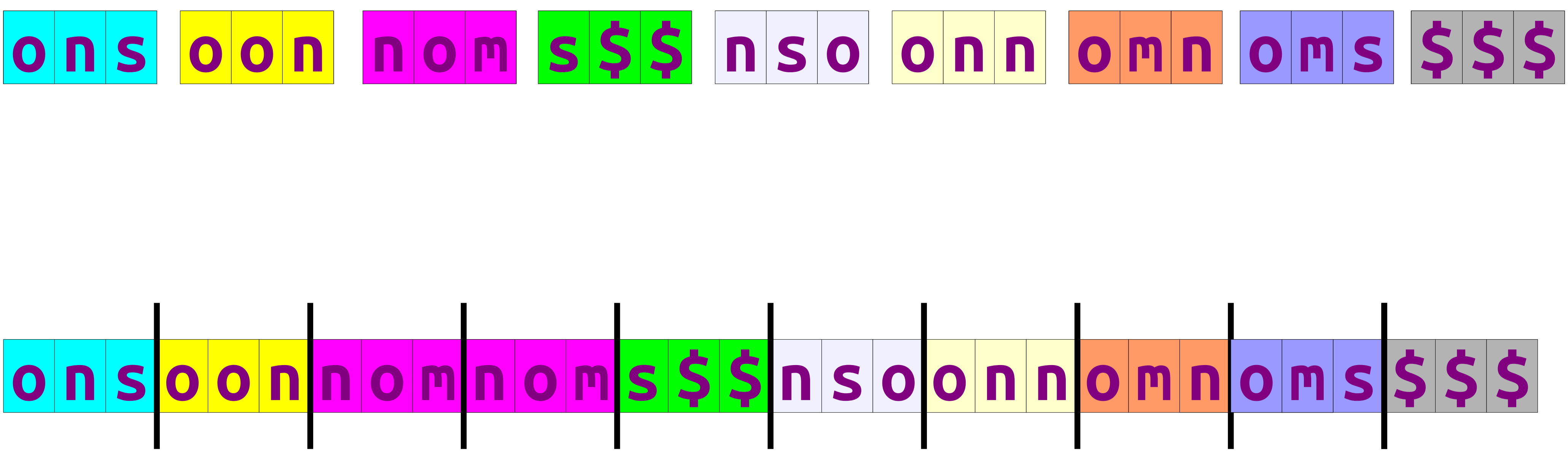
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



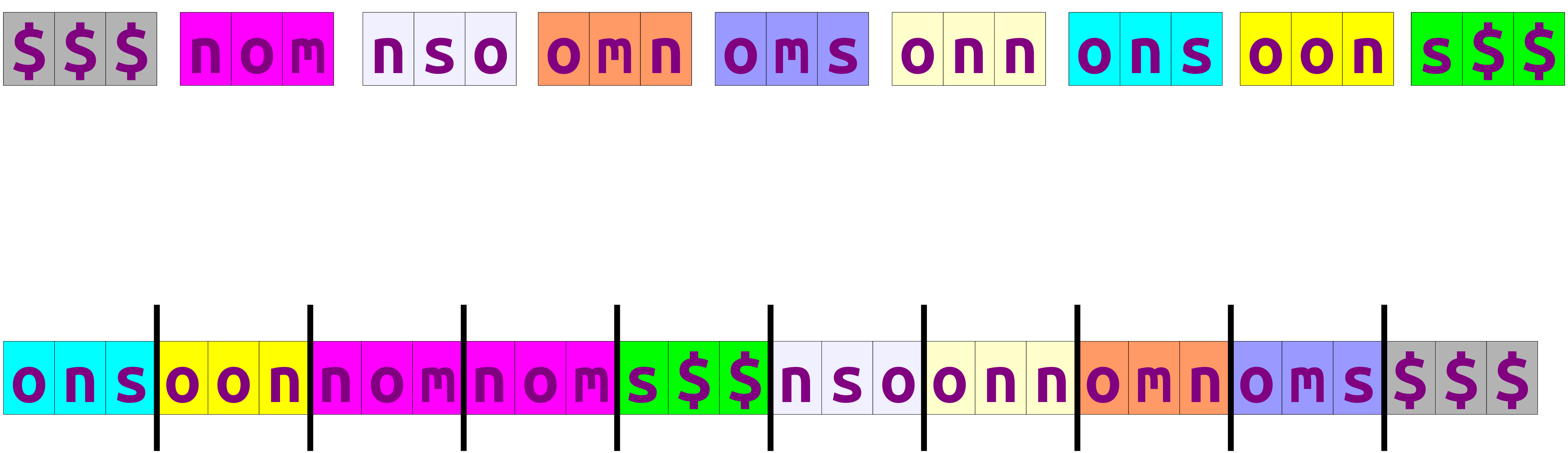
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



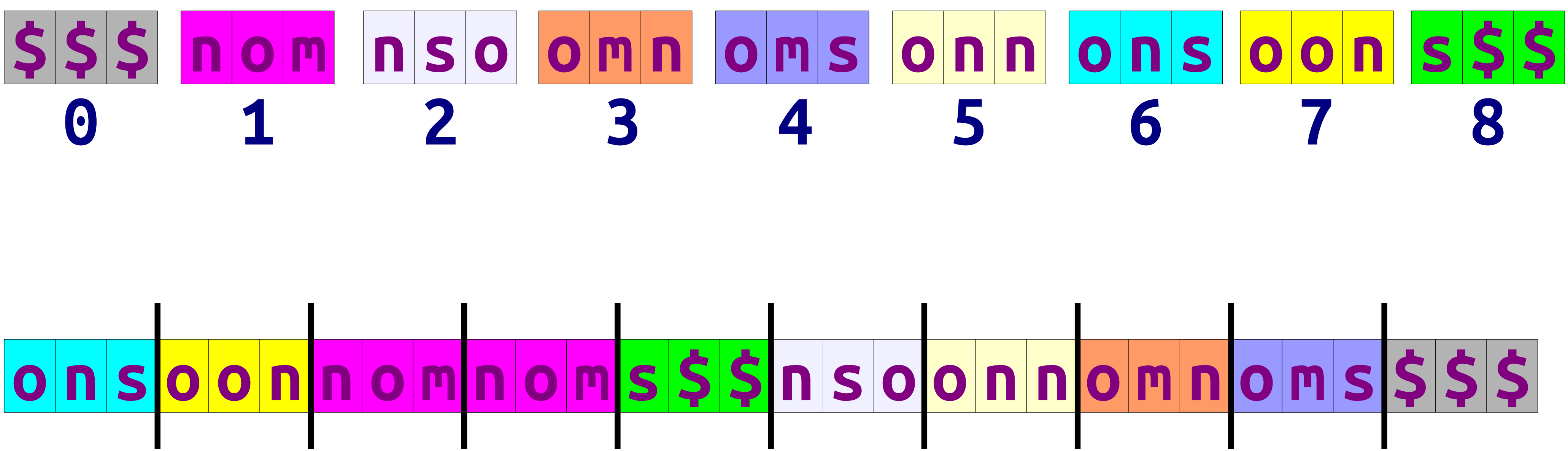
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



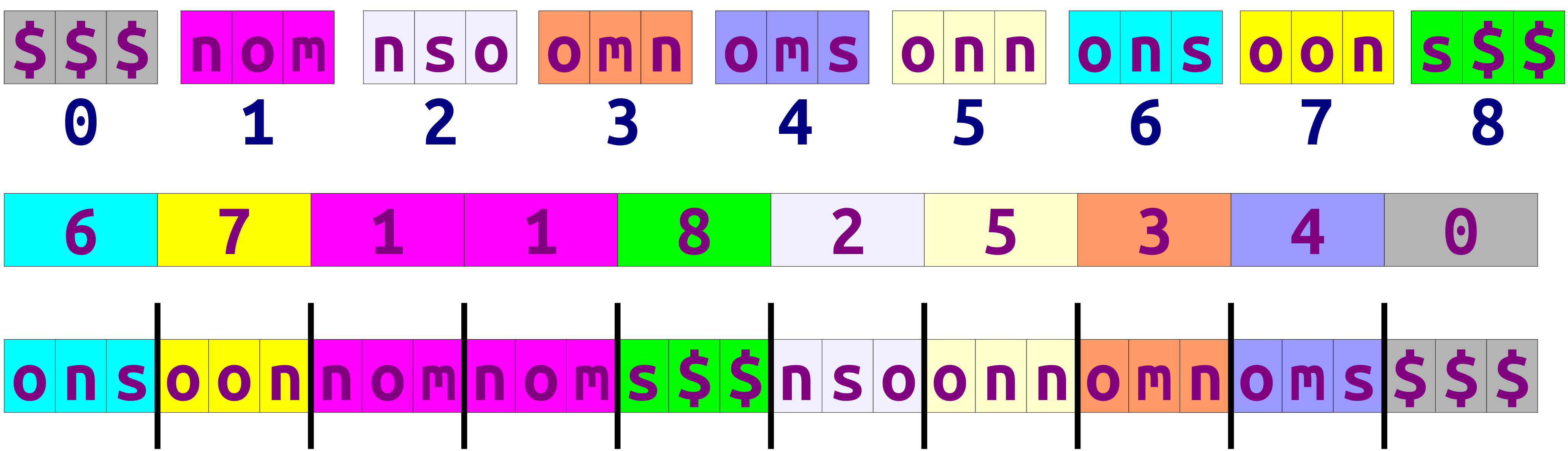
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



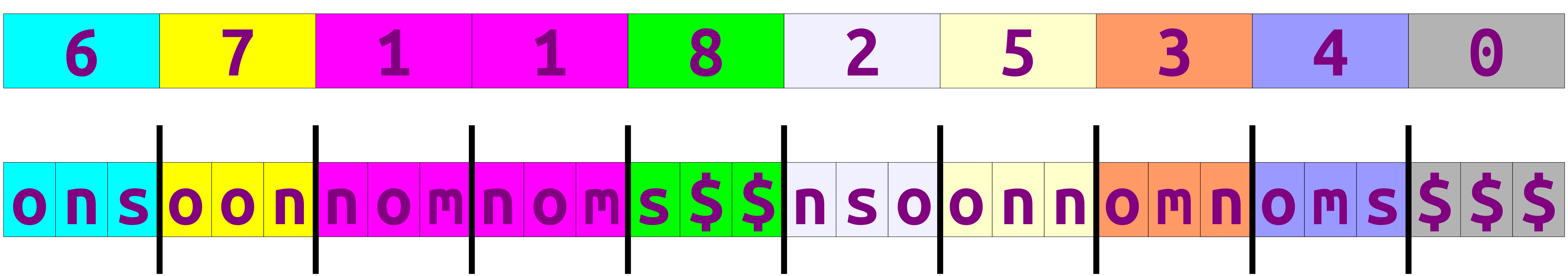
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



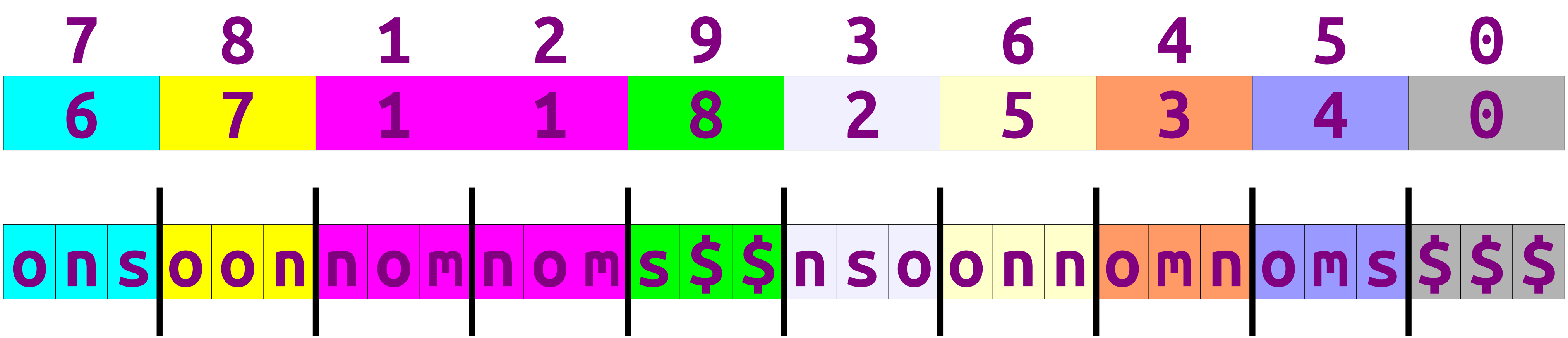
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



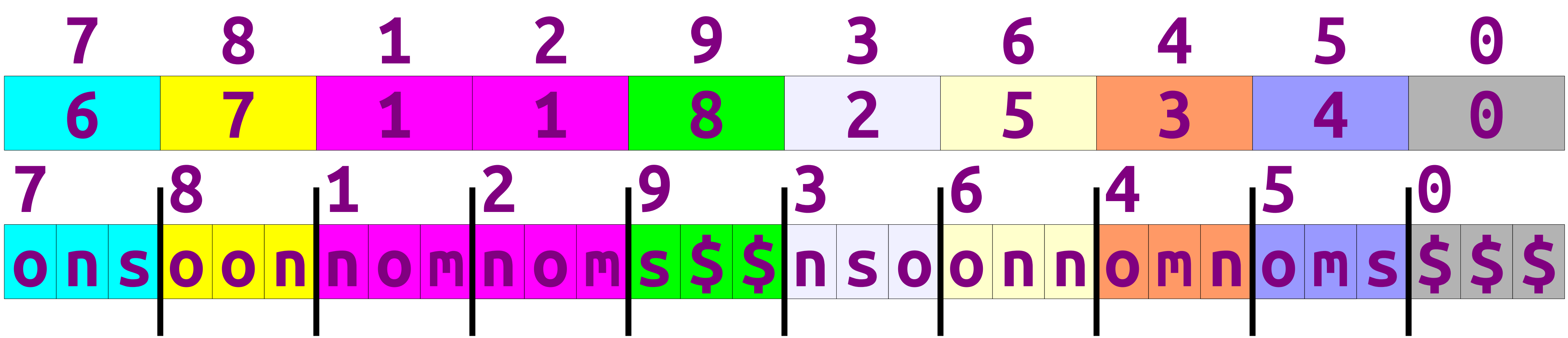
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



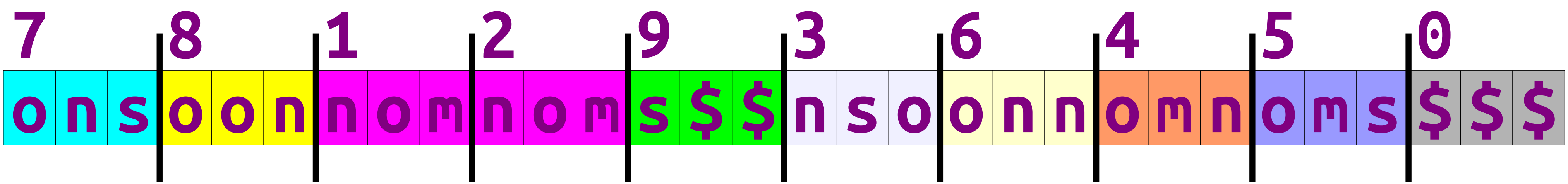
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



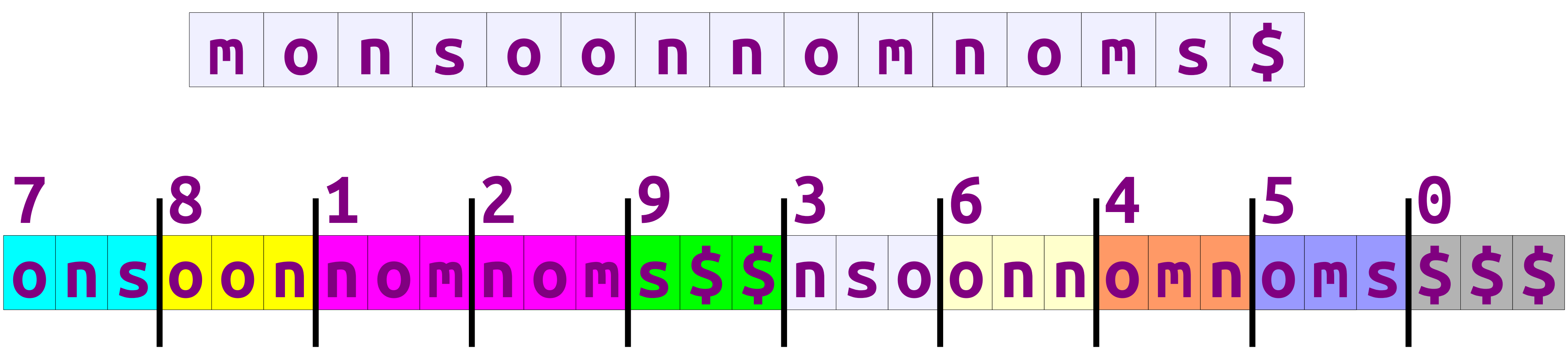
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



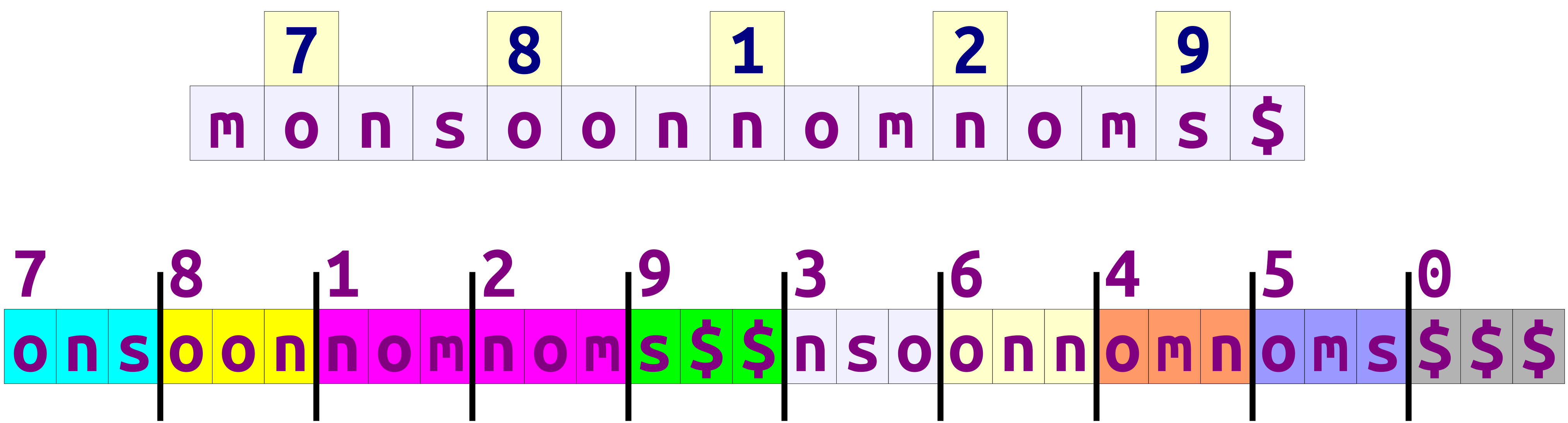
The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



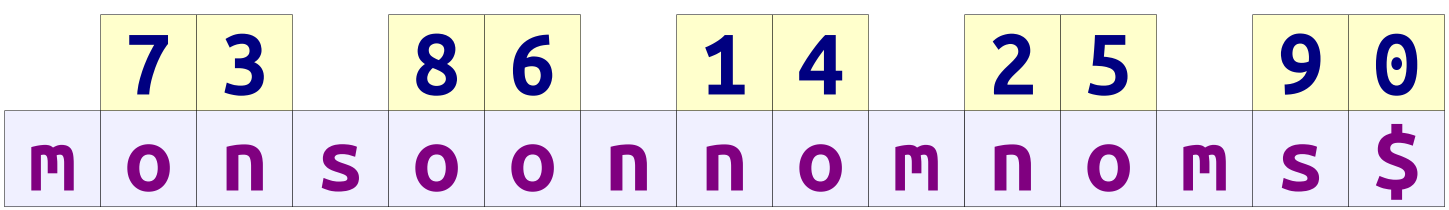
The Final Step

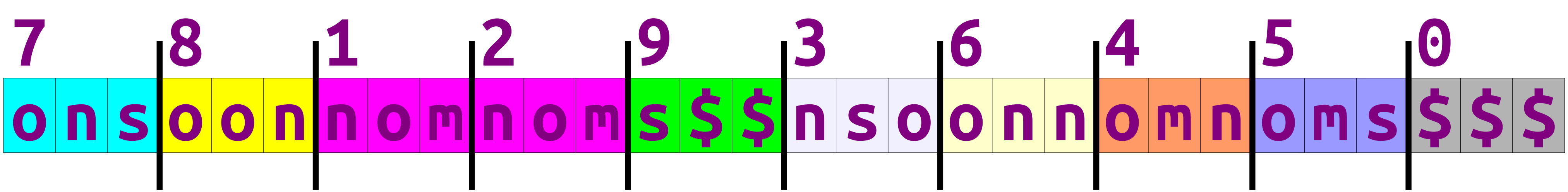
- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.



The Final Step

- **The Trick:** Treat each block of three characters as its own character.
- Can determine the relative ordering of those characters by an O(m)-time radix sort.
- To keep the alphabet small, replace each block of three characters with its index.
- Recursively compute the suffix array of that string.





Summarizing the Magic

- 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.

The Overall Algorithm

- The recursive step takes time $\Theta(m)$ plus the recursive call.
- Sorting T_0 takes time $\Theta(m)$.
- Merging T_0, T_1, and T_2 takes time $\Theta(m)$.
- Recurrence relation:

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

- Overall runtime: $\boldsymbol{\Theta(m)}$.

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.

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

- **Randomized Data Structures**
  - What happens if you trade accuracy or reliability for speed?
- **Streams and Sketches**
  - A new framework for envisioning algorithms.
- **Count-Min Sketches**
  - A simple and surprisingly powerful data structure for finding frequent elements.
- **Count Sketches**
  - A related data structure with slightly different guarantees.