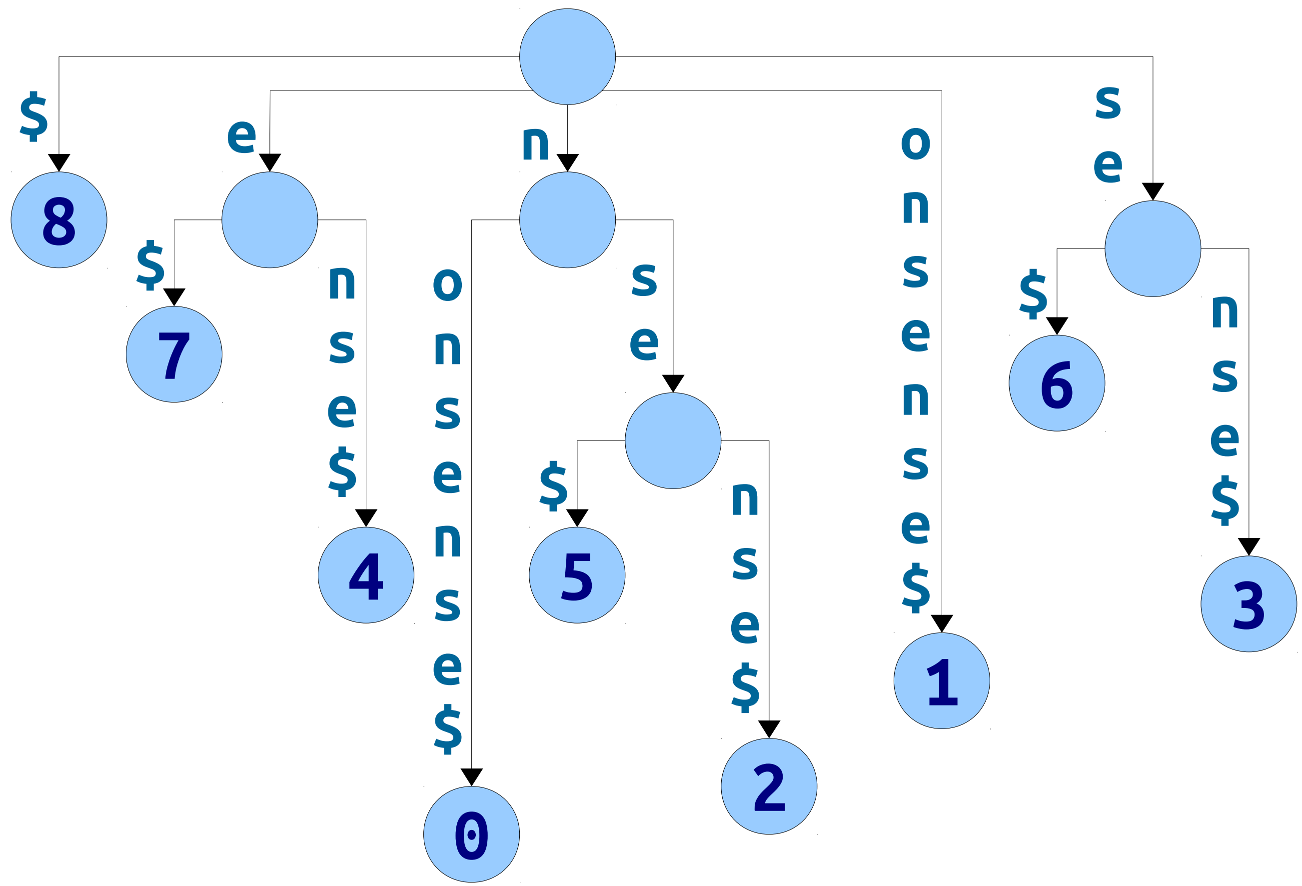
Building Suffix Arrays

Recap from Last Time



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

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

Suffix Arrays

- A ***suffix array*** for a string T is a sorted array of the suffixes of the string $T\$$.
- Suffix arrays distill out just the first component of suffix trees: they store suffixes in sorted order.

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

ABANANABANDANA\$

Storing Suffix Arrays

- ***Idea:*** Don't store the suffixes themselves. Just store the starting positions of the suffixes.
- Space: $\Theta(\boldsymbol{m})$, and with only one machine word used per character of input.

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

ABANANABANDANA$
012345678901234

LCP Arrays

- The ***LCP array***, often denoted ***H***, is an array where $H[i]$ is the length of the LCP of the ith and $(i+1)$st suffixes in the suffix array.
- LCP arrays can be computed in time $O(m)$ using ***Kasai's algorithm***.

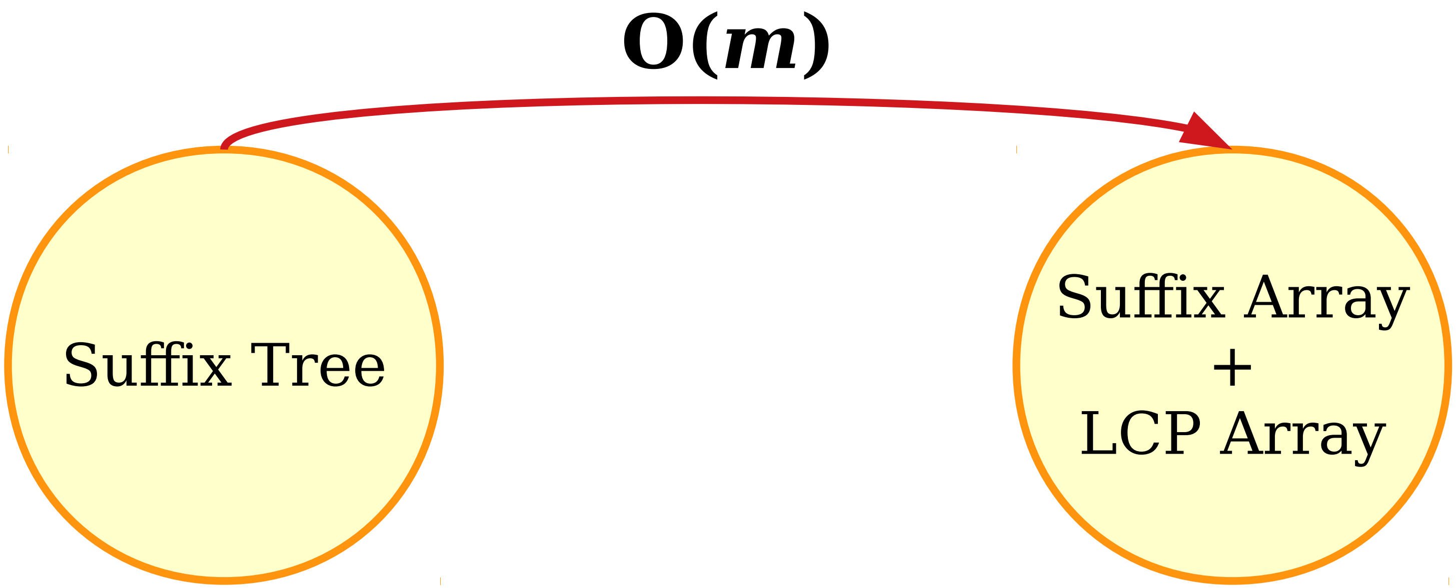
Suffix	H
$	0
A$	1
ABANANABANDANA$	4
ABANDANA$	1
ANA$	3
ANABANDANA$	3
ANANABANDANA$	2
ANDANA$	0
BANANABANDANA$	3
BANDANA$	0
DANA$	0
NA$	2
NABANDANA$	2
NANABANDANA$	1
NDANA$	

ABANANABANDANA$

Runtime Analysis

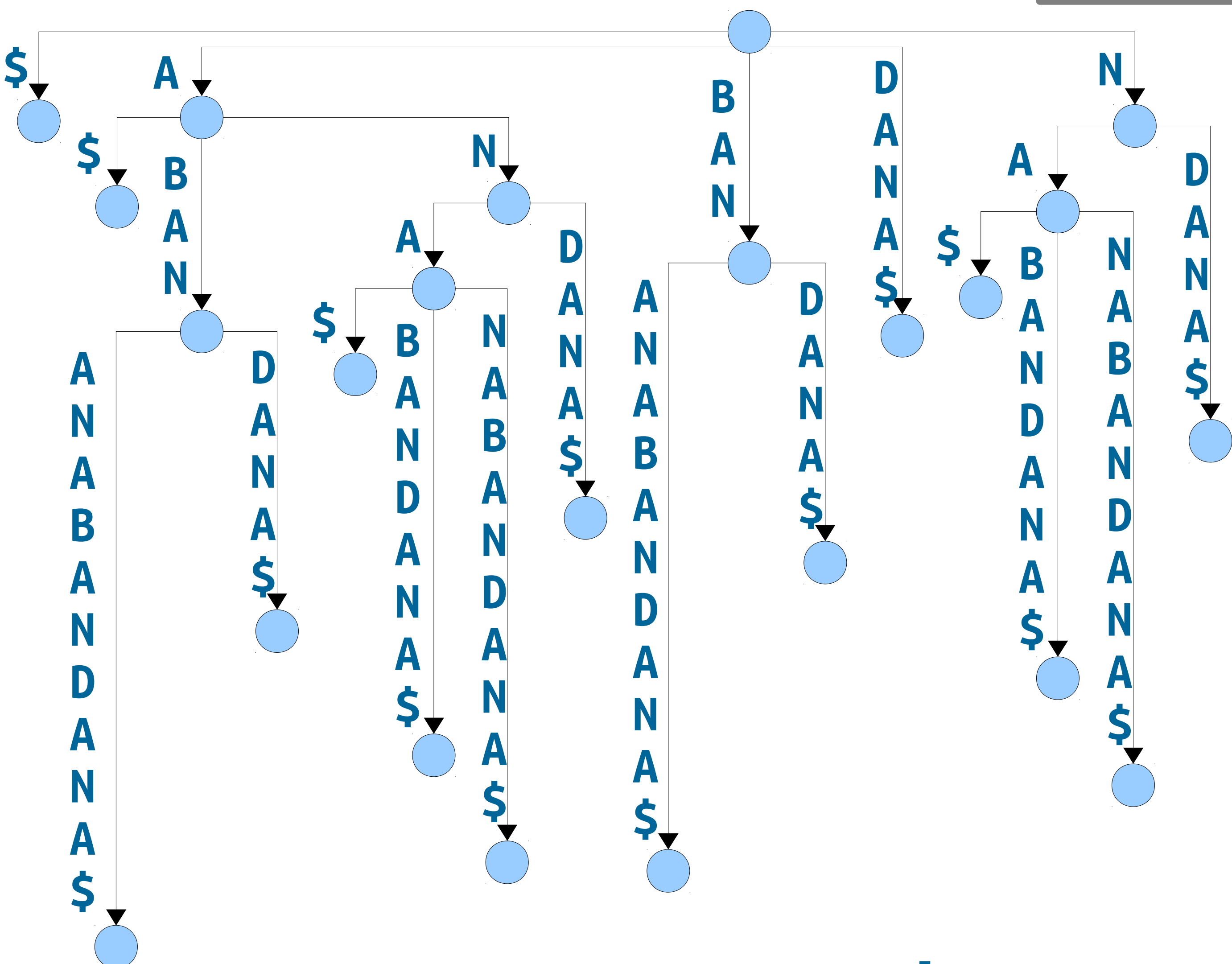
- Suffix trees give an
  - $\langle O(m), O(n + z)\rangle$-time data structure for the substring search problem, and an
  - $O(m)$-time solution for longest repeated substring.
- Suffix arrays, combined with LCP arrays, give an
  - $\langle O(m), O(n + \log m + z)\rangle$-time data structure for the substring search problem, and an
  - $O(m)$-time solution for longest repeated substring.
- All of these analyses assume that
  - we can build a suffix tree in time $O(m)$, and
  - we can build a suffix array in time $O(m)$.
- ***Question:*** How is this possible?

New Stuff!



O(*m*)
Suffix Tree
Suffix Array
+
LCP Array

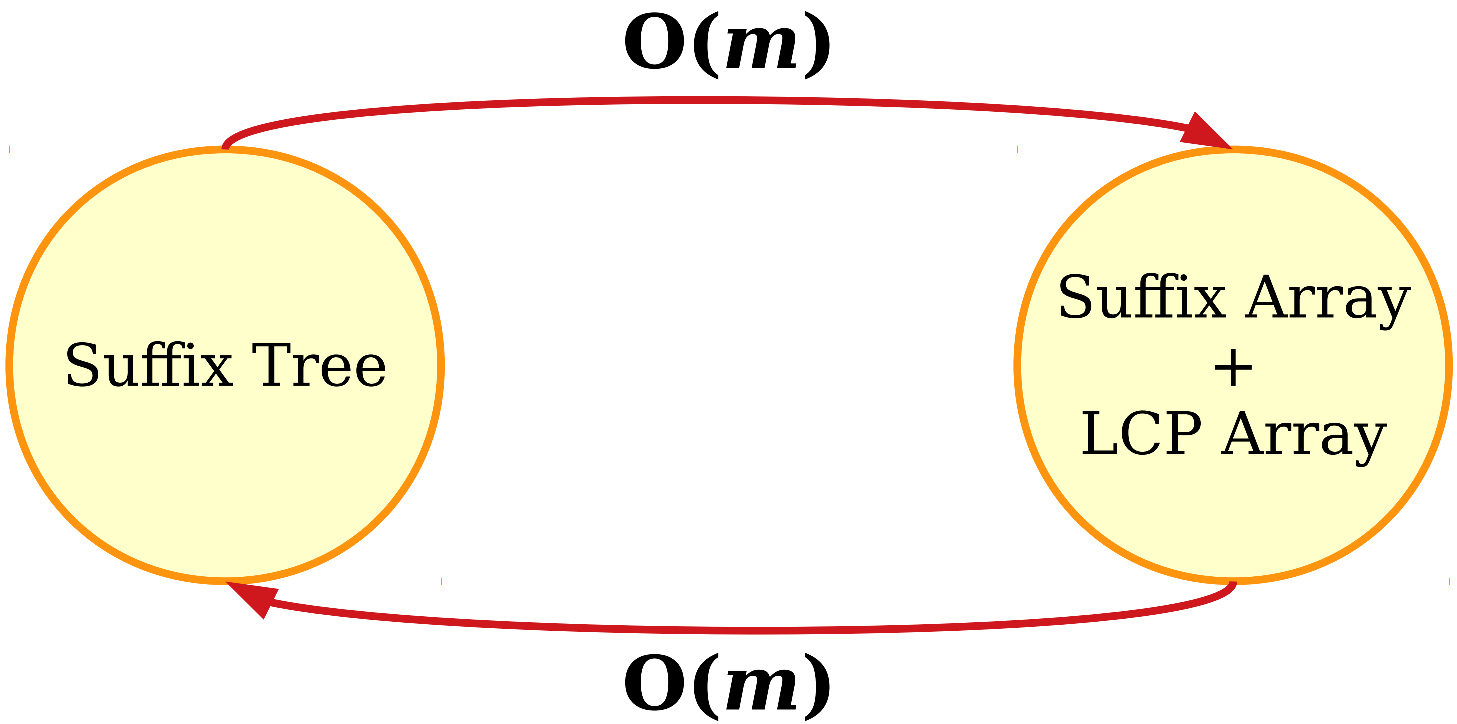
Do a DFS over the tree, visiting children in sorted order.



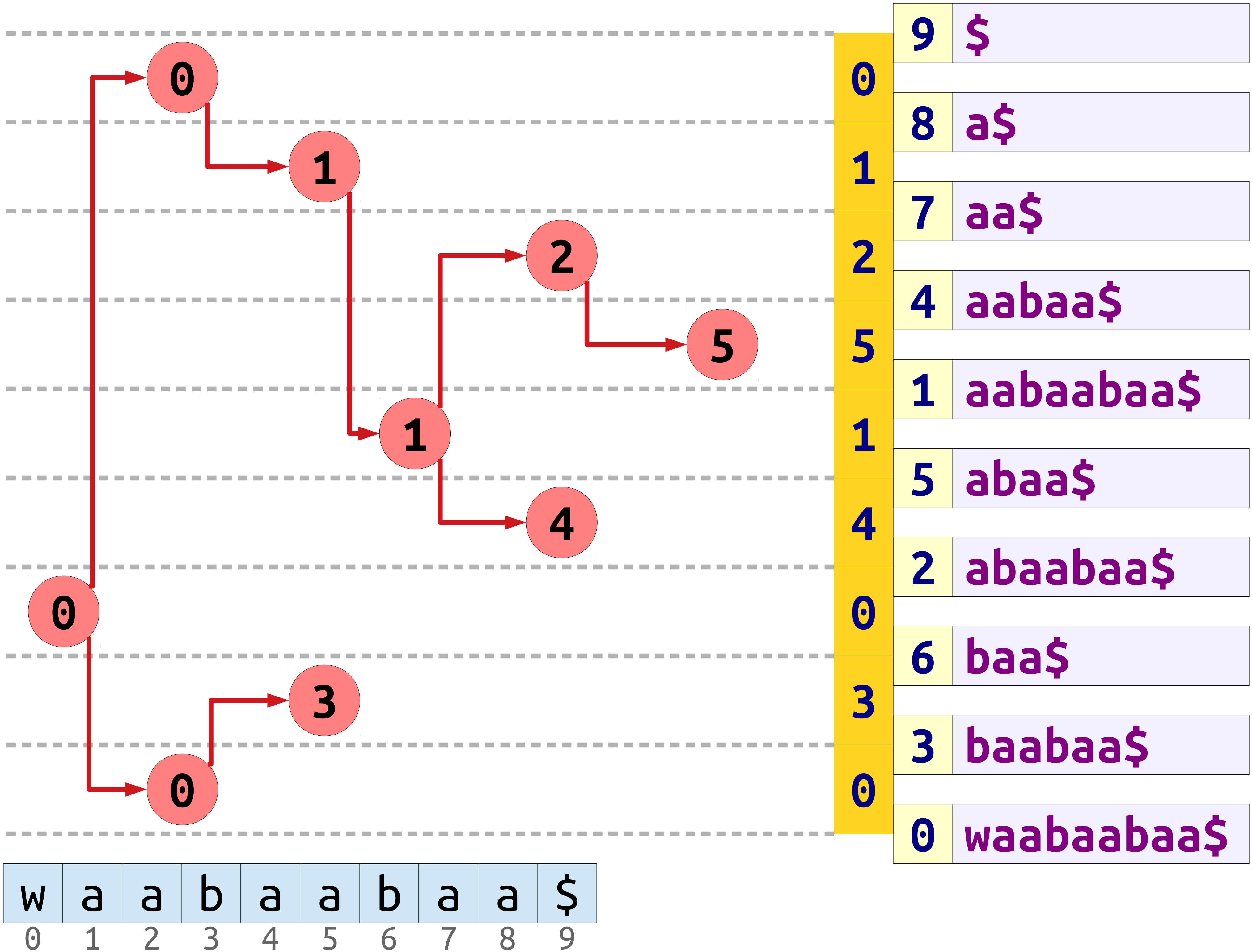
ABANANABANDANA$

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

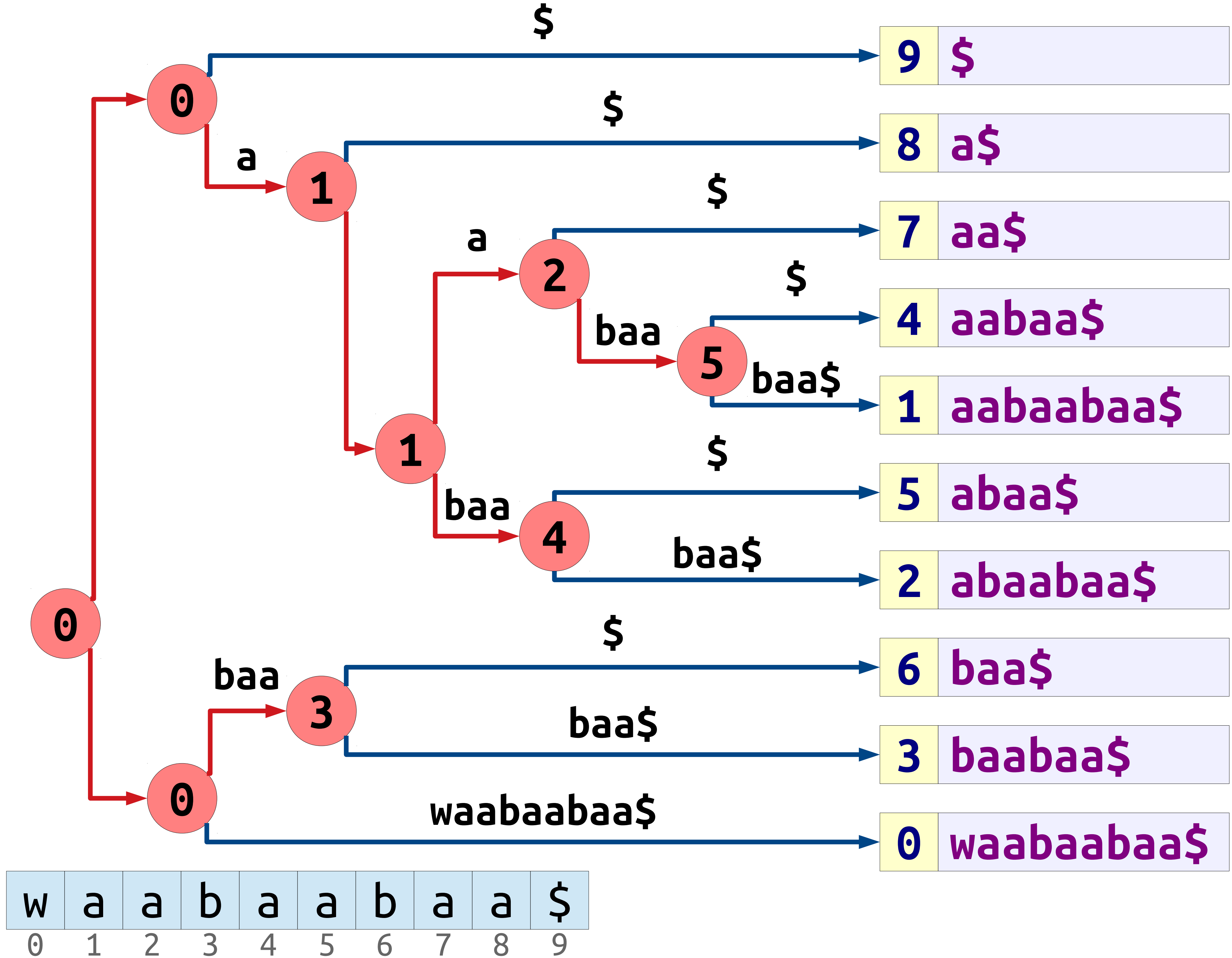
Track the length of the label on the last internal node backtracked through.



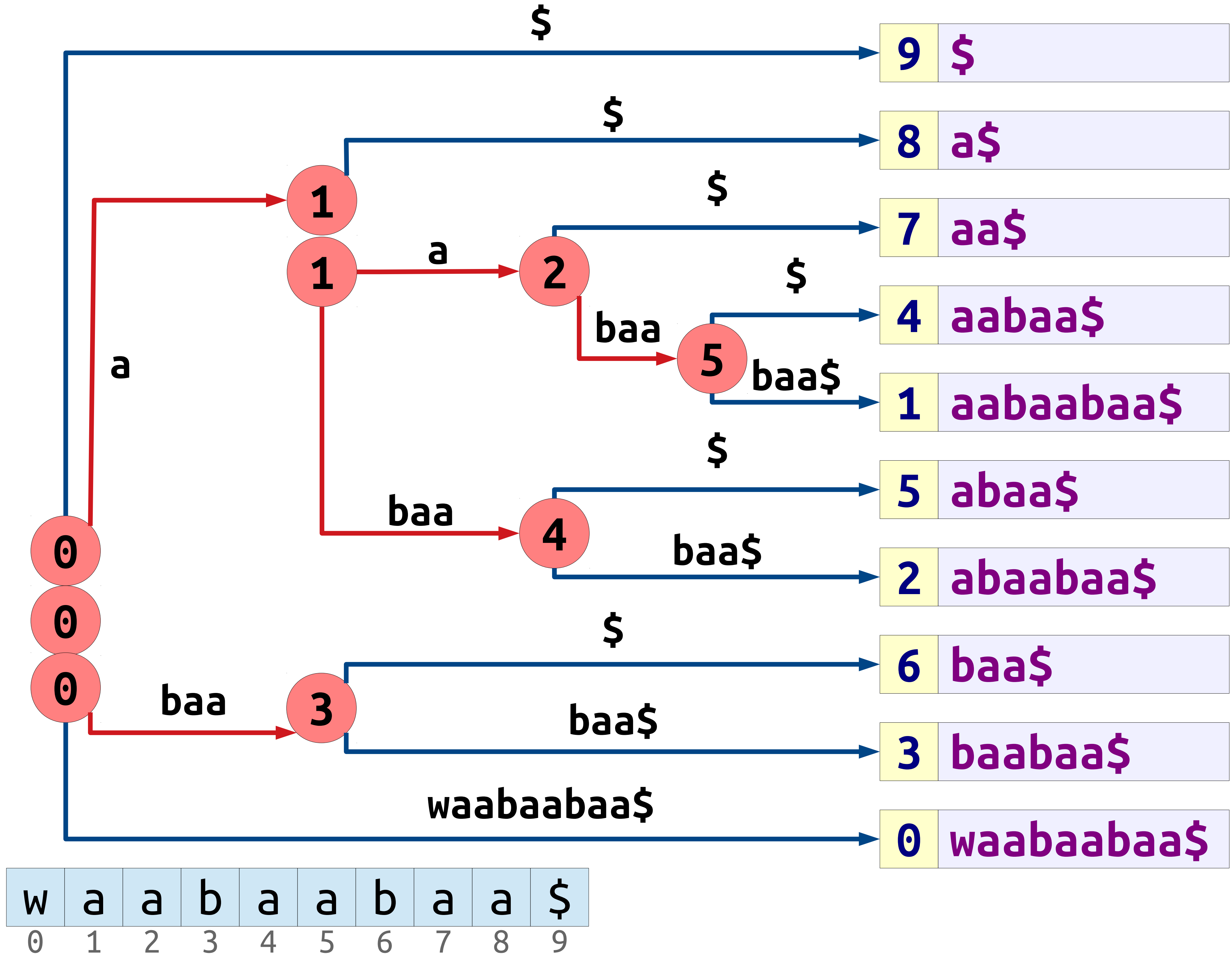
O(*m*)
Suffix Tree
Suffix Array
+
LCP Array
O(*m*)



0
1
2
5
1
4
0
3
0
9 $
8 a$
7 aa$
4 aabaa$
1 aabaabaa$
5 abaa$
2 abaabaa$
6 baa$
3 baabaa$
0 waabaabaa$
w a a b a a b a a $
0 1 2 3 4 5 6 7 8 9



$
$
$
$
baa$
$
baa$
$
baa$
waabaabaa$
a
a
baa
baa
baa
0
1
2
5
1
4
0
3
0
9 $
8 a$
7 aa$
4 aabaa$
1 aabaabaa$
5 abaa$
2 abaabaa$
6 baa$
3 baabaa$
0 waabaabaa$
w a a b a a b a a $
0 1 2 3 4 5 6 7 8 9



$
9 $
$
8 a$
$
7 aa$
$
4 aabaa$
baa$
1 aabaabaa$
$
5 abaa$
baa$
2 abaabaa$
$
6 baa$
baa$
3 baabaa$
waabaabaa$
0 waabaabaa$
a
a
baa
baa
baa
0
0
0
1
1
2
3
4
5
w a a b a a b a a $
0 1 2 3 4 5 6 7 8 9

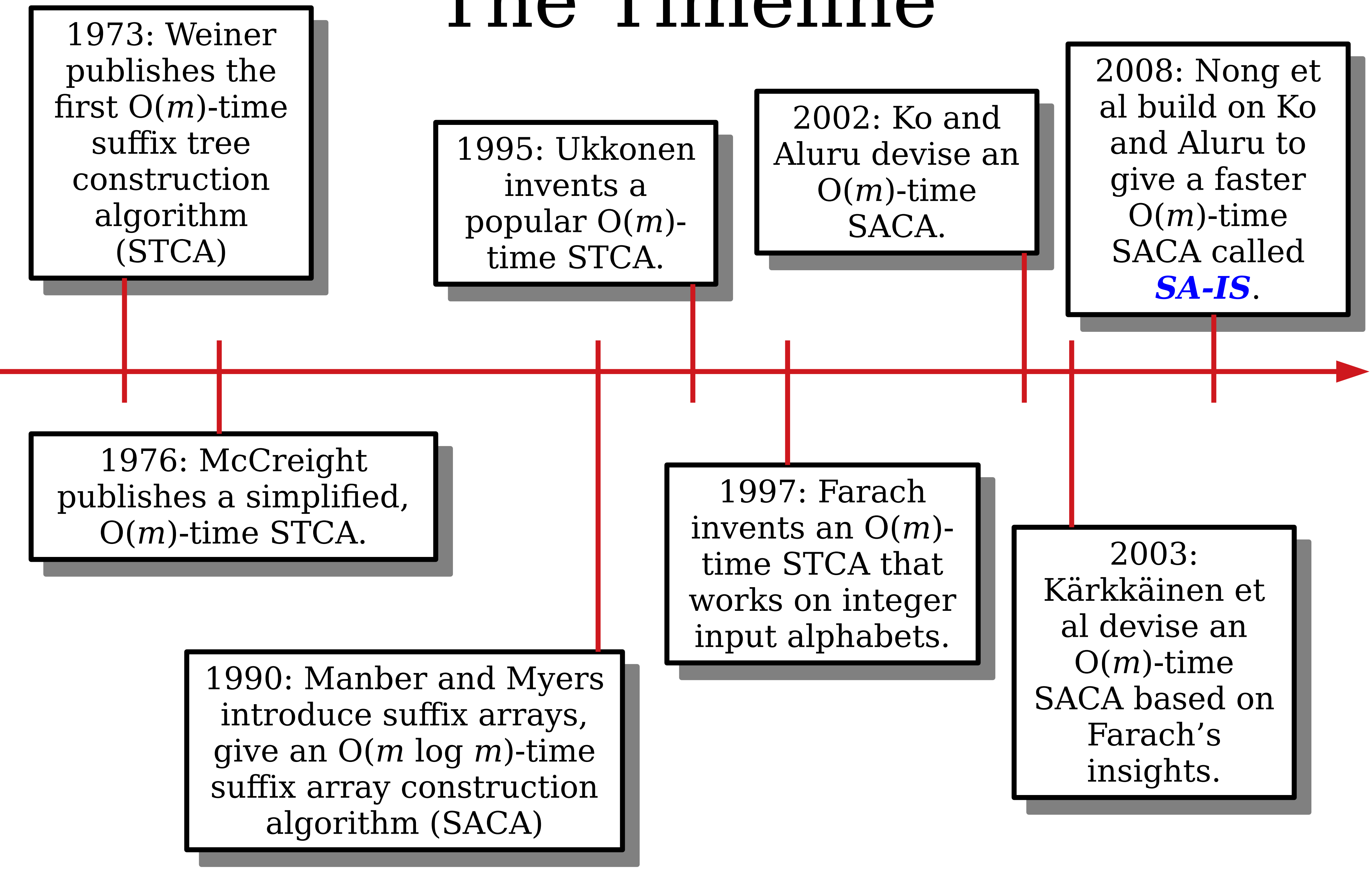
A Linear-Time Algorithm

- Construct the LCP array for the suffix array.
- Construct a Cartesian tree from that LCP array.
- Run a DFS over the Cartesian tree, adding in the suffixes in the order they appear whenever a node has a missing child.
- Fuse together any parent and child nodes with the same number in them.
- Assign labels to the edges based on the LCP values.
- Total time: **$O(m)$**.

Question: Why does this work?
As a hint, what's the connection between LCP arrays and suffix trees?

Constructing Suffix Arrays

The Timeline



The Timeline

1973: Weiner publishes the first O(m)-time suffix tree construction algorithm (STCA)

1976: McCreight publishes a simplified, O(m)-time STCA.

1990: Manber and Myers introduce suffix arrays, give an O(m log m)-time suffix array construction algorithm (SACA)

1995: Ukkonen invents a popular O(m)-time STCA.

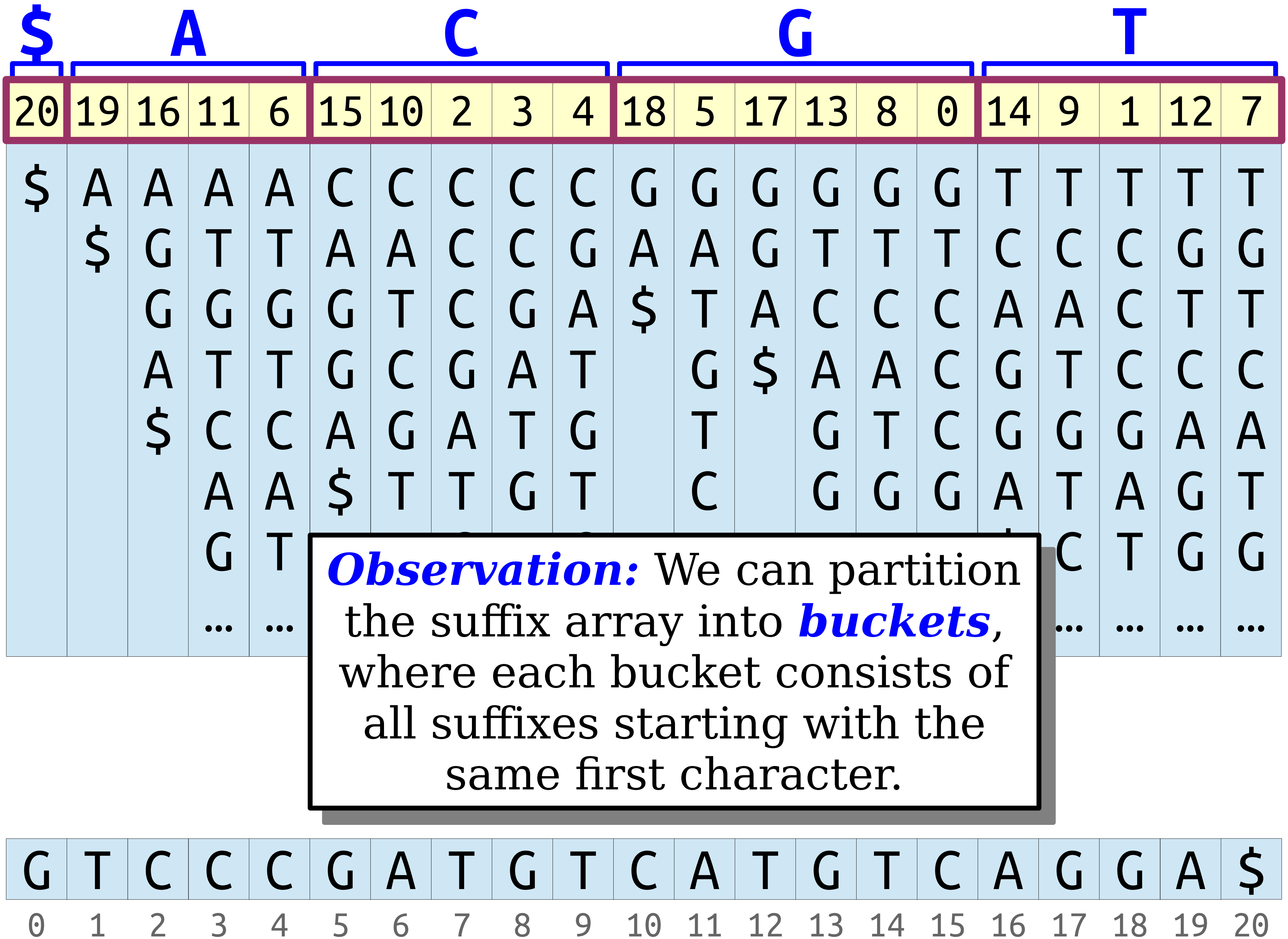
1997: Farach invents an O(m)-time STCA that works on integer input alphabets.

2002: Ko and Aluru devise an O(m)-time SACA.

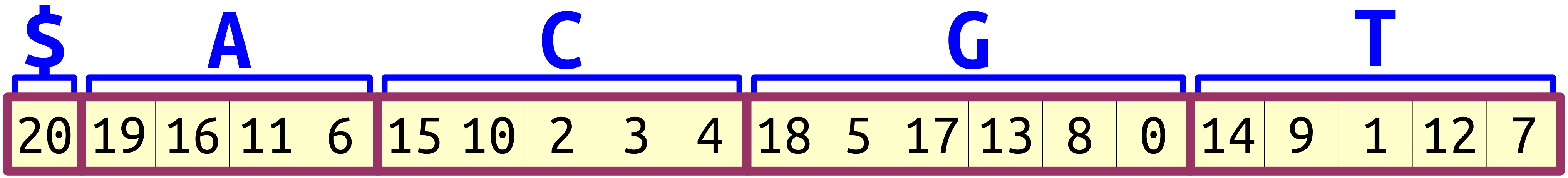
2003: Kärkkäinen et al devise an O(m)-time SACA based on Farach's insights.

2008: Nong et al build on Ko and Aluru to give a faster O(m)-time SACA called ***SA-IS***.

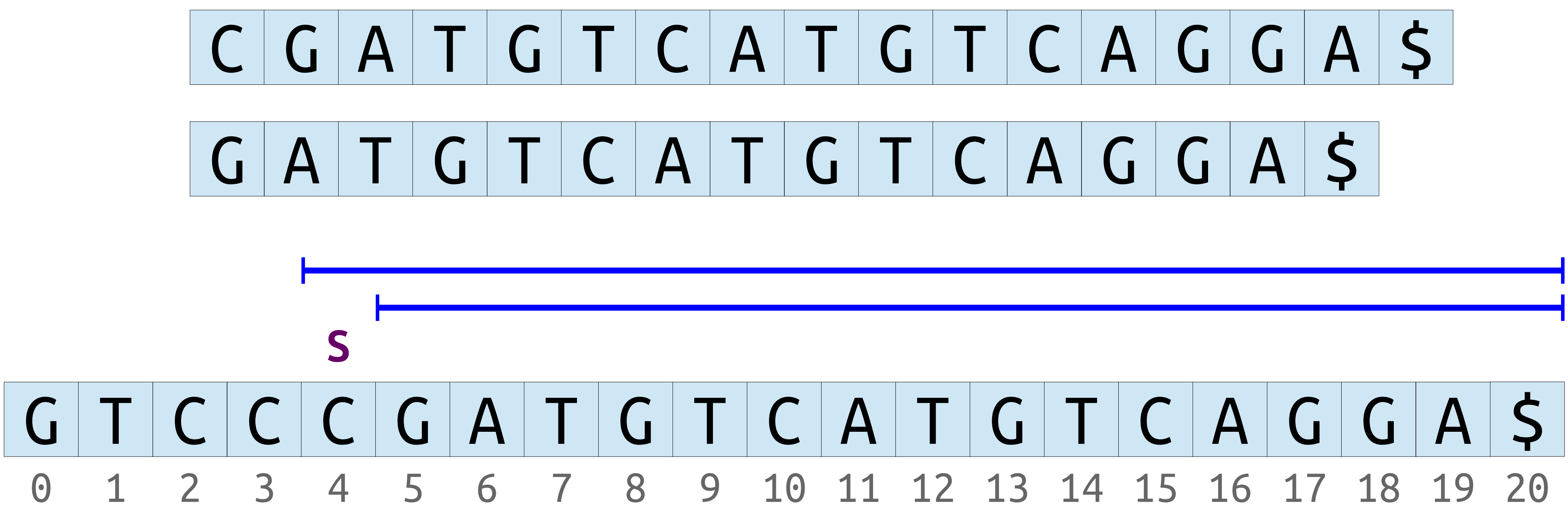
Some Observations about Suffix Arrays

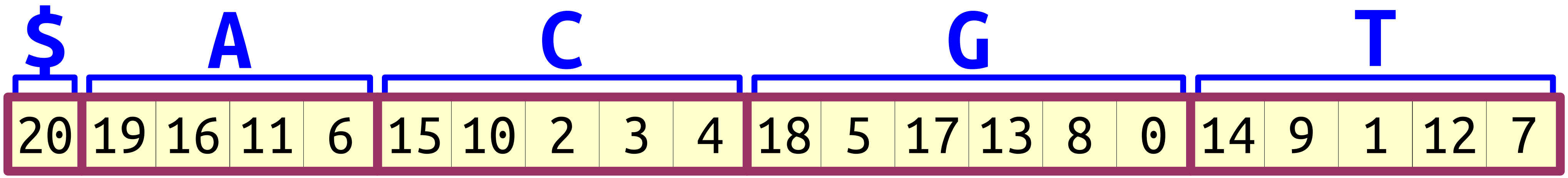


Observation: We can partition the suffix array into ***buckets***, where each bucket consists of all suffixes starting with the same first character.

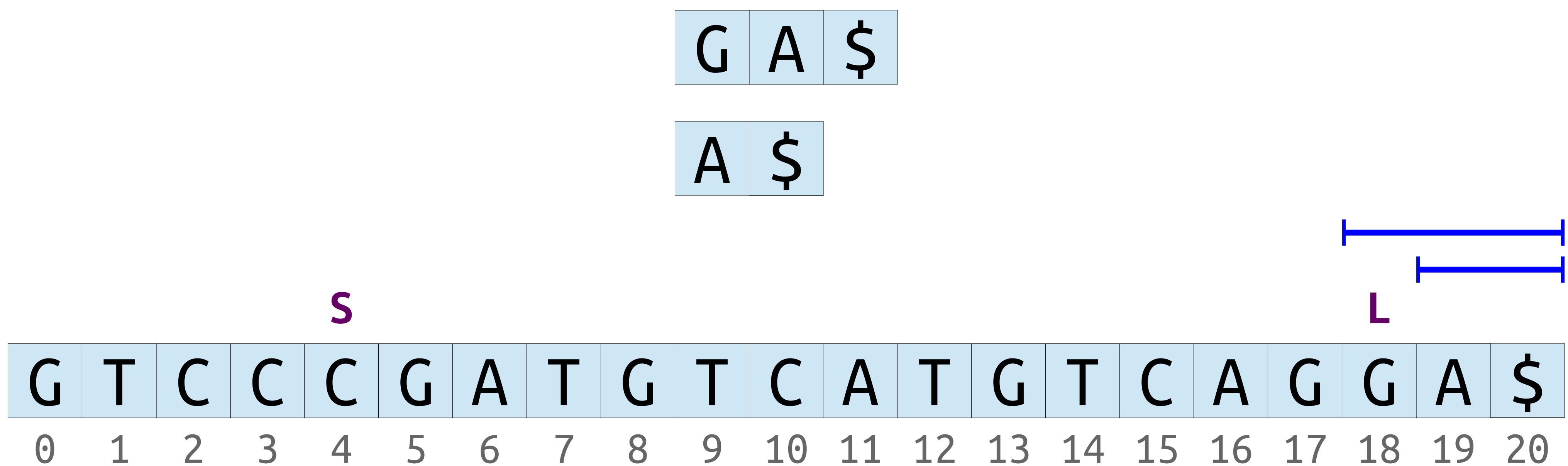


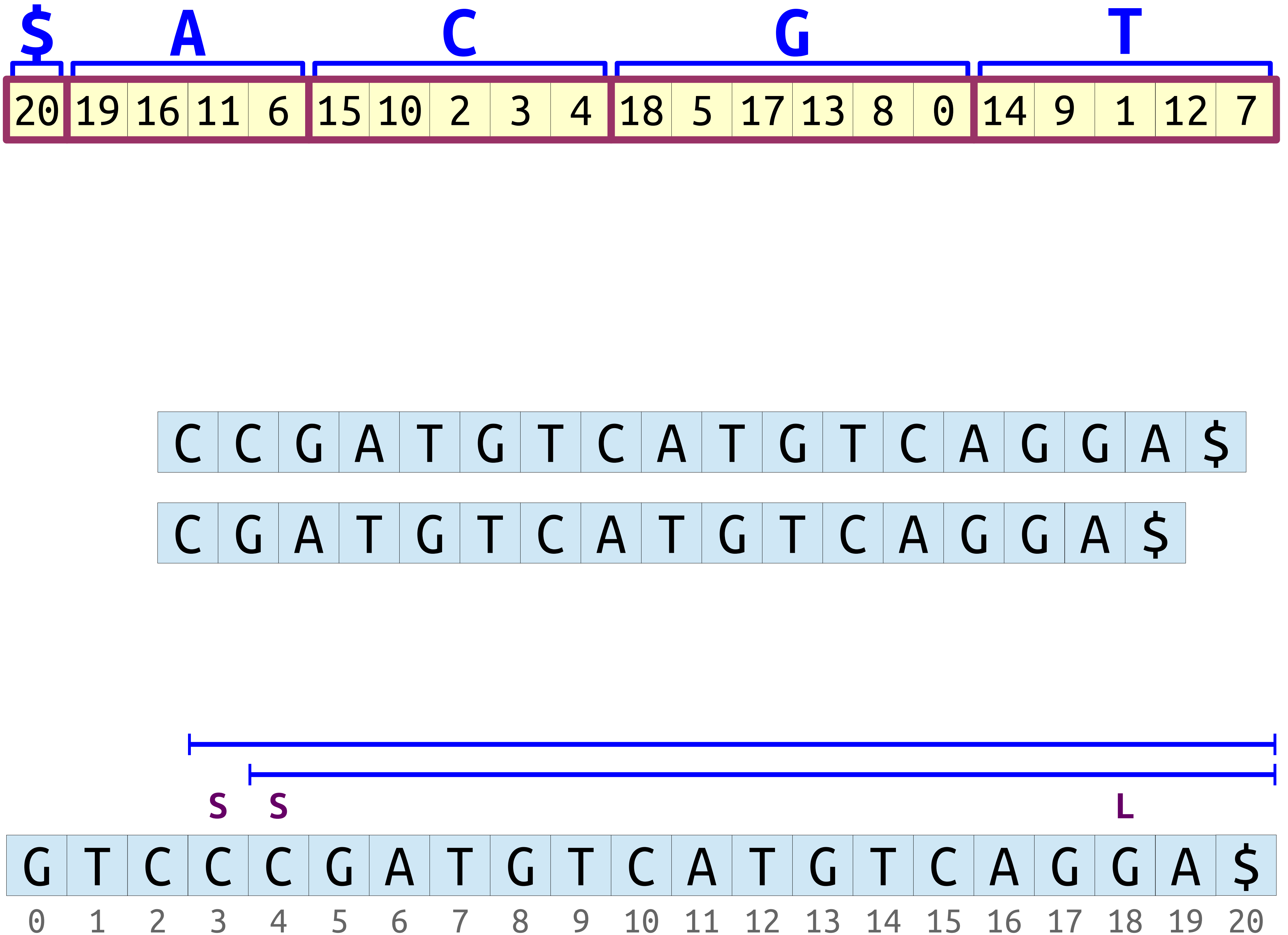
We'll call the suffix at position 4 an ***S-type*** suffix (***S*** for ***s***maller), since it lexicographically precedes the suffix at the position immediately after it.



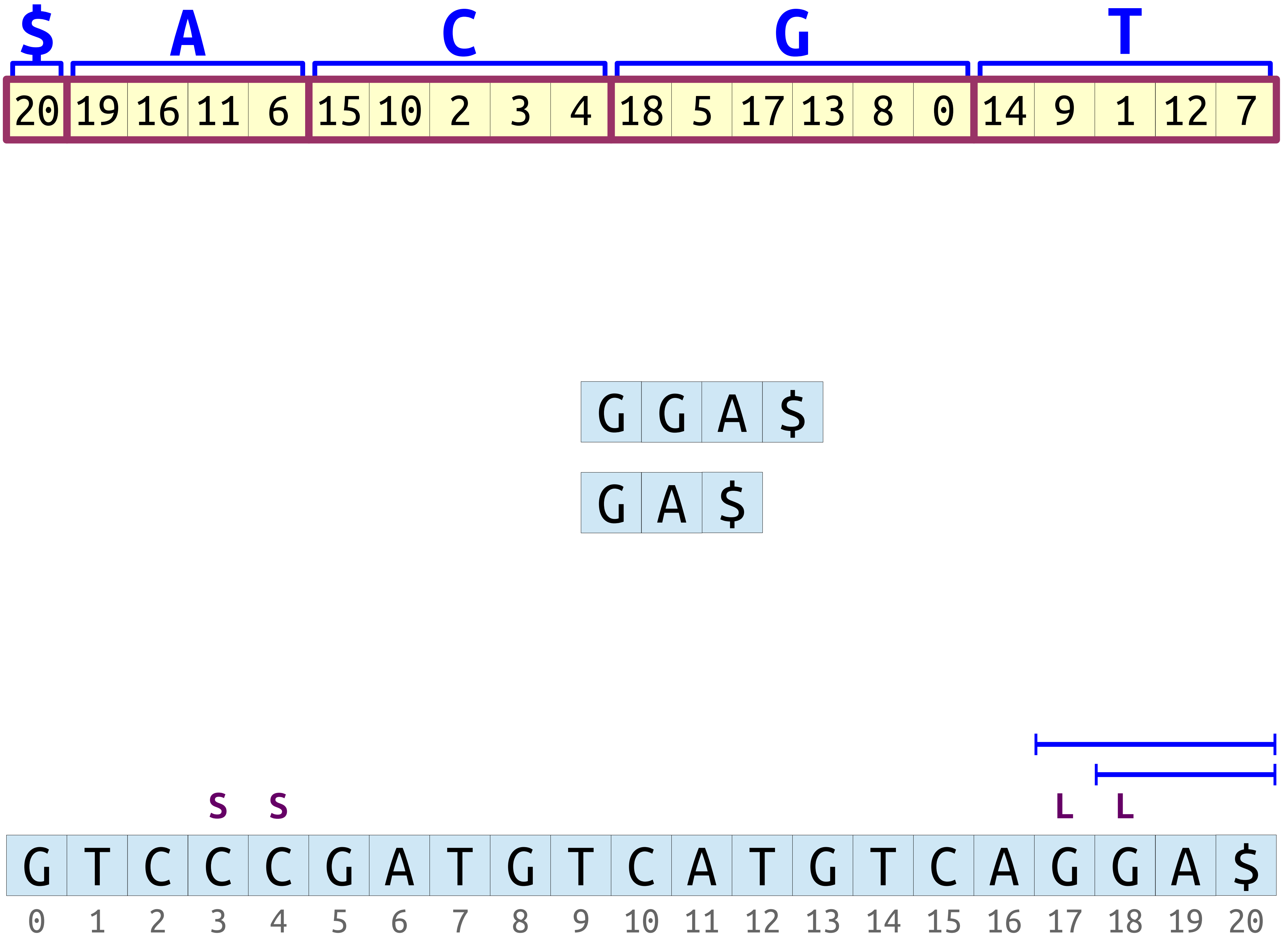


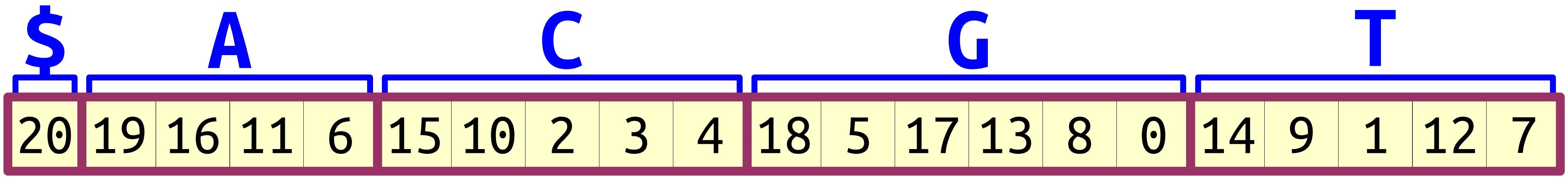
We'll call the suffix at position 18 an ***L-type*** suffix (***L*** for ***l***arger), since it lexicographically comes after the suffix at the position immediately after it.



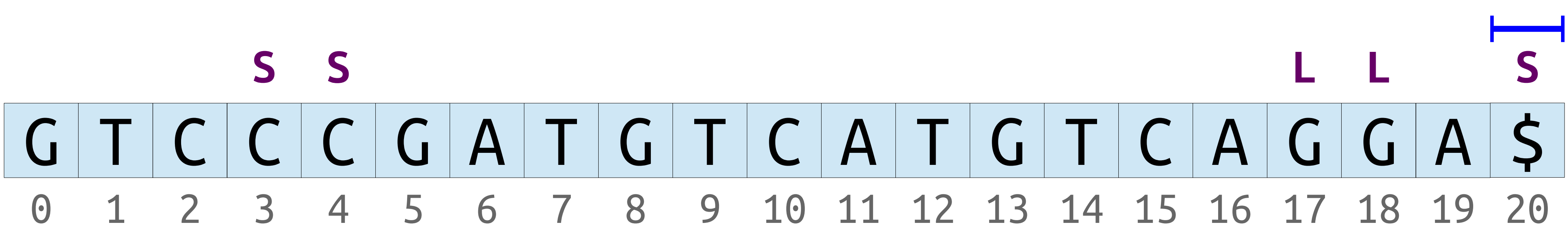


$ A C G T
20 19 16 11 6 15 10 2 3 4 18 5 17 13 8 0 14 9 1 12 7
C C G A T G T C A T G T C A G G A $
C G A T G T C A T G T C A G G A $
S S L
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


$ A C G T
20 19 16 11 6 15 10 2 3 4 18 5 17 13 8 0 14 9 1 12 7
G G A $
G A $
S S L L
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



By definition, the suffix starting at the sentinel is considered an *S*-type suffix.



$	A				C					G						T				
20	19	16	11	6	15	10	2	3	4	18	5	17	13	8	0	14	9	1	12	7

Theorem: A suffix starting at position k is an S-type suffix if

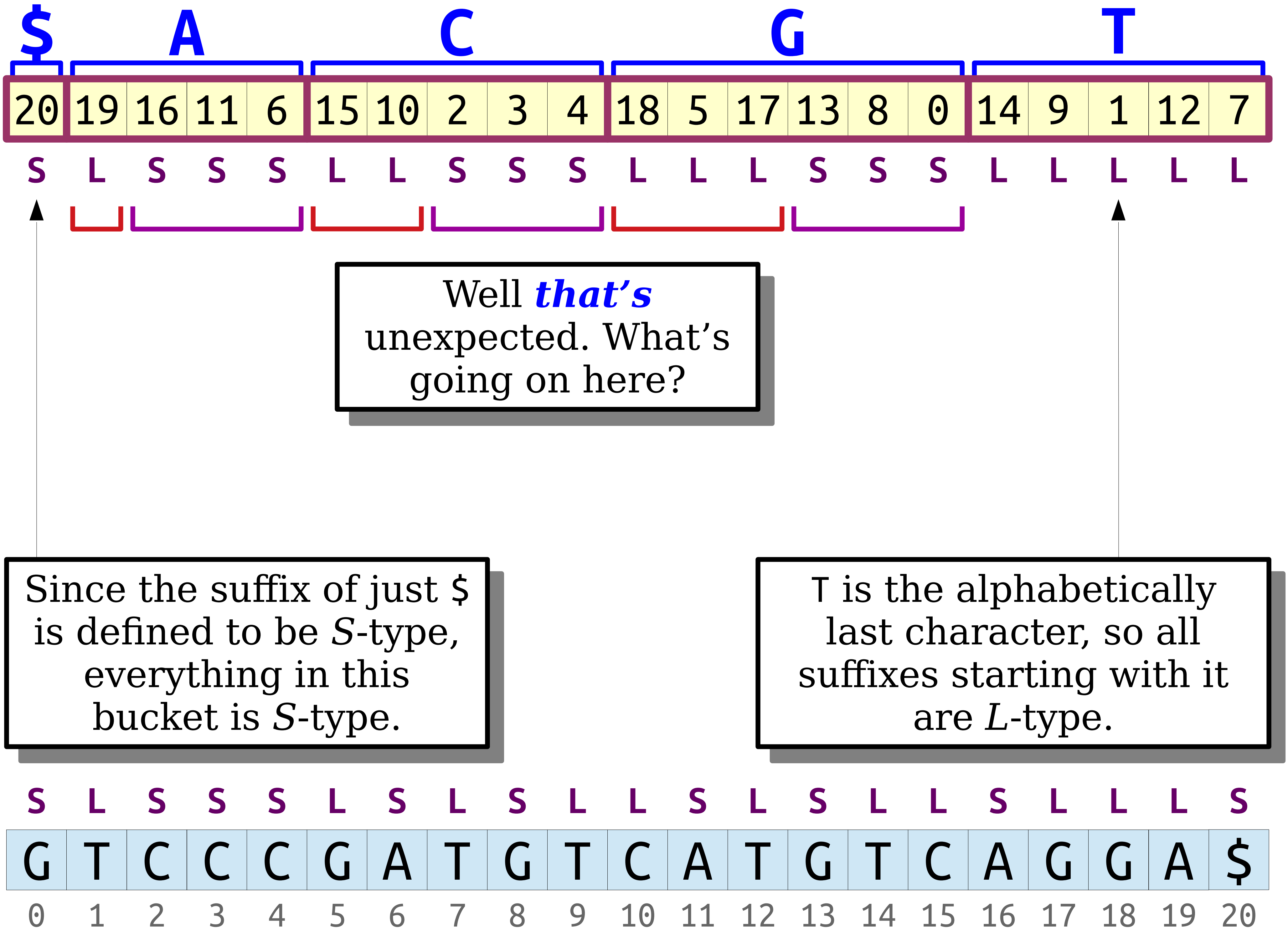
- $Text[k] < Text[k+1]$, or
- $Text[k] = Text[k+1]$ and the suffix at index $k+1$ is S-type, or
- $Text[k] = \$$.

A suffix starting at position k is a L-type suffix if

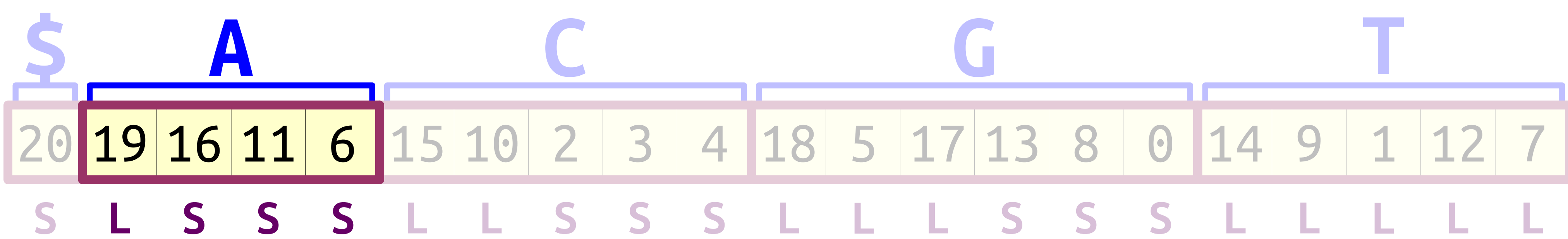
- $Text[k] > Text[k+1]$, or
- $Text[k] = Text[k+1]$ and the suffix at position $k+1$ is L-type.

We can tag each suffix as S-type or L-type in time $O(m)$ by scanning *Text* from right-to-left and applying the above rules.

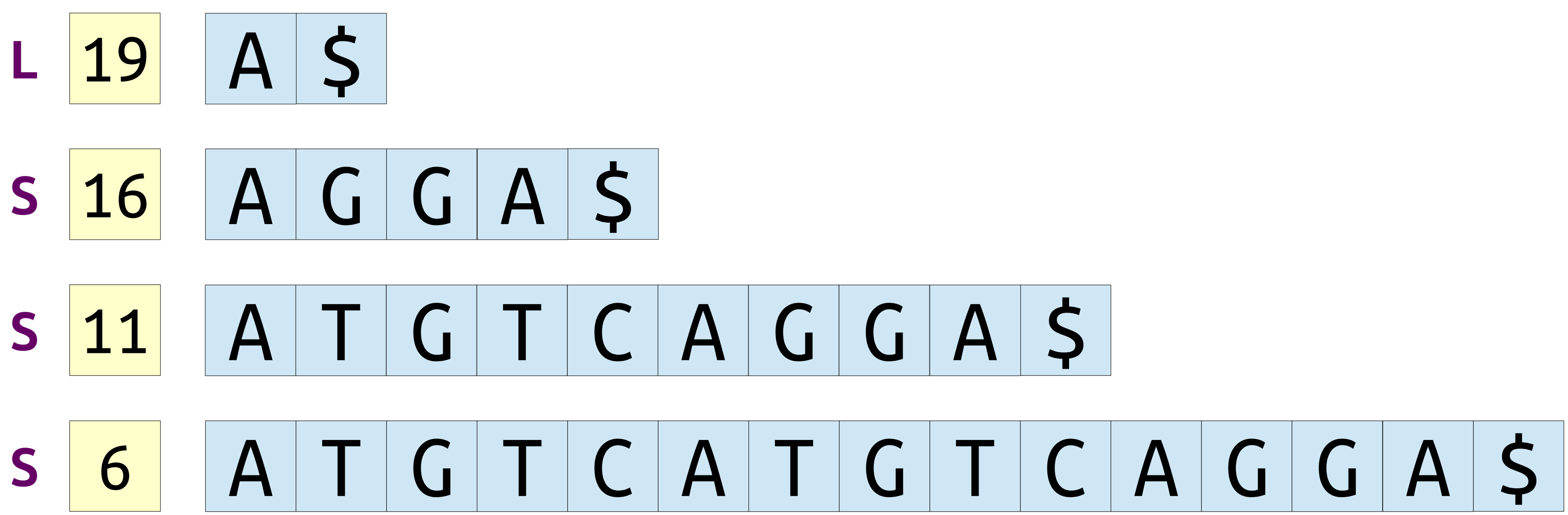
S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

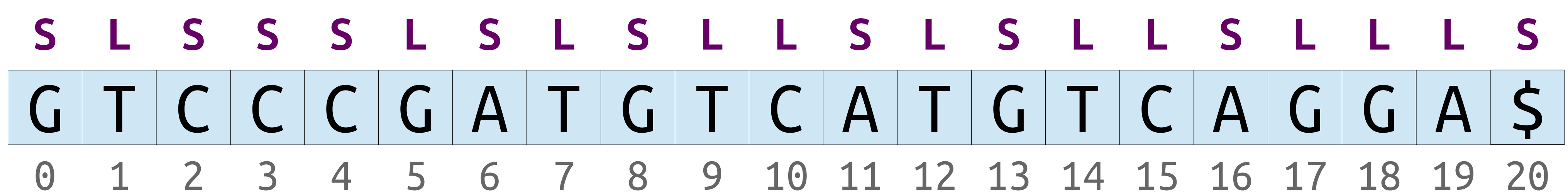


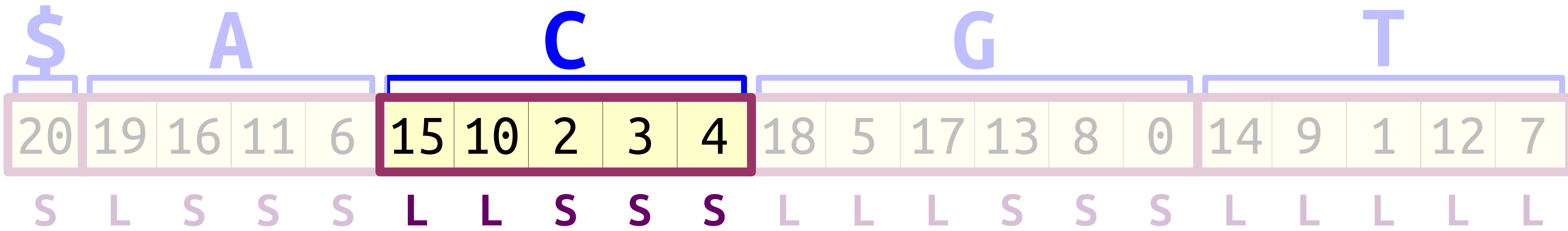
$ A C G T
20 19 16 11 6 15 10 2 3 4 18 5 17 13 8 0 14 9 1 12 7
S L S S S L L S S S L L L S S S L L L L L
Well ***that's*** unexpected. What's going on here?
Since the suffix of just $ is defined to be *S*-type, everything in this bucket is *S*-type.
T is the alphabetically last character, so all suffixes starting with it are *L*-type.
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



Theorem: Let i and j be indices of two suffixes that start with the same character. Then if i is L-type and j is S-type, the suffix beginning at position i lexicographically precedes the suffix beginning at position j.



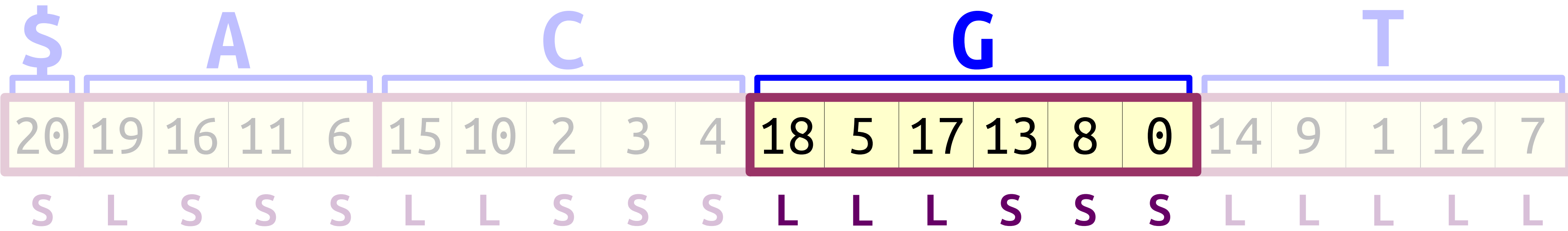




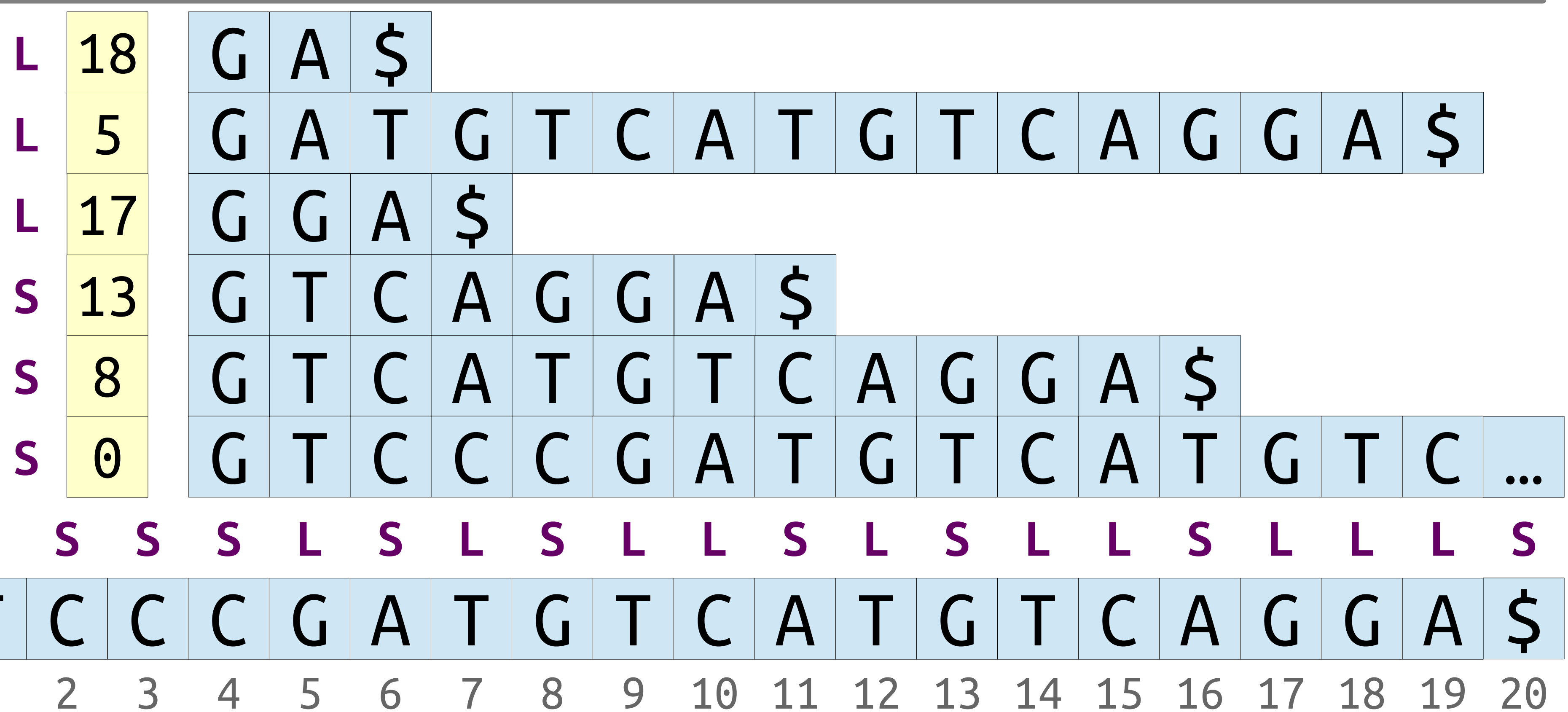
Theorem: Let i and j be indices of two suffixes that start with the same character. Then if i is L-type and j is S-type, the suffix beginning at position i lexicographically precedes the suffix beginning at position j.

L	15	C	A	G	G	A	$										
L	10	C	A	T	G	T	C	A	G	G	A	$					
S	2	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	…
S	3	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	…
S	4	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	…

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



Theorem: Let i and j be indices of two suffixes that start with the same character. Then if i is L-type and j is S-type, the suffix beginning at position i lexicographically precedes the suffix beginning at position j.



Where We Stand

- We can efficiently classify each suffix as either S-type or L-type in time $O(m)$.
- We know a good amount about the relative positioning of the suffixes:
  - All suffixes are bucketed by their first character.
  - All L-type suffixes come before all S-type suffixes.
- If we can get everything relatively positioned within its group, we're done!

SA-IS at a Glance

- There are three core insights that collectively give us the SA-IS algorithm.
- First:

There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

- Second:

Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

- Third:

With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.

SA-IS at a Glance

There are three core insights that collectively give us the SA-IS algorithm.

- First:

There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

Second:

Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

Third:

With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.

These suffixes are called ***LMS suffixes*** (***L***eft***M***ost ***S***-type).

A suffix is an LMS suffix if it's S-type and the suffix before it is L-type.

This suffix isn't an LMS suffix because it isn't preceded by a suffix at all!

This suffix isn't LMS because the suffix before it isn't L-type.

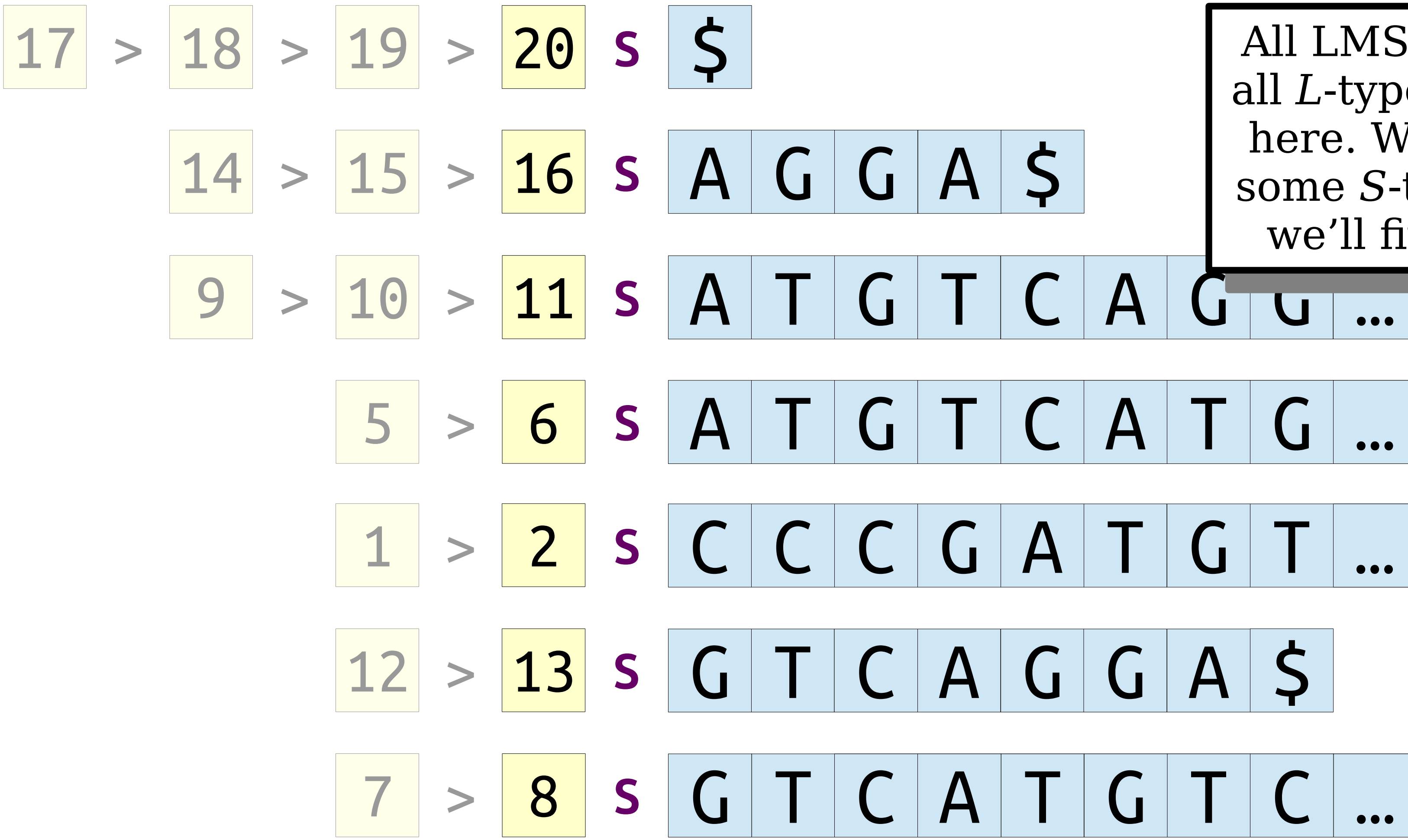
The sentinel by itself is always considered an LMS suffix.

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Key Theorem: If we can get the LMS suffixes – and just the LMS suffixes – in sorted order, then we can, in time O(m), get all the other suffixes in order as well.

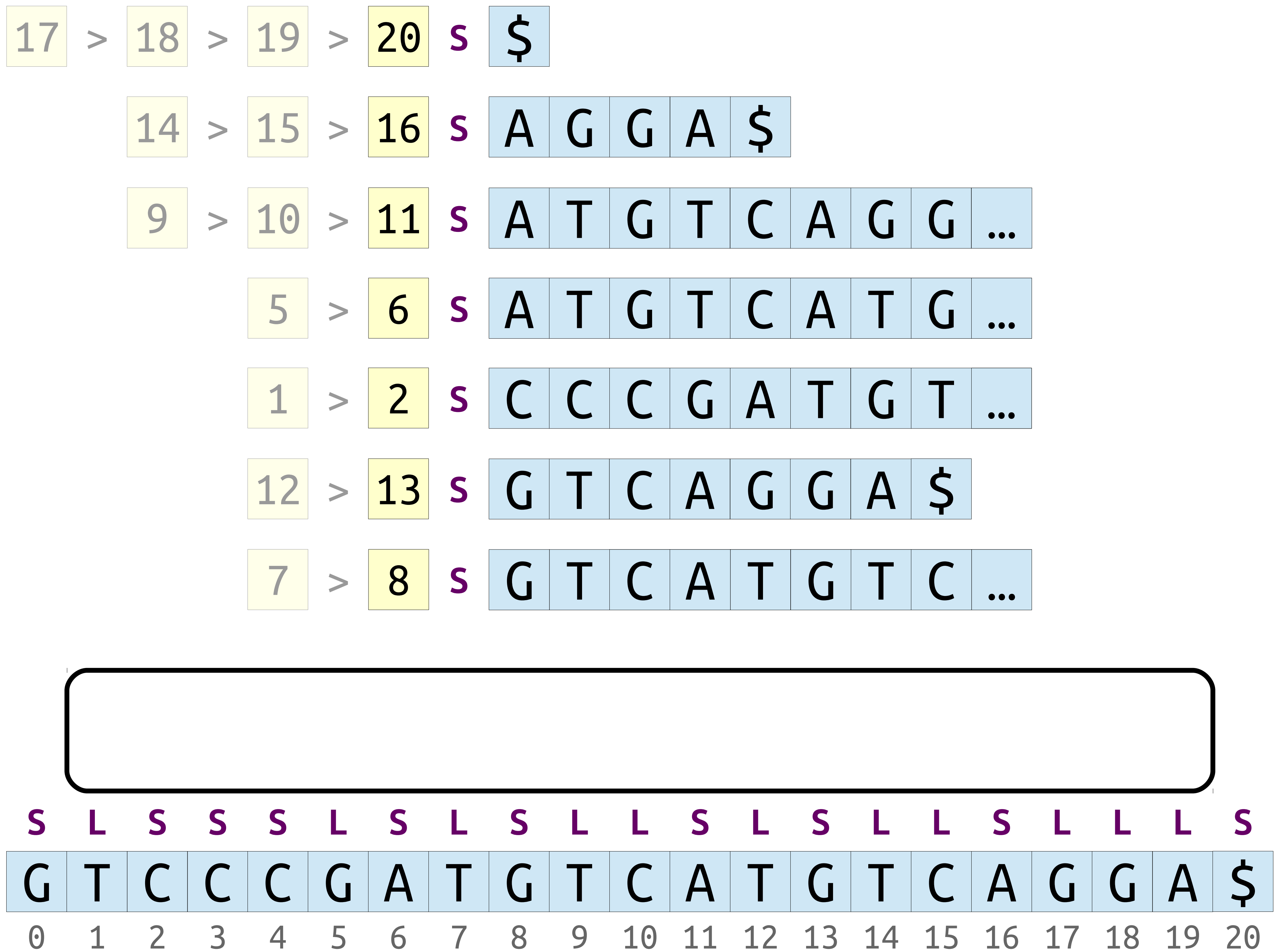
The algorithm for doing this is called ***induced sorting***. This is the “IS” in SA-IS.

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

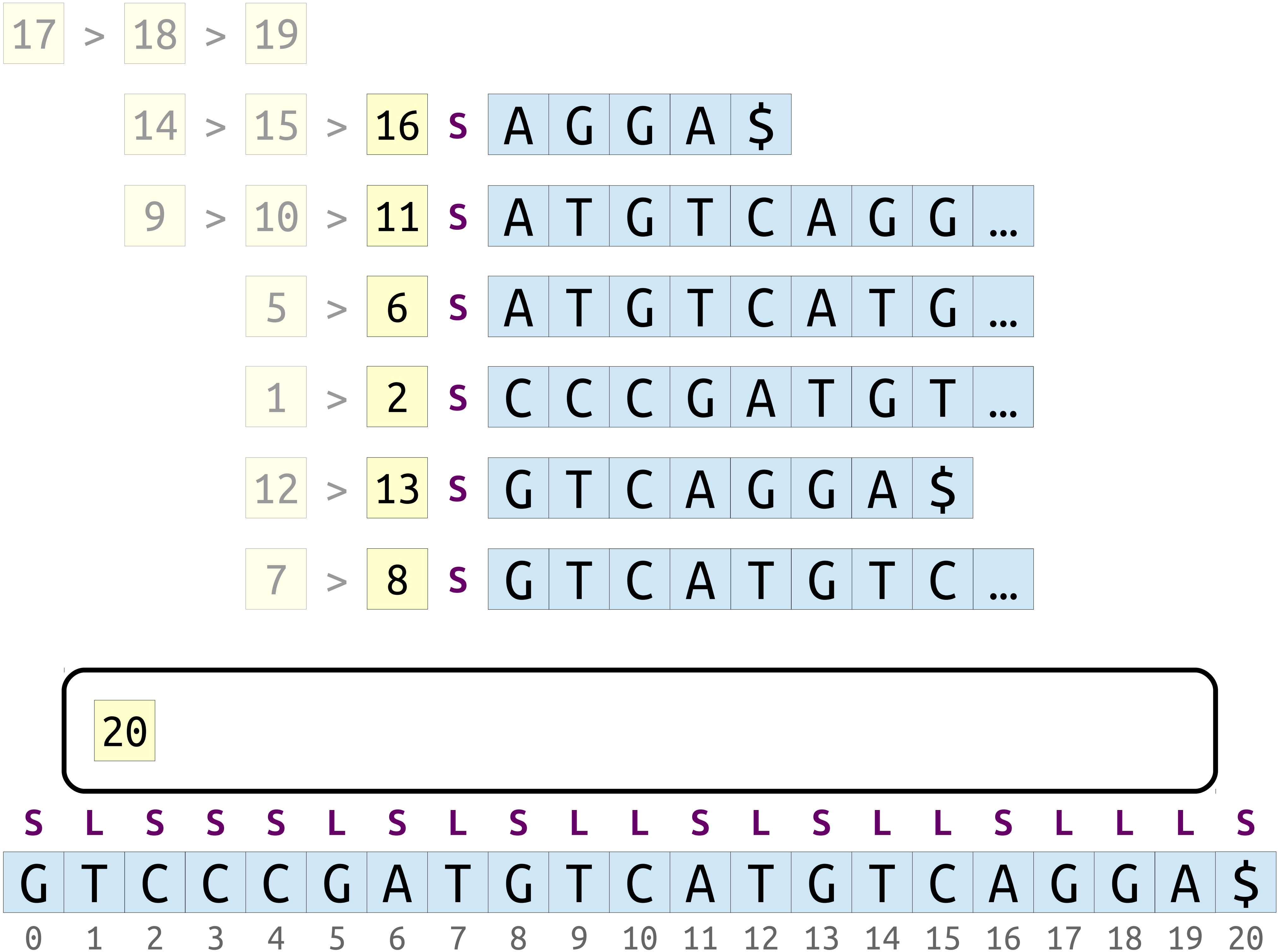


All LMS suffixes and all *L*-type suffixes are here. We're missing some *S*-type suffixes; we'll fix that later.

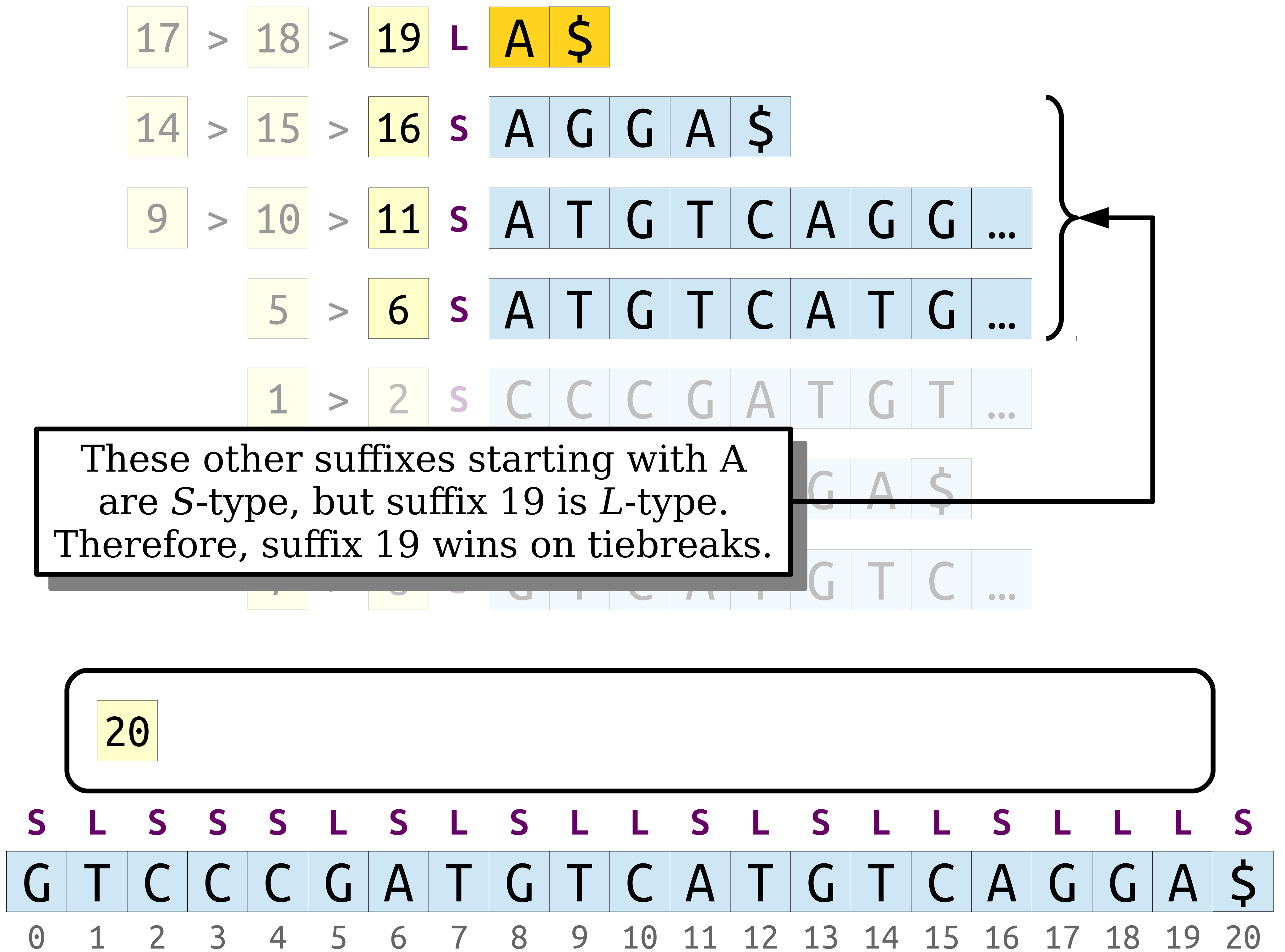
This is a multiway merge! Each list is sorted, and we want to unify them all together.



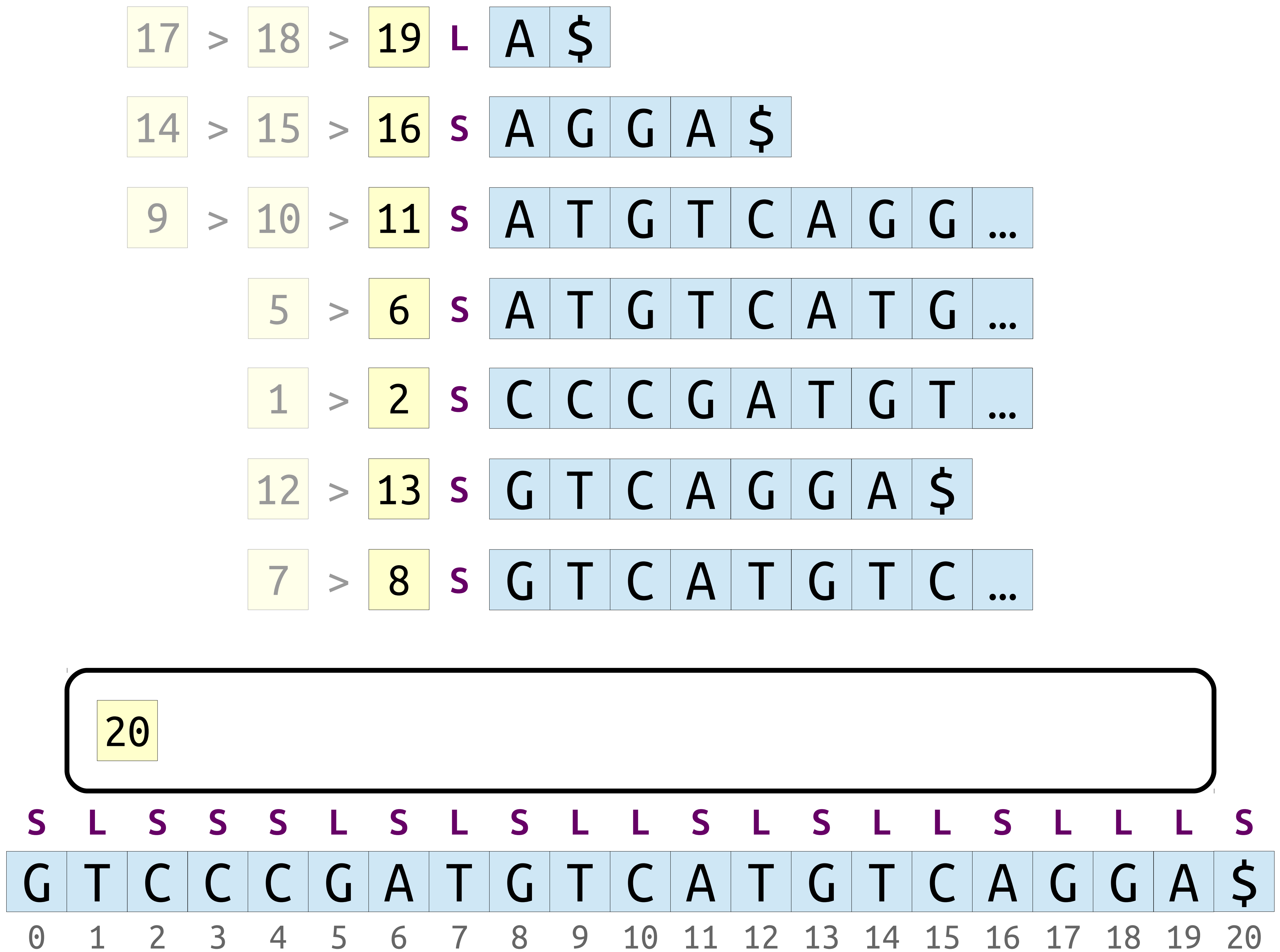
17 > 18 > 19 > 20 S $
14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G …
5 > 6 S A T G T C A T G …
1 > 2 S C C C G A T G T …
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C …
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



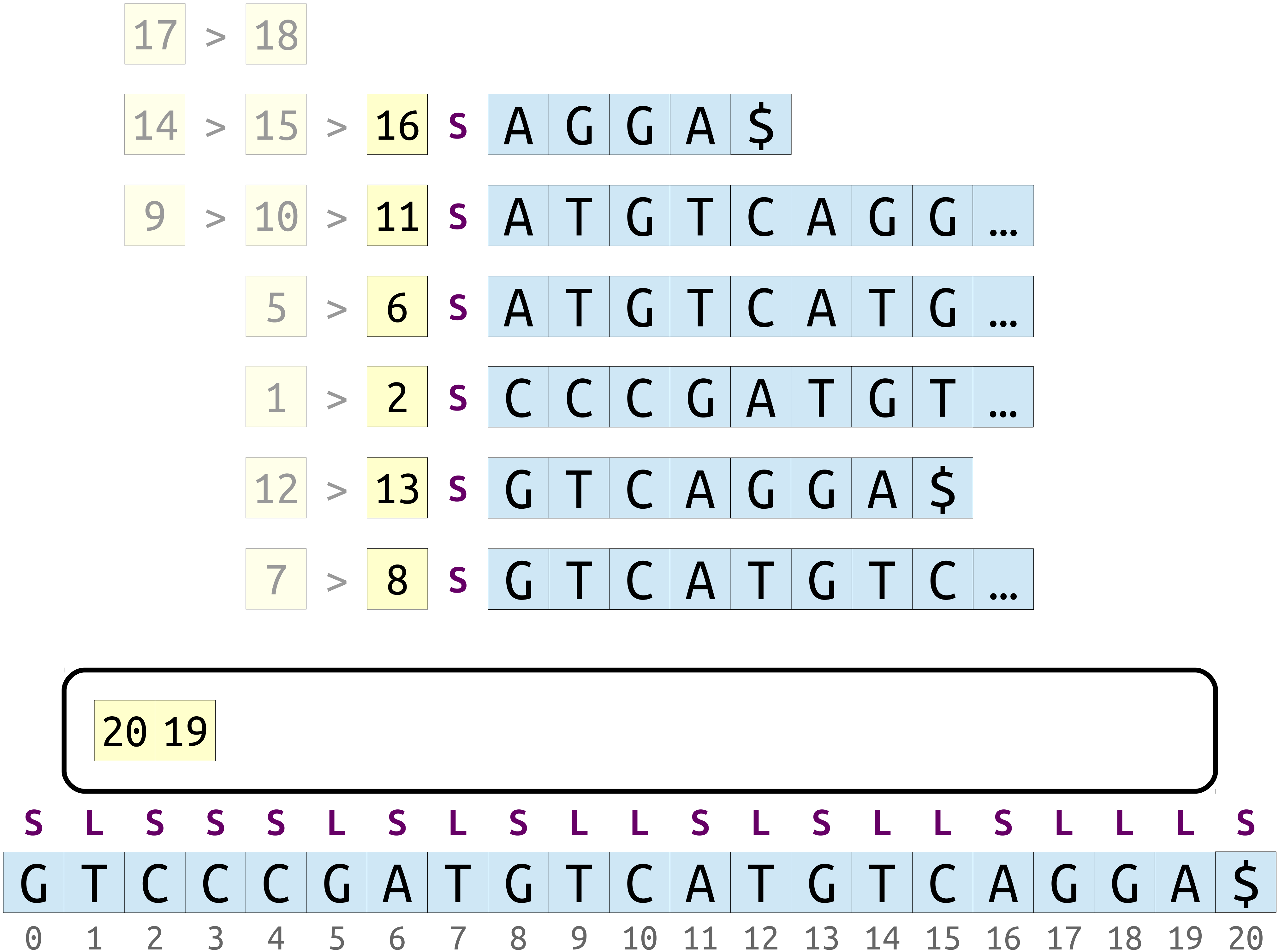
17 > 18 > 19
14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



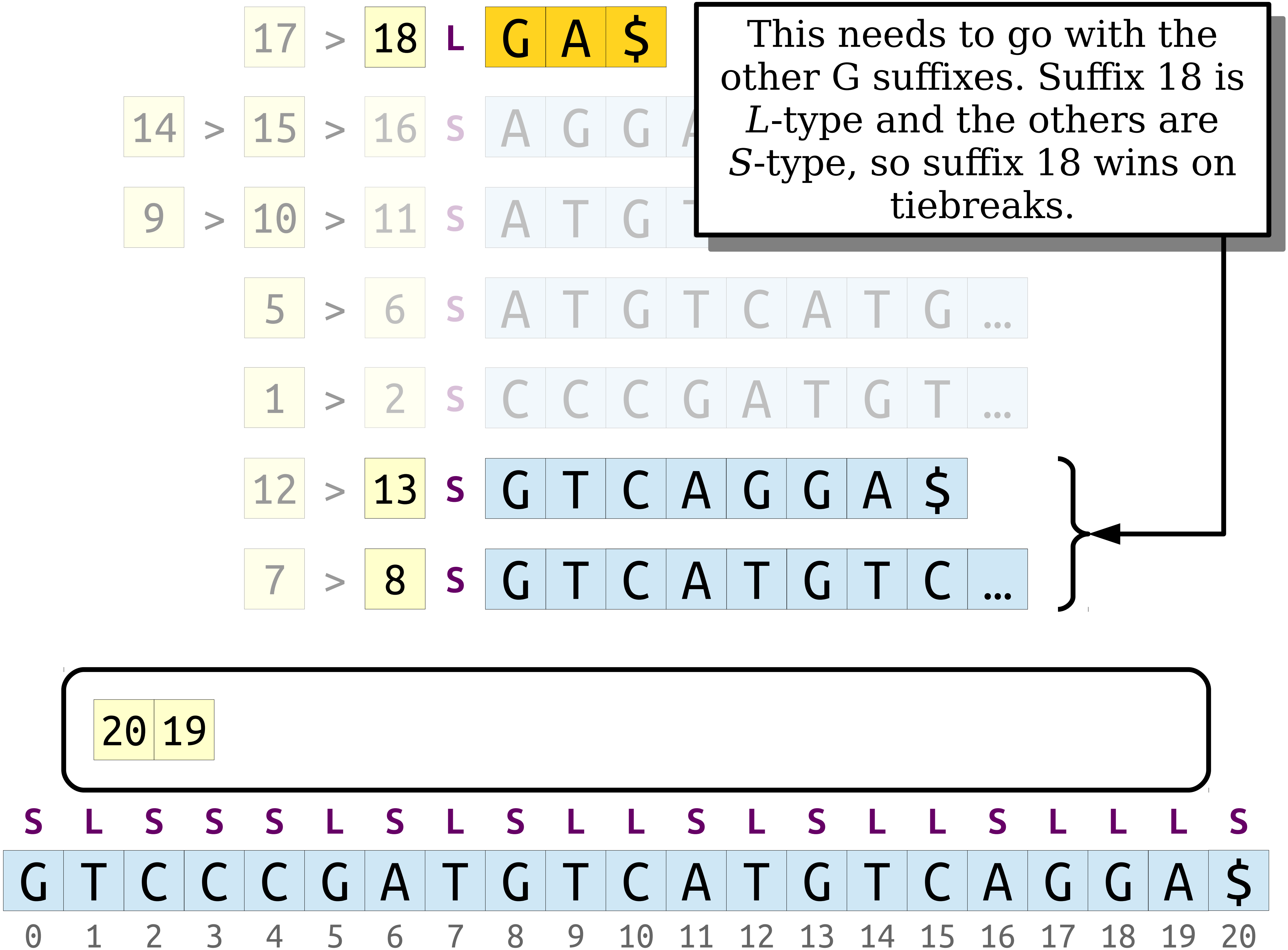
17 > 18 > 19 L A $
14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
These other suffixes starting with A are S-type, but suffix 19 is L-type. Therefore, suffix 19 wins on tiebreaks.
20
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

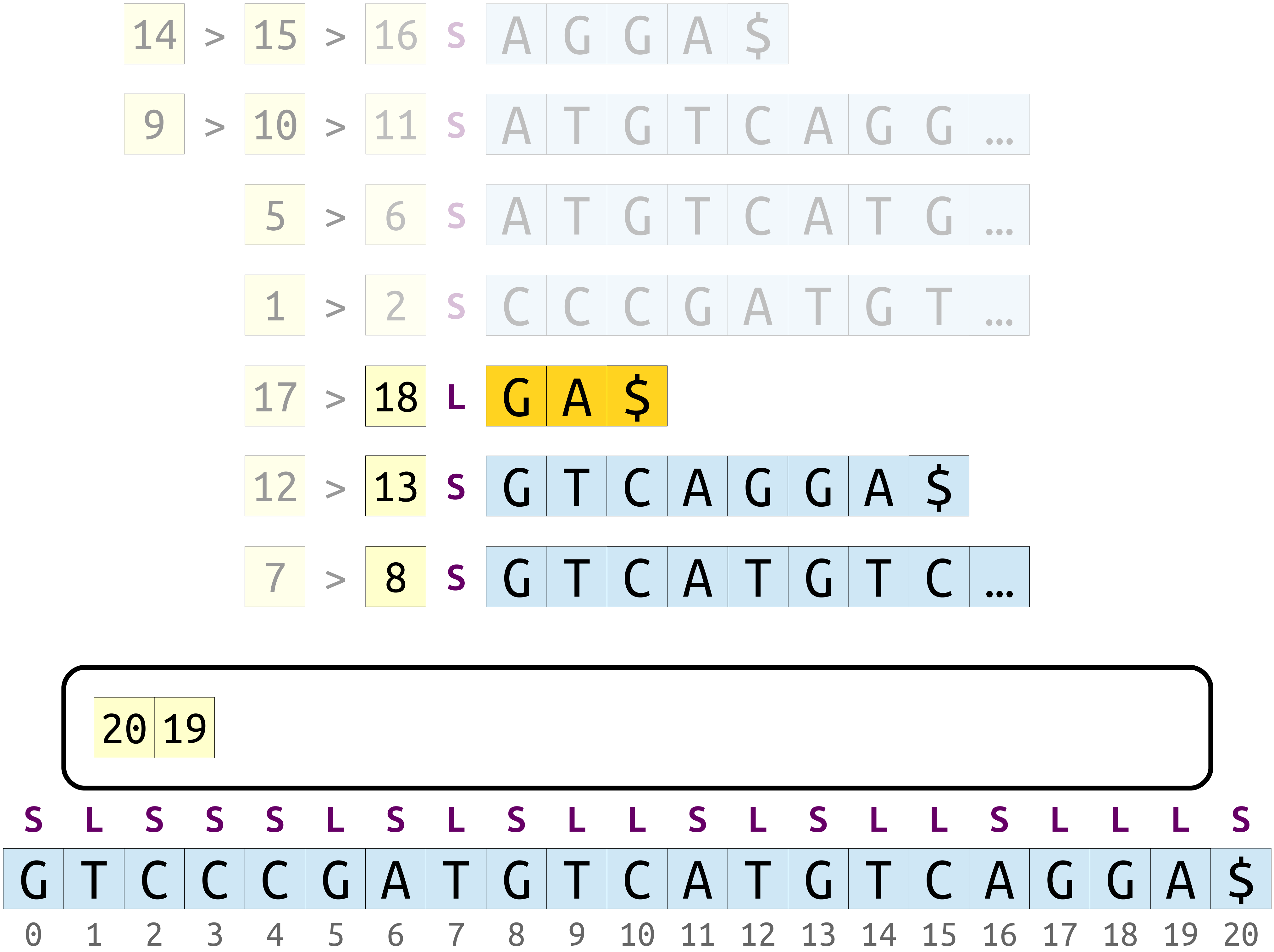


17 > 18 > 19 L A $
14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

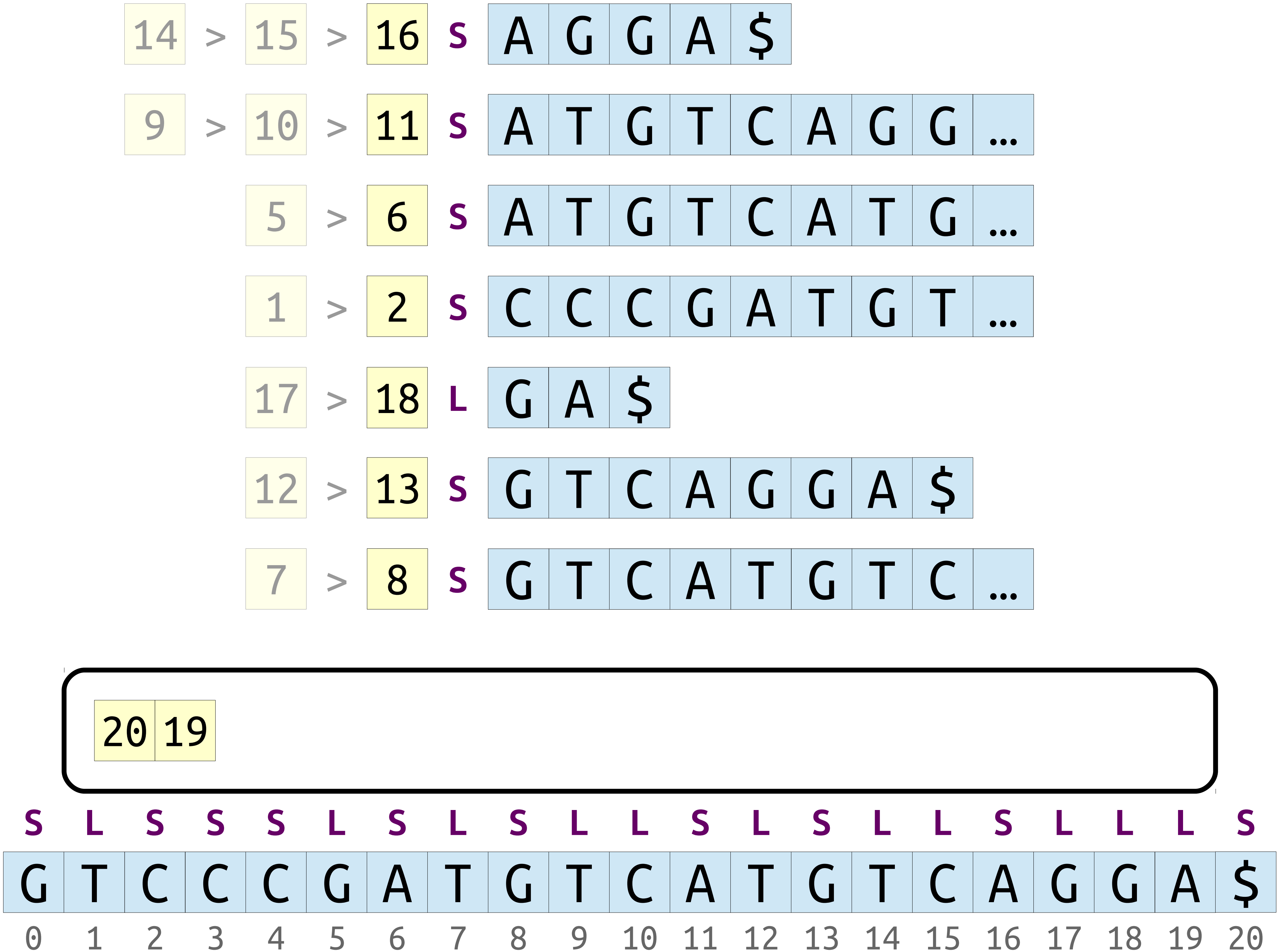


17 > 18
14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G …
5 > 6 S A T G T C A T G …
1 > 2 S C C C G A T G T …
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C …
20 19
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

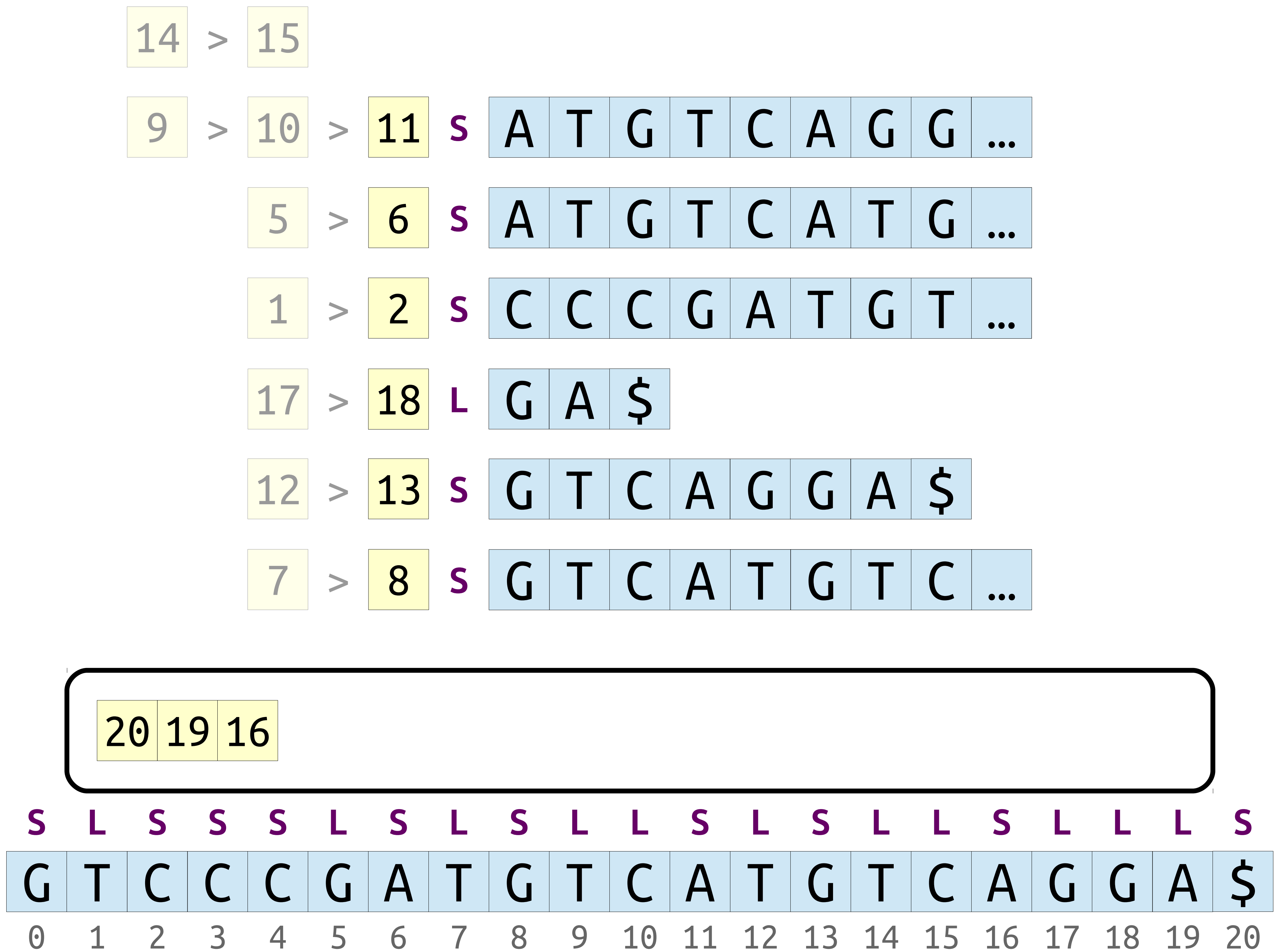

This needs to go with the other G suffixes. Suffix 18 is L-type and the others are S-type, so suffix 18 wins on tiebreaks.
17 > 18 L G A $
14 > 15 > 16 S A G G A
9 > 10 > 11 S A T G T
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

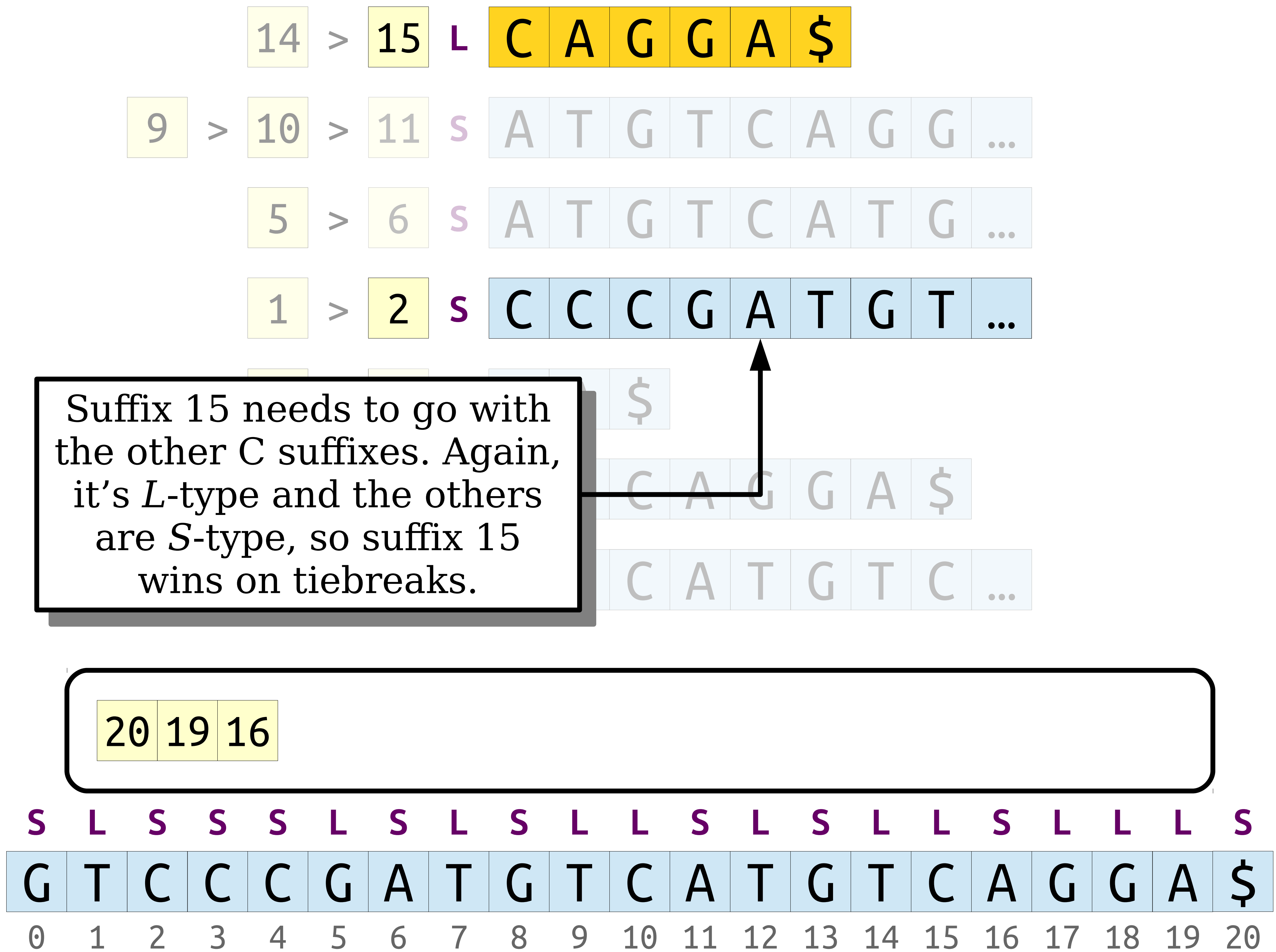


14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

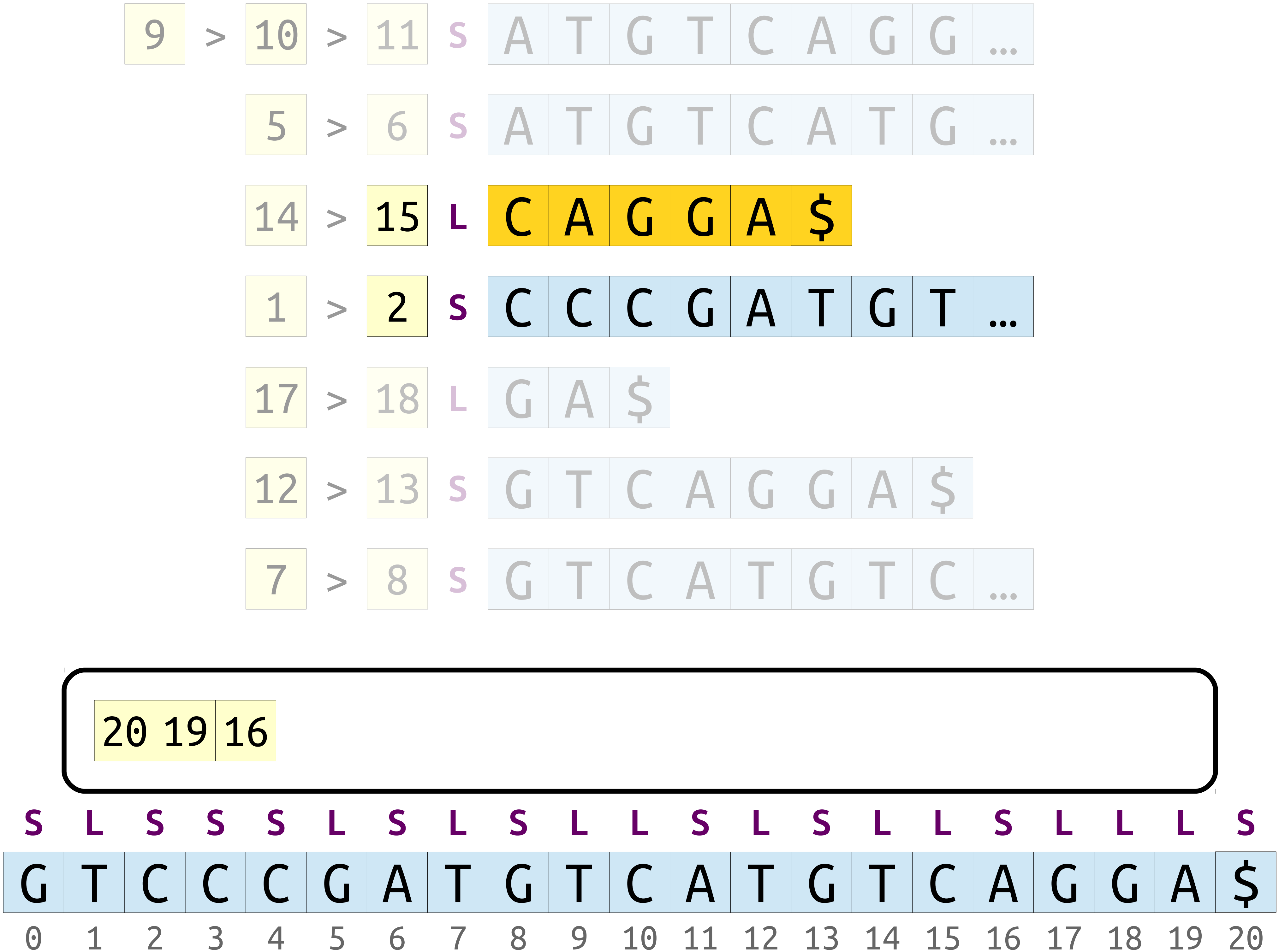


14 > 15 > 16 S A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

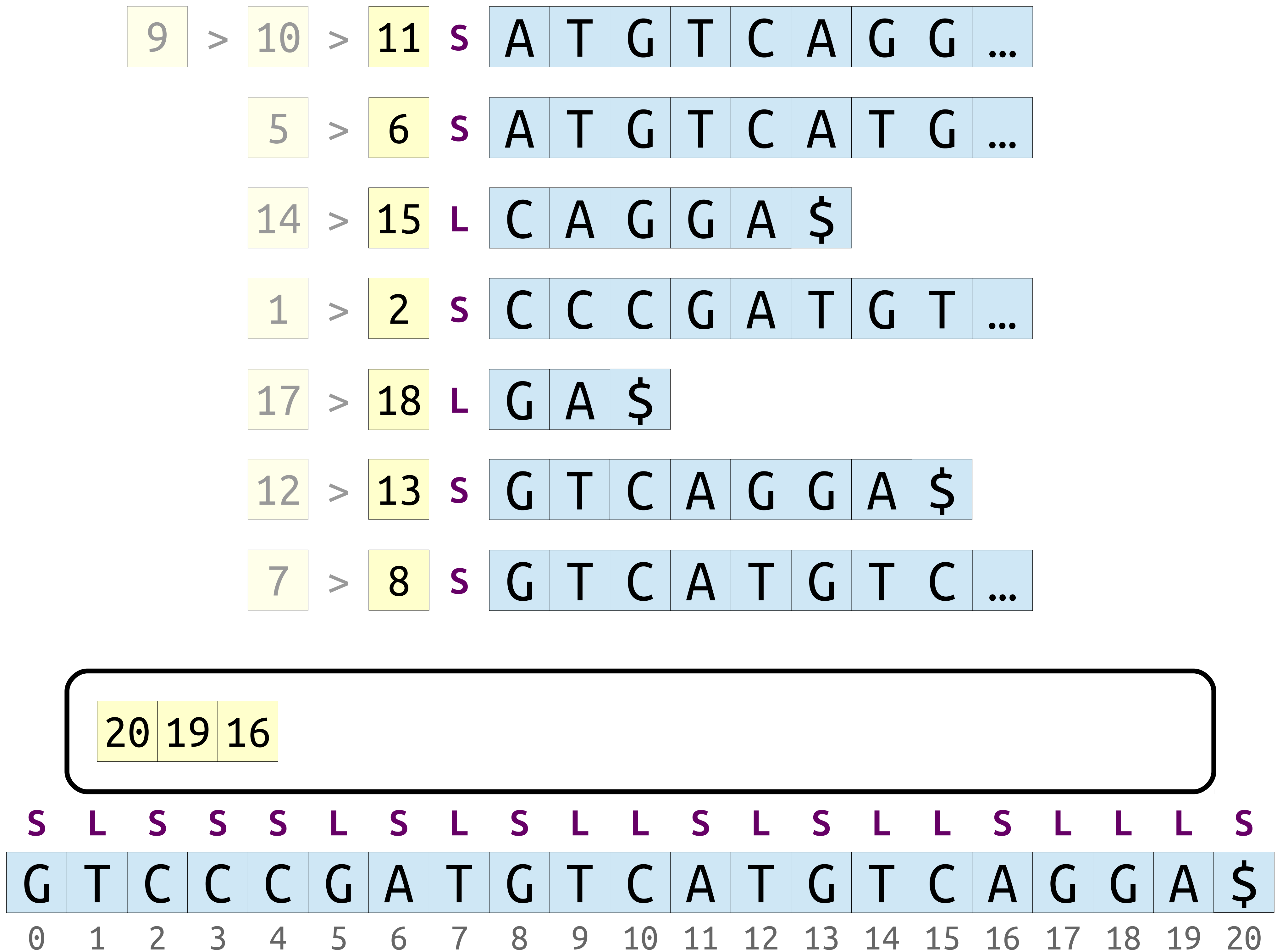

14 > 15
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

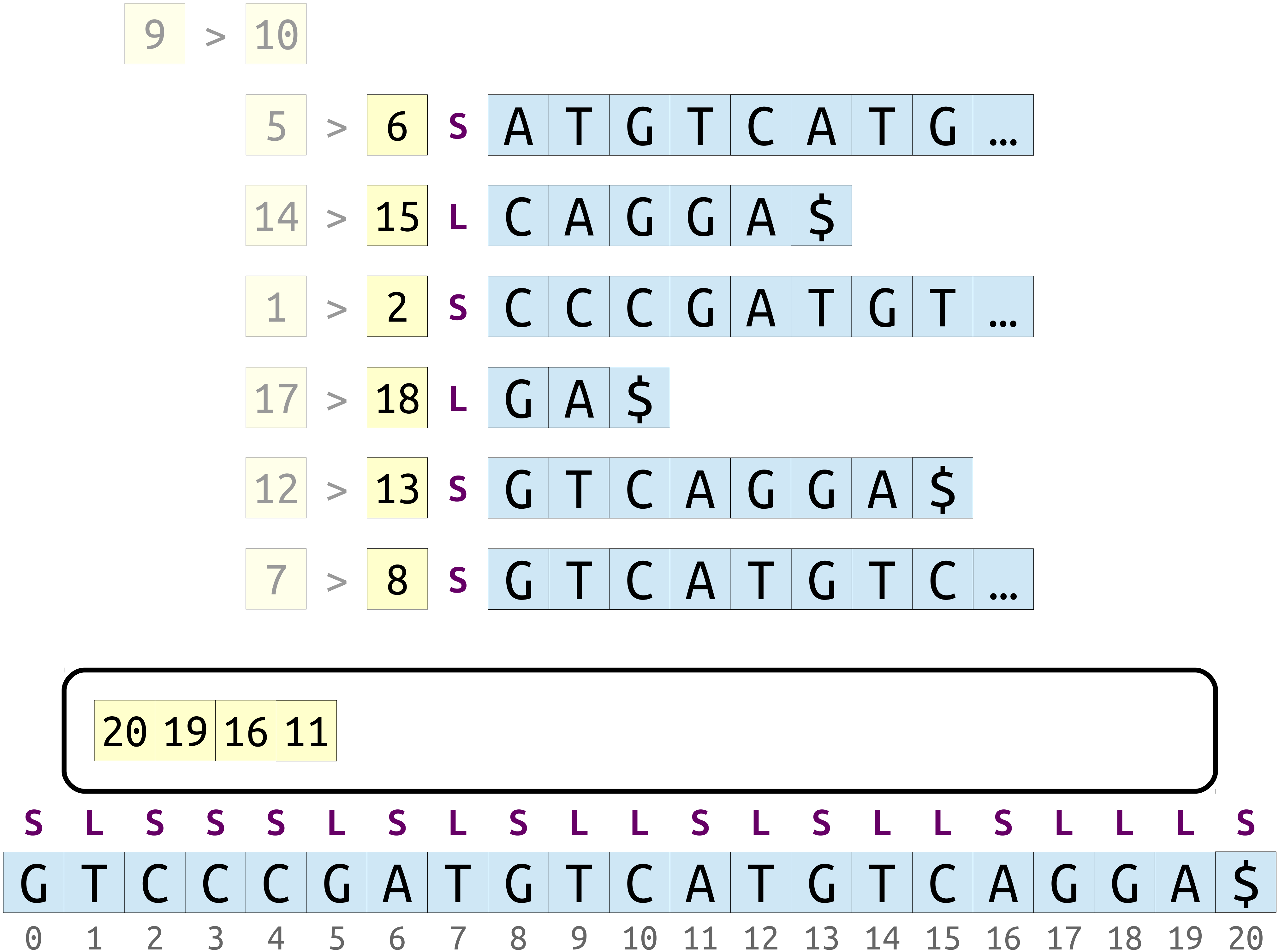


14 > 15 L C A G G A $
9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
1 > 2 S C C C G A T G T ...
$
C A G G A $
C A T G T C ...
Suffix 15 needs to go with the other C suffixes. Again, it's L-type and the others are S-type, so suffix 15 wins on tiebreaks.
20 19 16
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

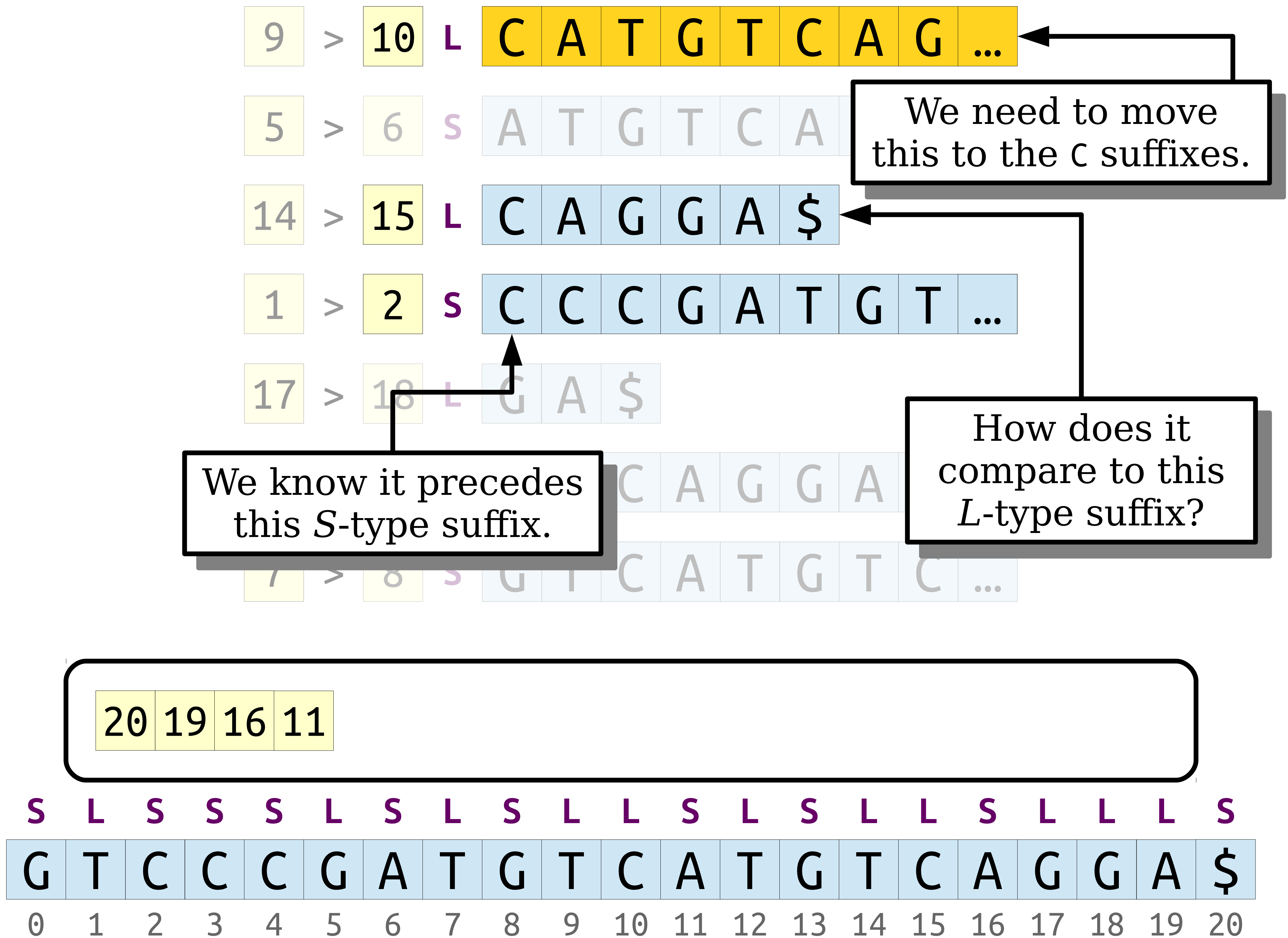


9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

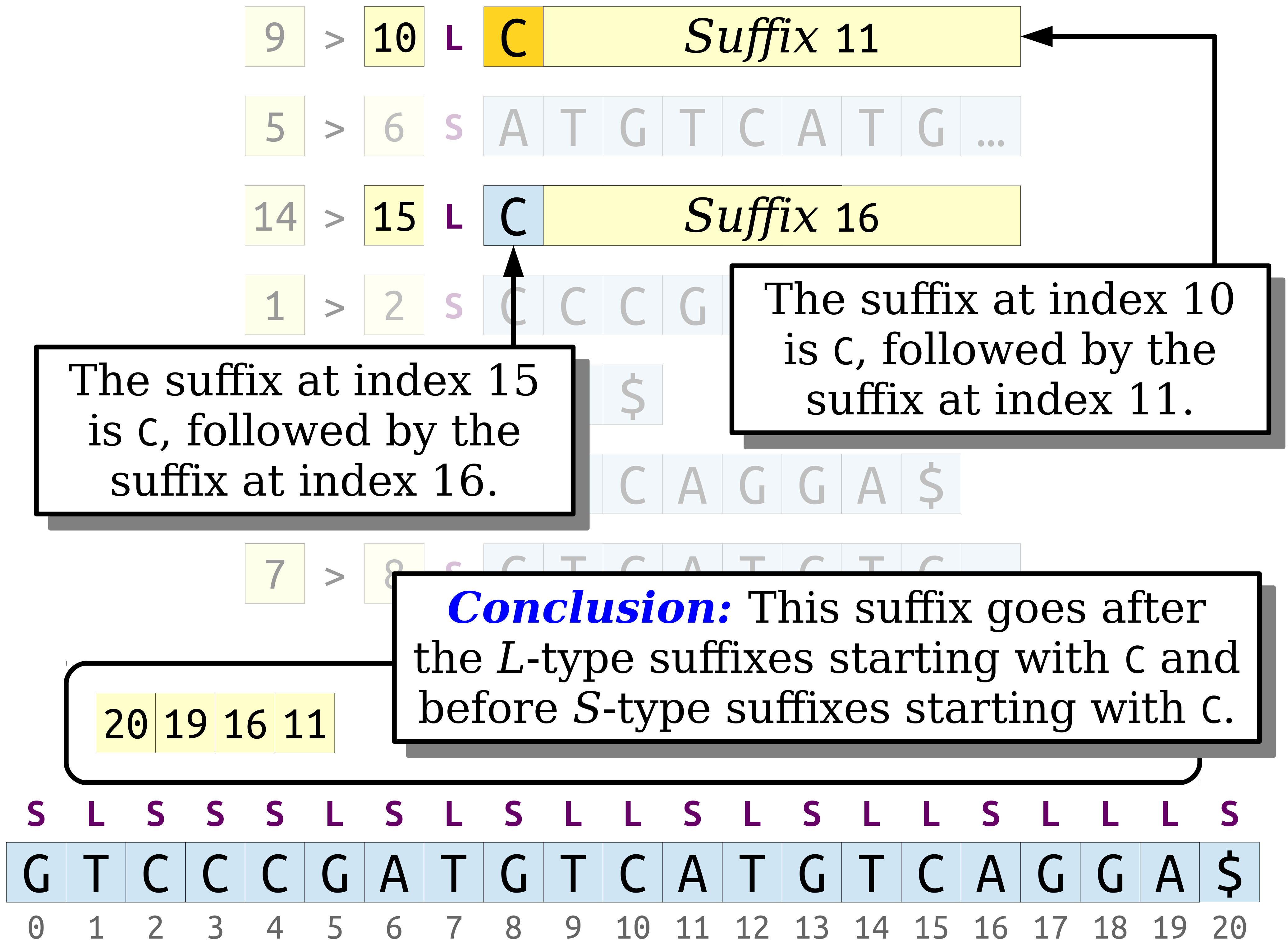

9 > 10 > 11 S A T G T C A G G ...
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



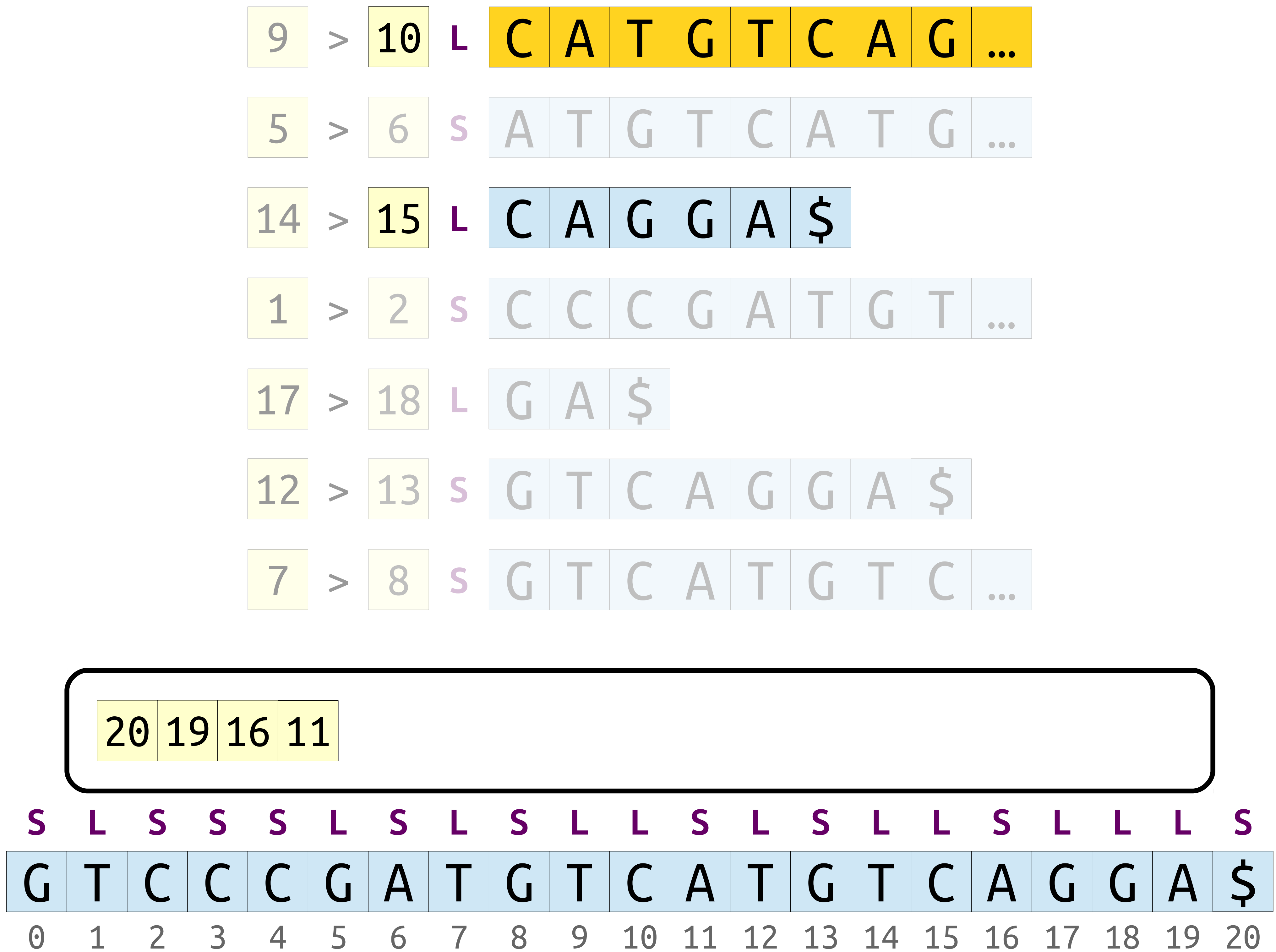
9 > 10
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

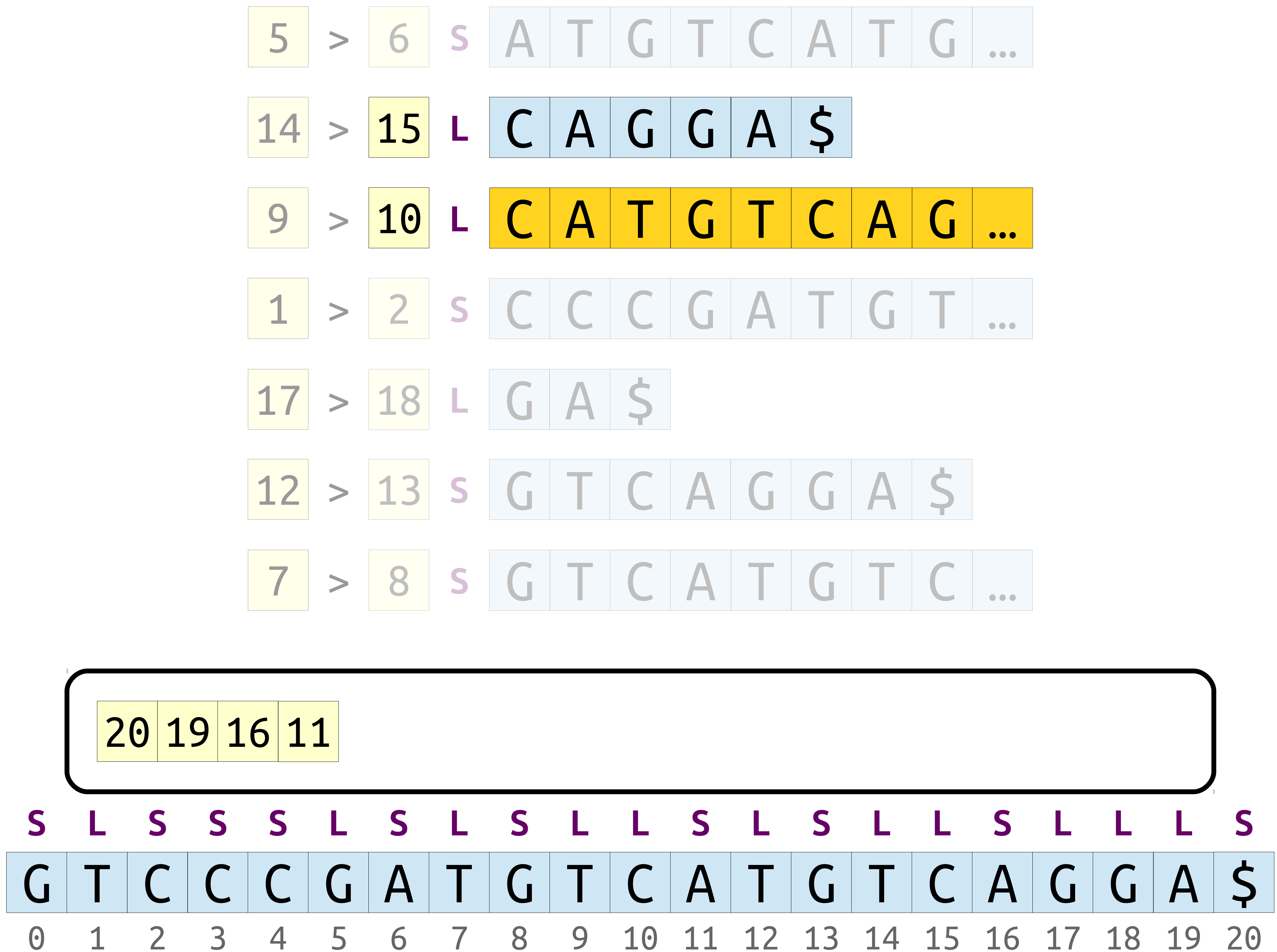


We need to move this to the C suffixes.
How does it compare to this L-type suffix?
We know it precedes this S-type suffix.
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

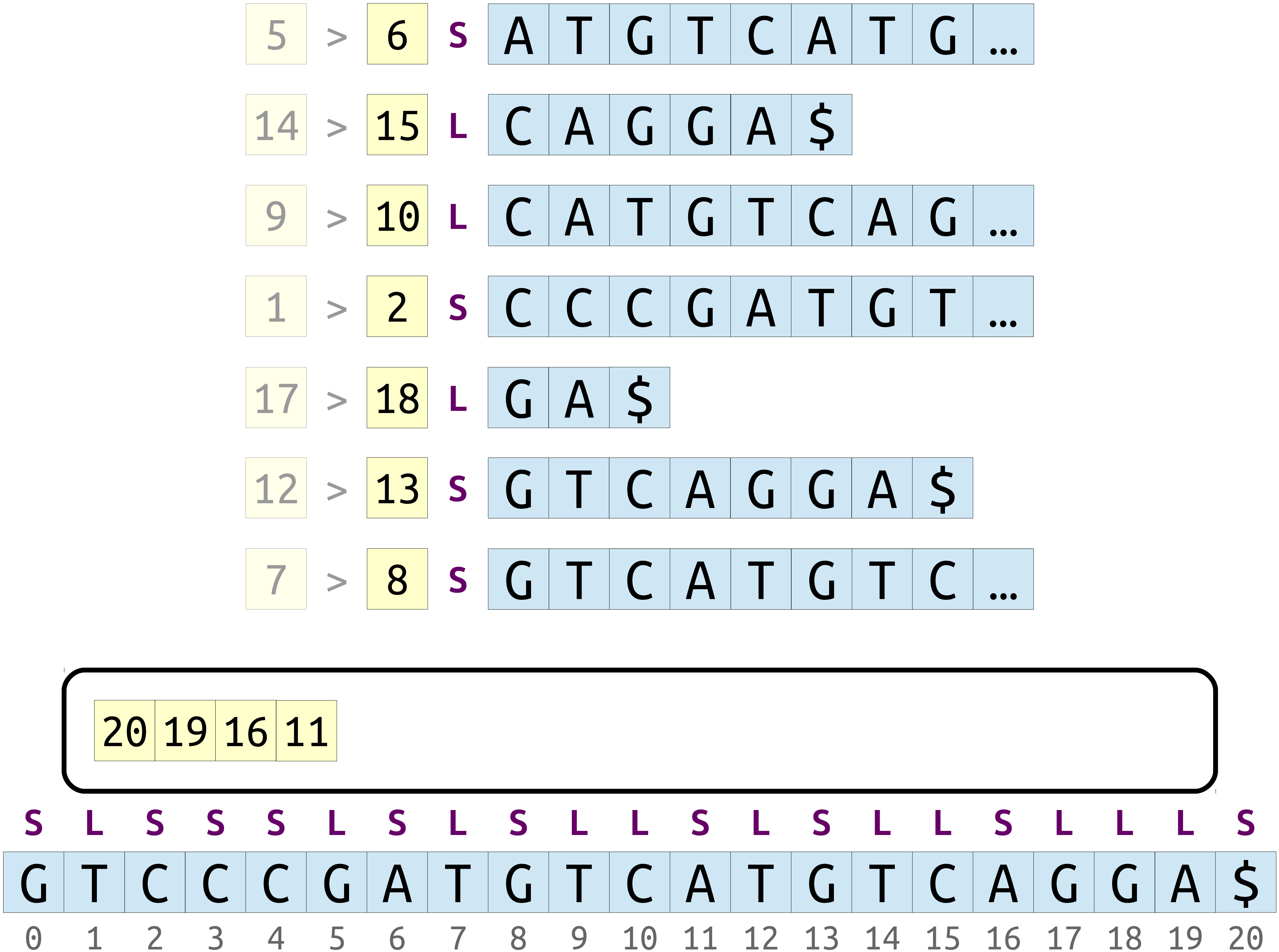


9 > 10 L C Suffix 11
5 > 6 S A T G T C A T G ...
14 > 15 L C Suffix 16
1 > 2 S C C C G
$
C A G G A $
7 > 8 S
The suffix at index 10 is C, followed by the suffix at index 11.
The suffix at index 15 is C, followed by the suffix at index 16.
Conclusion: This suffix goes after the L-type suffixes starting with C and before S-type suffixes starting with C.
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

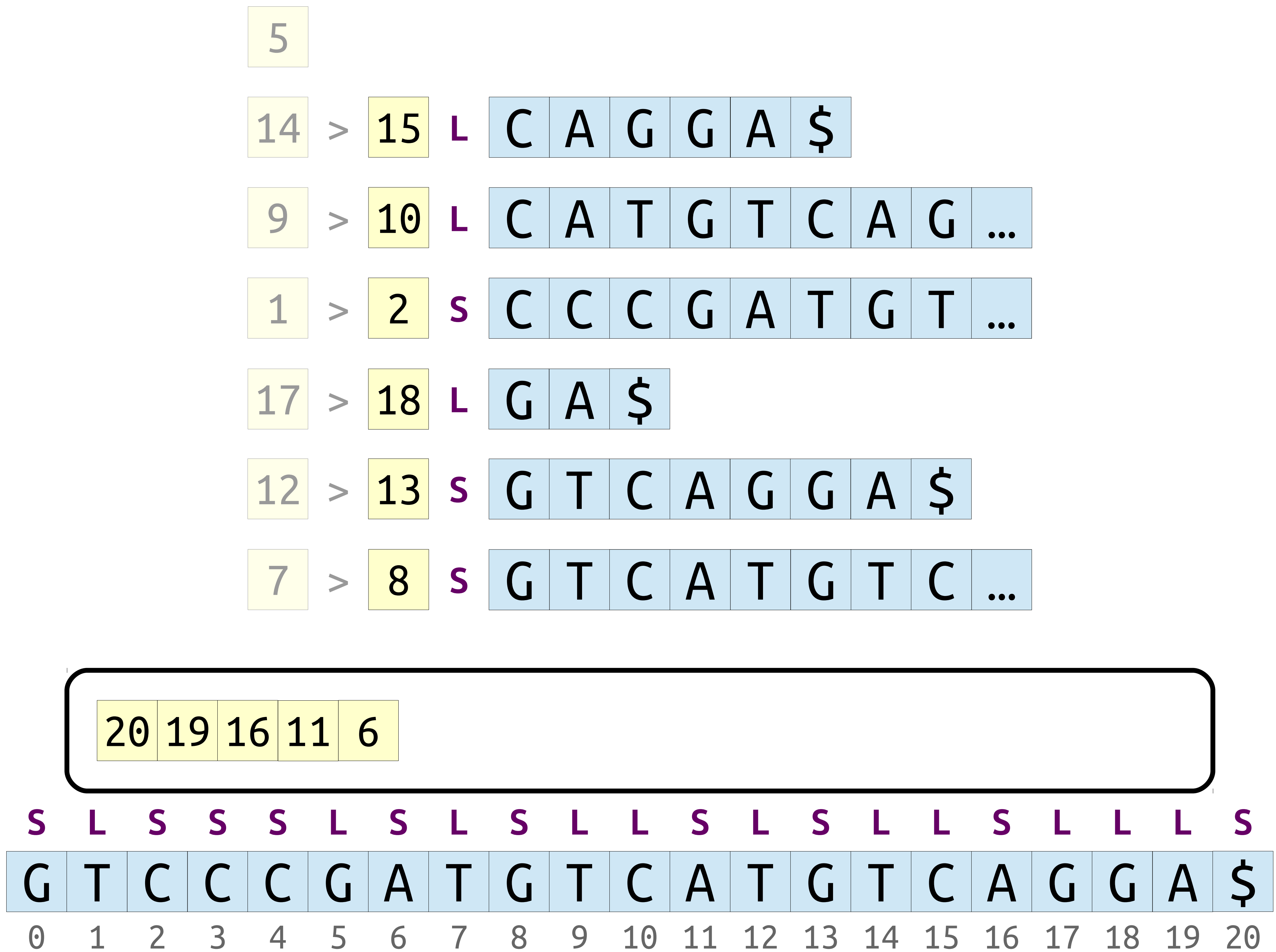

9 > 10 L C A T G T C A G ...
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



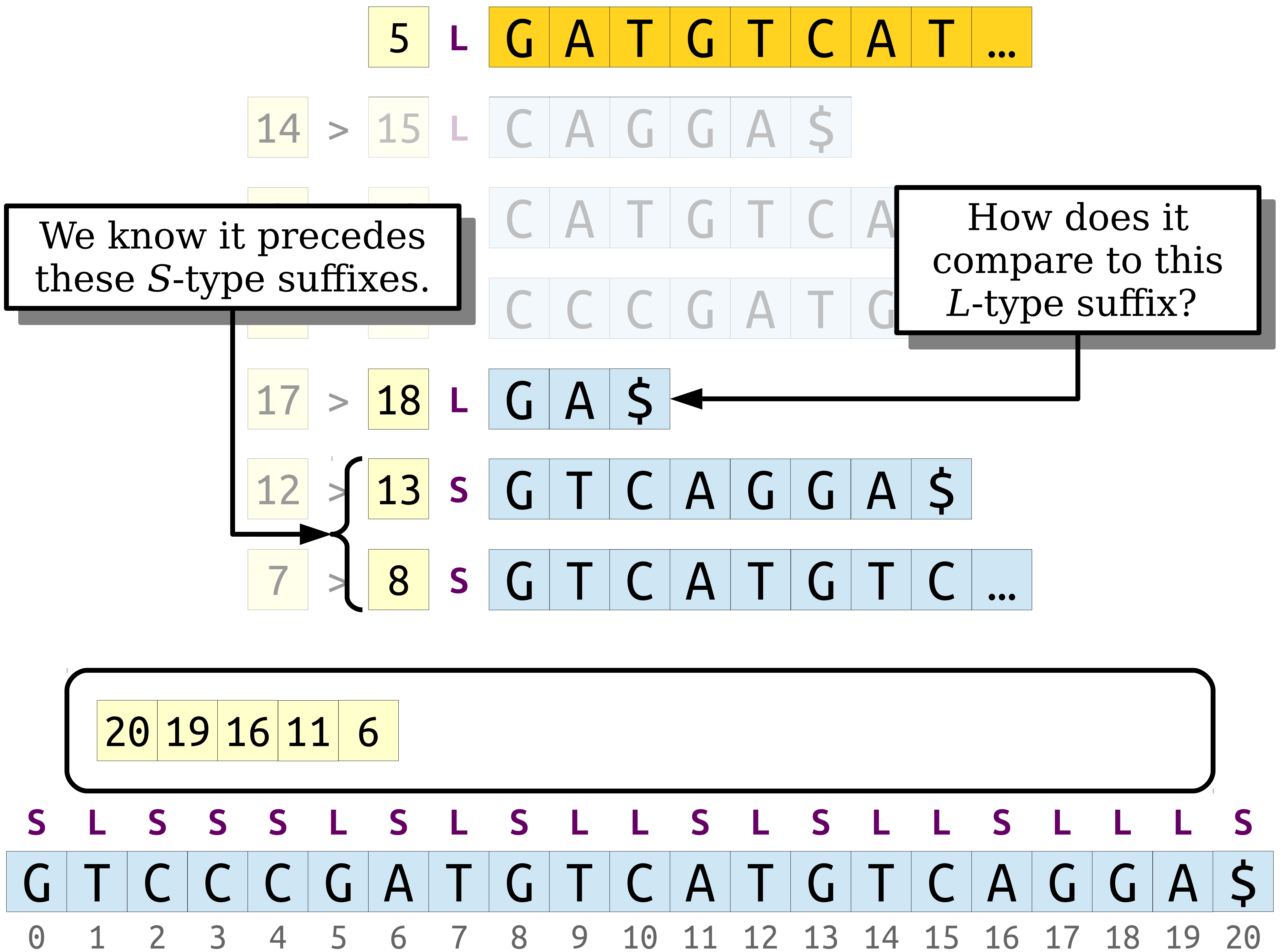
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
9 > 10 L C A T G T C A G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



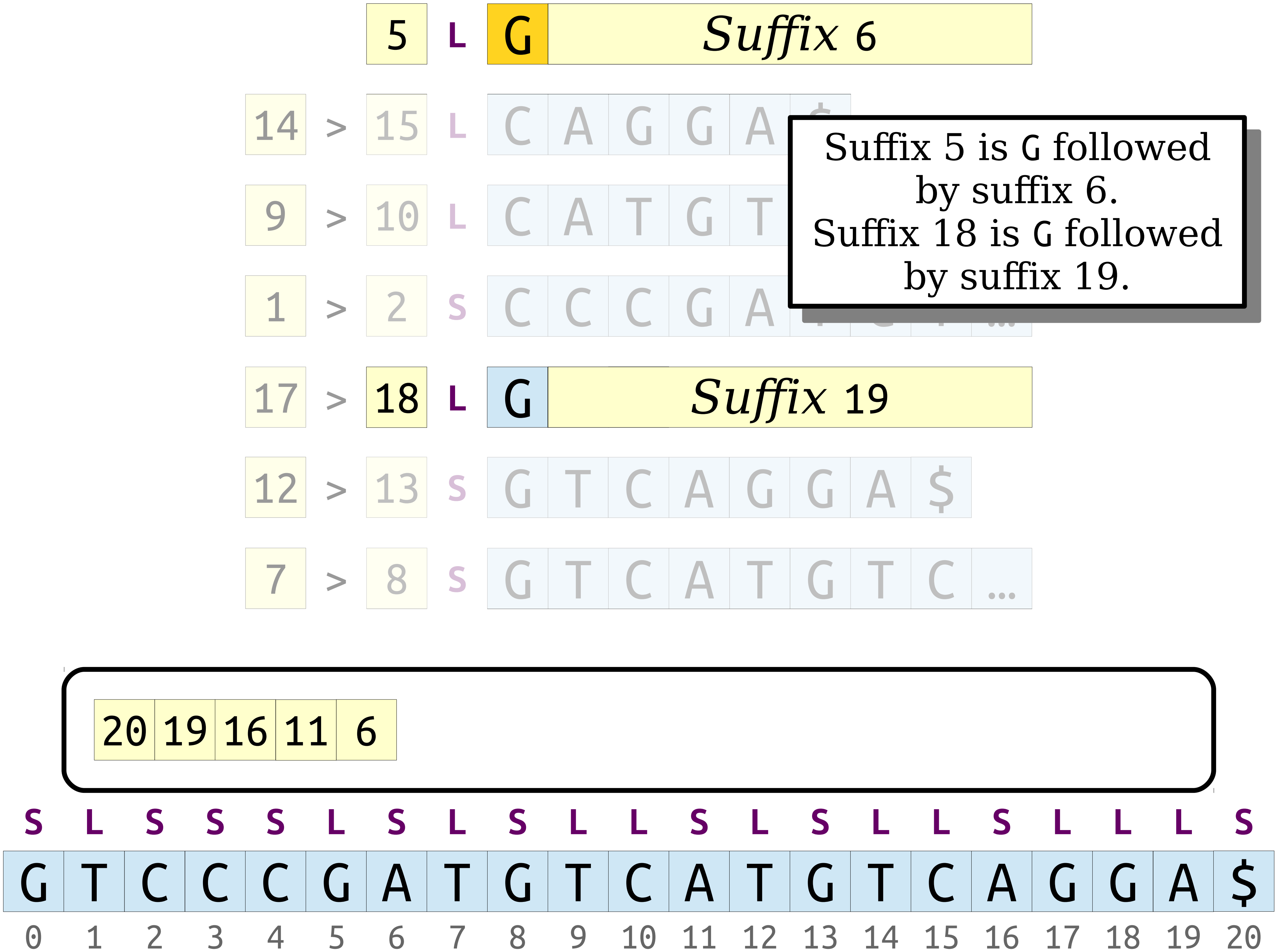
5 > 6 S A T G T C A T G ...
14 > 15 L C A G G A $
9 > 10 L C A T G T C A G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

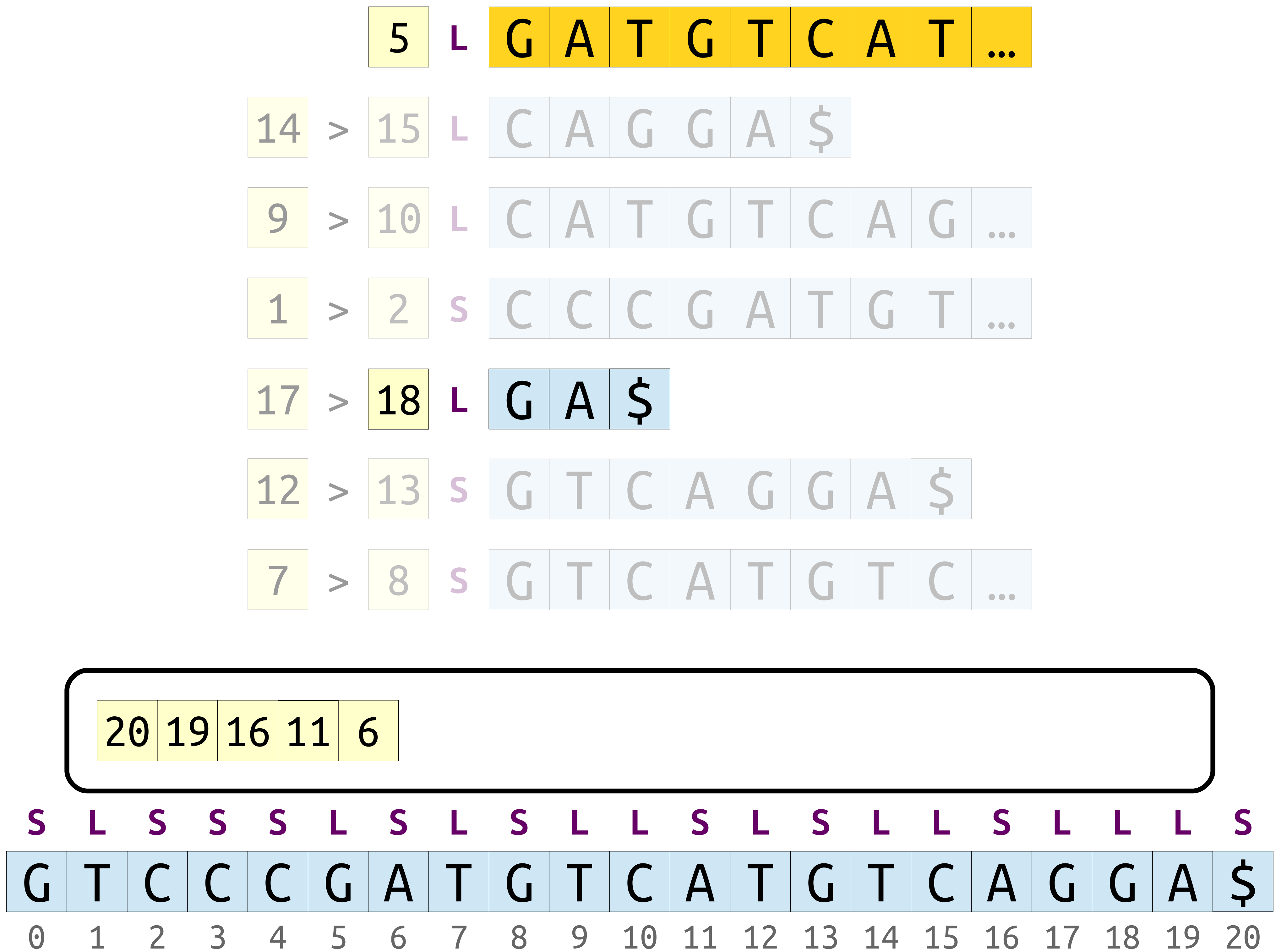


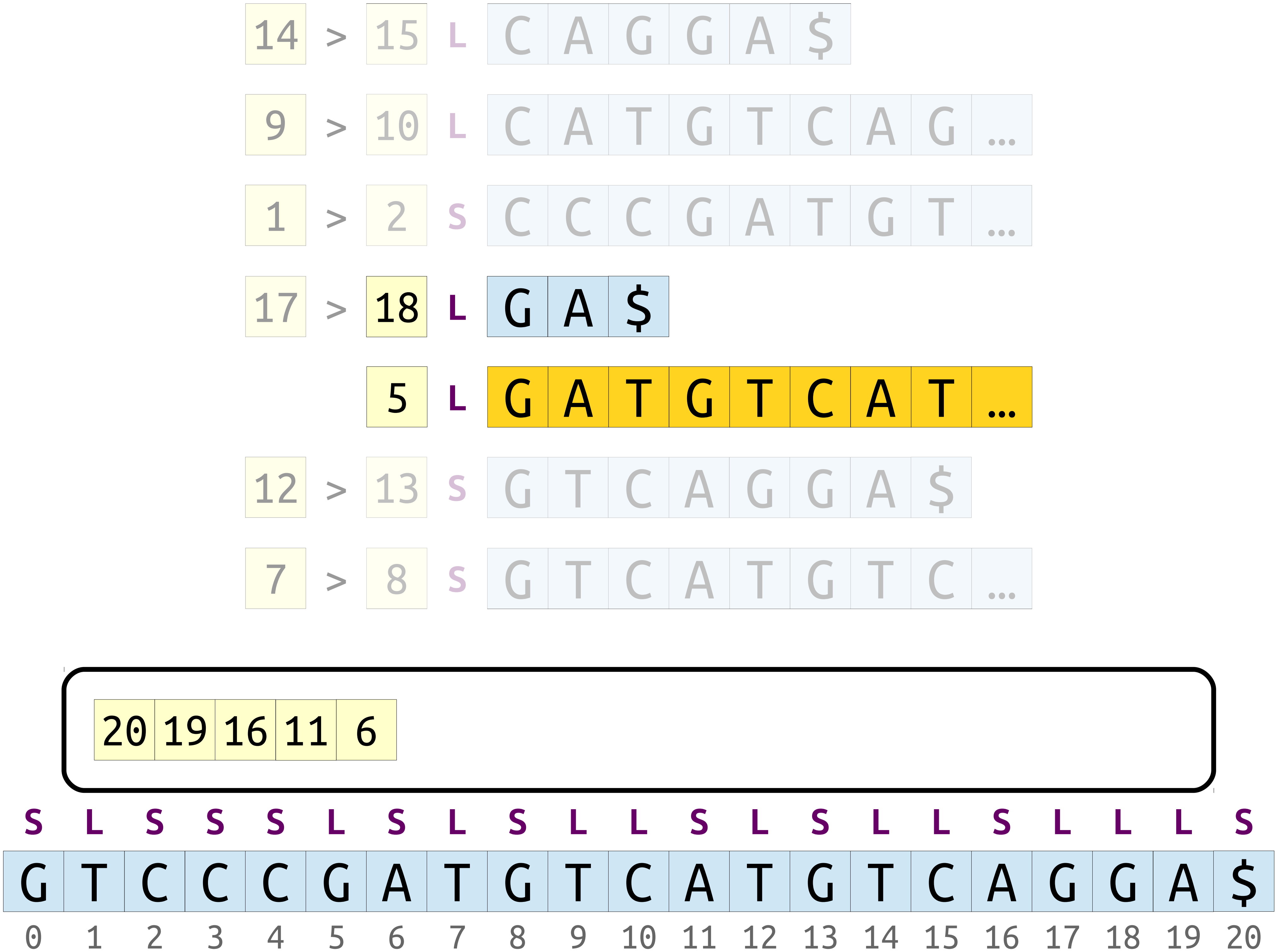
5
14 > 15 L C A G G A $
9 > 10 L C A T G T C A G …
1 > 2 S C C C G A T G T …
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C …
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

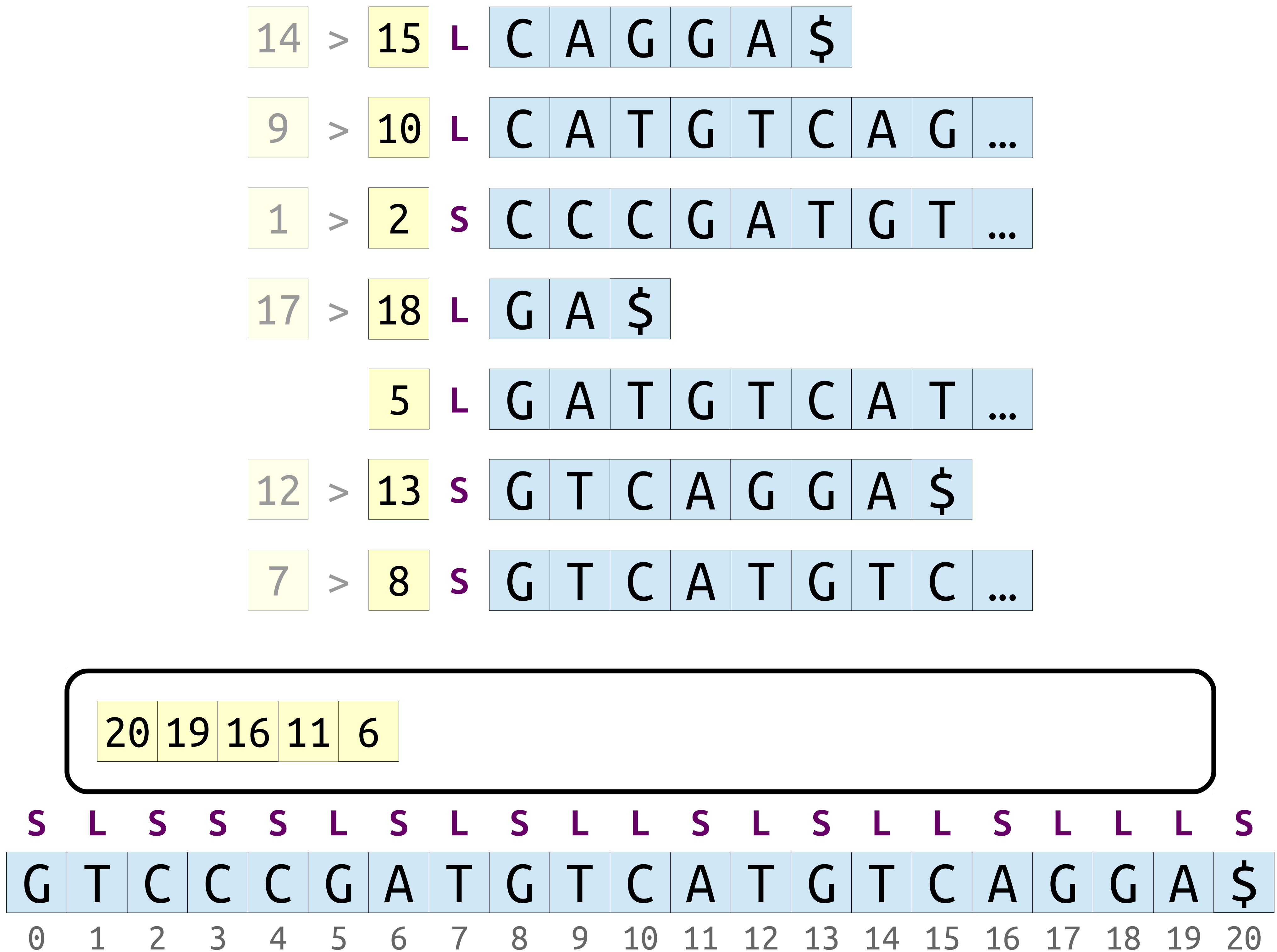


5 L G A T G T C A T ...
14 > 15 L C A G G A $
C A T G T C A
C C C G A T G
We know it precedes these S-type suffixes.
How does it compare to this L-type suffix?
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


5 L G *Suffix* 6
14 > 15 L C A G G A $
9 > 10 L C A T G T
1 > 2 S C C C G A
Suffix 5 is G followed by suffix 6.
Suffix 18 is G followed by suffix 19.
17 > 18 L G *Suffix* 19
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C …
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


5 L G A T G T C A T ...
14 > 15 L C A G G A $
9 > 10 L C A T G T C A G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


14 > 15 L C A G G A $
9 > 10 L C A T G T C A G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
5 L G A T G T C A T ...
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

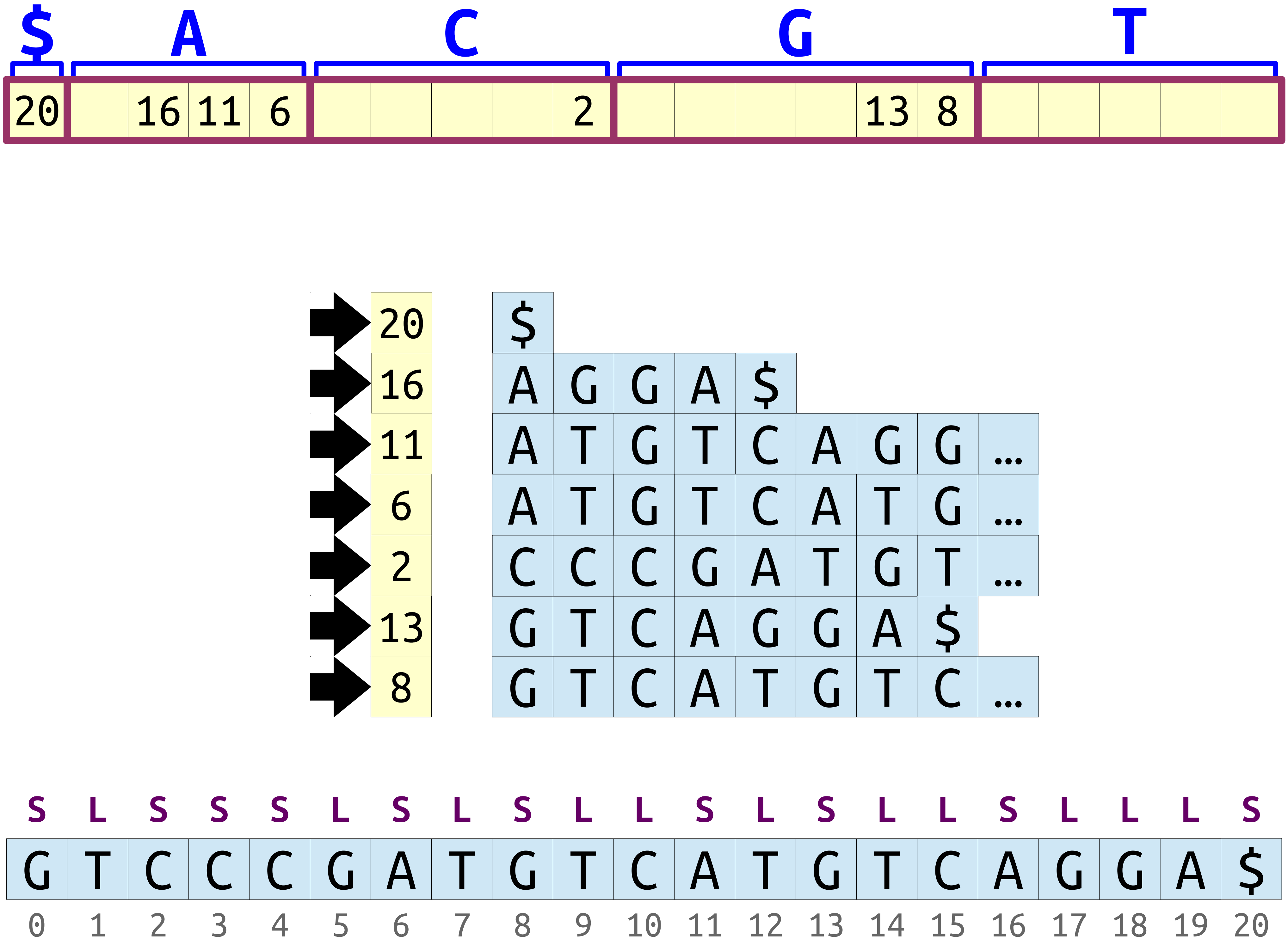


14 > 15 L C A G G A $
9 > 10 L C A T G T C A G ...
1 > 2 S C C C G A T G T ...
17 > 18 L G A $
5 L G A T G T C A T ...
12 > 13 S G T C A G G A $
7 > 8 S G T C A T G T C ...
20 19 16 11 6
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

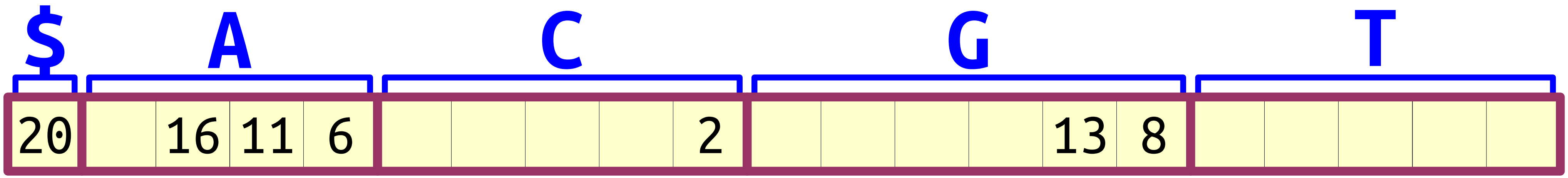
Some Observations

- All the new suffixes we uncover are L-type.
- Whenever we uncover a new suffix:
  - that suffix comes ***before*** all S-type suffixes in the list with the same first character, and
  - that suffix comes ***after*** all L-type suffixes in the list with the same first character.
- Notice that ***we never make any string comparisons*** in the course of carrying out this multiway merge!
- If we can maintain these buckets efficiently, we could complete this merge in time $O(m)$.

Okay, this next part is pretty cool.
Props to Ko and Aluru for figuring it out.

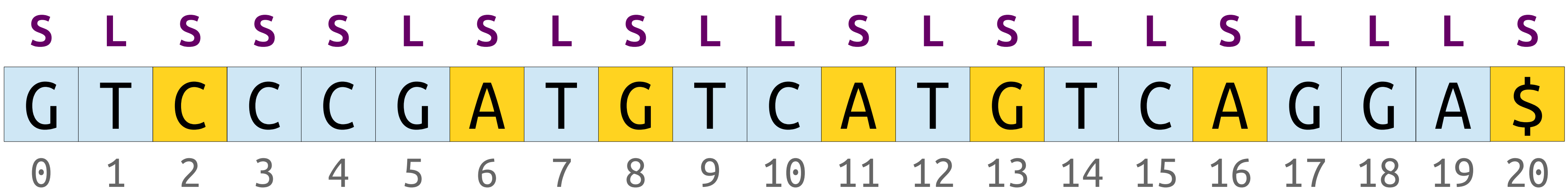


$ A C G T
20 16 11 6 2 13 8
20 $
16 A G G A $
11 A T G T C A G G ...
6 A T G T C A T G ...
2 C C C G A T G T ...
13 G T C A G G A $
8 G T C A T G T C ...
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



We can compute the bucket boundaries in time O(m) by just counting up how frequently each character appears in the string. If we store those boundaries in an array indexed by character, we can put each element in the right place in time O(1).

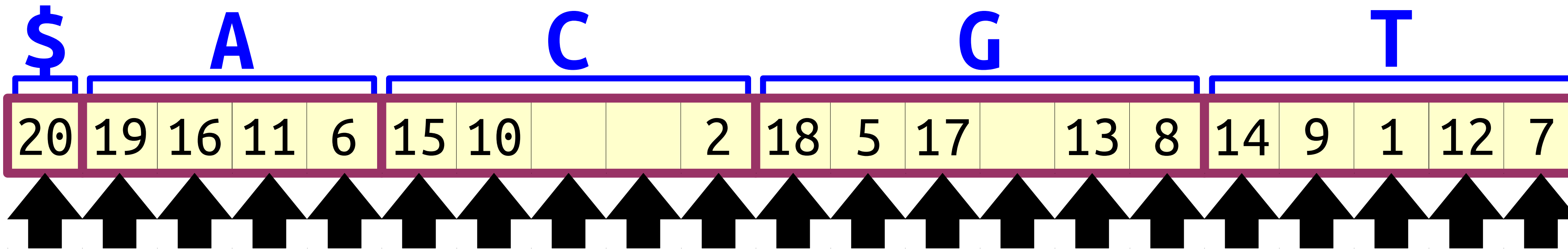
Total time so far: **O(m)**.



$	A				C					G						T				
20		16	11	6					2					13	8					

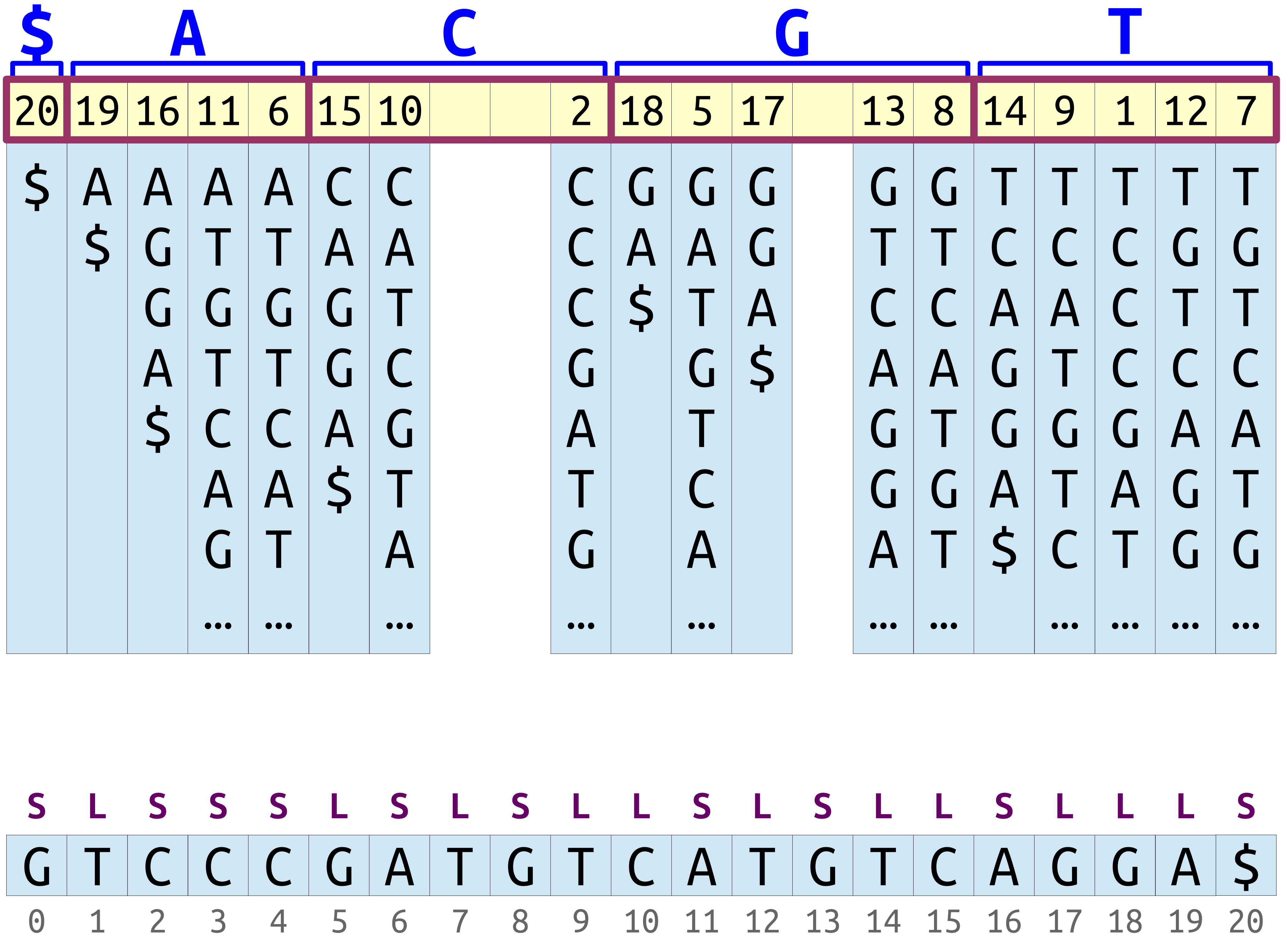
Watch how we implement the multiway merge.

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20




```
for (each index i in SA) {
   if (SA[i] isn't empty and SA[i] > 0 and
       text[SA[i] - 1] is L-type) {
      put SA[i] - 1 at the next free slot
        at the front of text[SA[i] - 1];
   }
}
```

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20


$ A C G T
20 19 16 11 6 15 10 2 18 5 17 13 8 14 9 1 12 7
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

$	A				C					G						T				
20	19	16	11	6	15	10			2	18	5	17		13	8	14	9	1	12	7

Theorem: If you have all the *L*-type suffixes in sorted order, you can use that to induce the order of the *S*-type suffixes by making a reverse pass over the array and following a similar algorithm.

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

$	A				C					G						T				
20	19	16	11	6	15	10			2	18	5	17		13	8	14	9	1	12	7

Theorem: If you have all the *L*-type suffixes in sorted order, you can use that to induce the order of the *S*-type suffixes by making a reverse pass over the array and following a similar algorithm.

Important detail: The ends of each bucket currently have some, but not all, of the *S*-type suffixes in them. ***These items may be out of place*** because we don't know how they relate to other *S*-type suffixes. Therefore, when doing this backwards pass, we'll allow ourselves to overwrite the old *S*-type suffixes as we go. Anything that wasn't overwritten was already in the right place.

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

$	A				C					G						T				
20	19	16	11	6	15	10			2	18	5	17		13	8	14	9	1	12	7

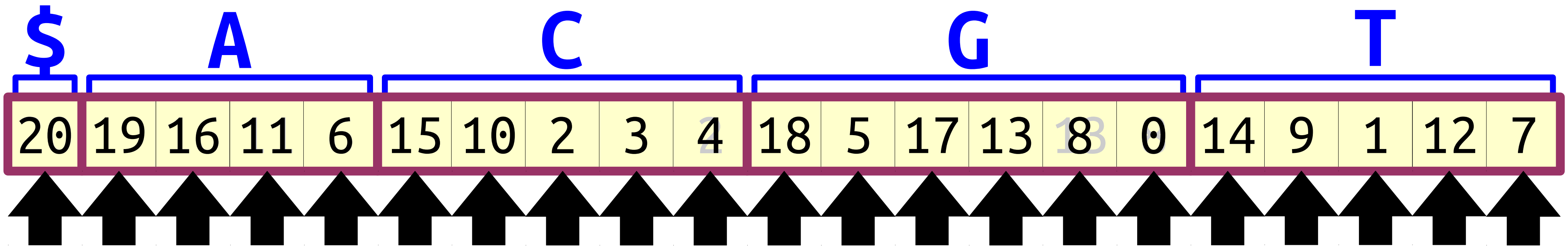
```
reset the indices of each bucket's next free slot at the end.
for (each index i in SA, in reverse order) {
   if (SA[i] isn't empty and SA[i] > 0 and
        text[SA[i] - 1] is S-type) {
       put SA[i] - 1 at the next free slot
         at the end of text[SA[i] - 1];
   }
}
```

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

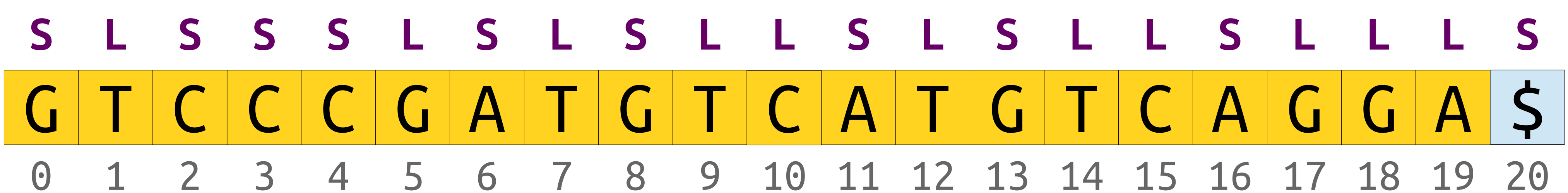
$	A				C					G						T				
20	19	16	11	6	15	10			2	18	5	17		13	8	14	9	1	12	7

```
reset the indices of each bucket's next free slot at the end.
for (each index i in SA, in reverse order) {
   if (SA[i] isn't empty and SA[i] > 0 and
       text[SA[i] - 1] is S-type) {
      put SA[i] - 1 at the next free slot
        at the end of text[SA[i] - 1];
   }
}
```

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20




```
reset the indices of each bucket's next free slot at the end.
for (each index i in SA, in reverse order) {
   if (SA[i] isn't empty and SA[i] > 0 and
       text[SA[i] - 1] is S-type) {
      put SA[i] - 1 at the next free slot
        at the end of text[SA[i] - 1];
   }
}
```



$	A				C					G						T				
20	19	16	11	6	15	10	2	3	4	18	5	17	13	8	0	14	9	1	12	7
$	A	A	A	A	C	C	C	C	C	G	G	G	G	G	G	T	T	T	T	T
	$	G	T	T	A	A	C	C	G	A	A	G	T	T	T	C	C	C	G	G
		G	G	G	G	T	C	G	A	$	T	A	C	C	C	A	A	C	T	T
		A	T	T	G	C	G	A	T		G	$	A	A	C	G	T	C	C	C
		$	C	C	A	G	A	T	G		T		G	T	C	G	G	G	A	A
			A	A	$	T	T	G	T		C		G	G	G	A	T	A	G	T
			G	T		A	G	T	C		A		A	T	A	$	C	T	G	G
			...	...		...	...	...	...		...		...	...	...		...	...	...	...

G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

To Recap

- Suppose that – somehow – we can sort the LMS suffixes.
- We can then make three linear scans to sort all the suffixes:
  - one ***reverse*** pass over the sorted LMS suffixes, placing them at the ends of their buckets;
  - one ***forward*** pass over the suffix array, placing L-type suffixes at the fronts of their buckets; and
  - one ***reverse*** pass over the suffix array, placing S-type suffix at the ends of their buckets (making sure to reset the end positions of each bucket first.)
- This runs in time $O(m)$ and has *excellent* locality of reference. It's incredibly fast in practice.

Time-Out for Announcements!

CS106 SECTION LEADING

Deadline for students who have completed CS106B/X (or equivalent): Thursday, April 25th, 11:59p.m.

Deadline for students currently enrolled in CS106B/X: Friday, May 10th, 11:59p.m.

All majors and backgrounds welcome!

Applicants must be currently enrolled in or have completed CS106B/X, or have equivalent experience.

Apply online!

cs198.stanford.edu

Questions? Contact the CS198 Coordinators (Luke Sturm, Nicholas Bowman, and Sonja Johnson-Yu) at cs198@cs.stanford.edu

Problem Sets

- Problem Set One was due today at 2:30PM.
  - Need some more time? You have two, 48-hour late periods you can use throughout the quarter.
- Problem Set Two goes out today. It’s due next Thursday at 2:30PM.
  - Play around with tries, suffix trees, suffix arrays, and their algorithms!

Back to CS166!

SA-IS at a Glance

- There are three core insights that collectively give us the SA-IS algorithm.
- First:

There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

- Second:

Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

- Third:

With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.

SA-IS at a Glance

There are three core insights that collectively give us the SA-IS algorithm.

First:

There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

- Second:

Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

Third:

With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.

C C C G A T G T C A T G T C A G G A $

A T G T C A T G T C A G G A $

G T C A T G T C A G G A $

A T G T C A G G A $

G T C A G G A $

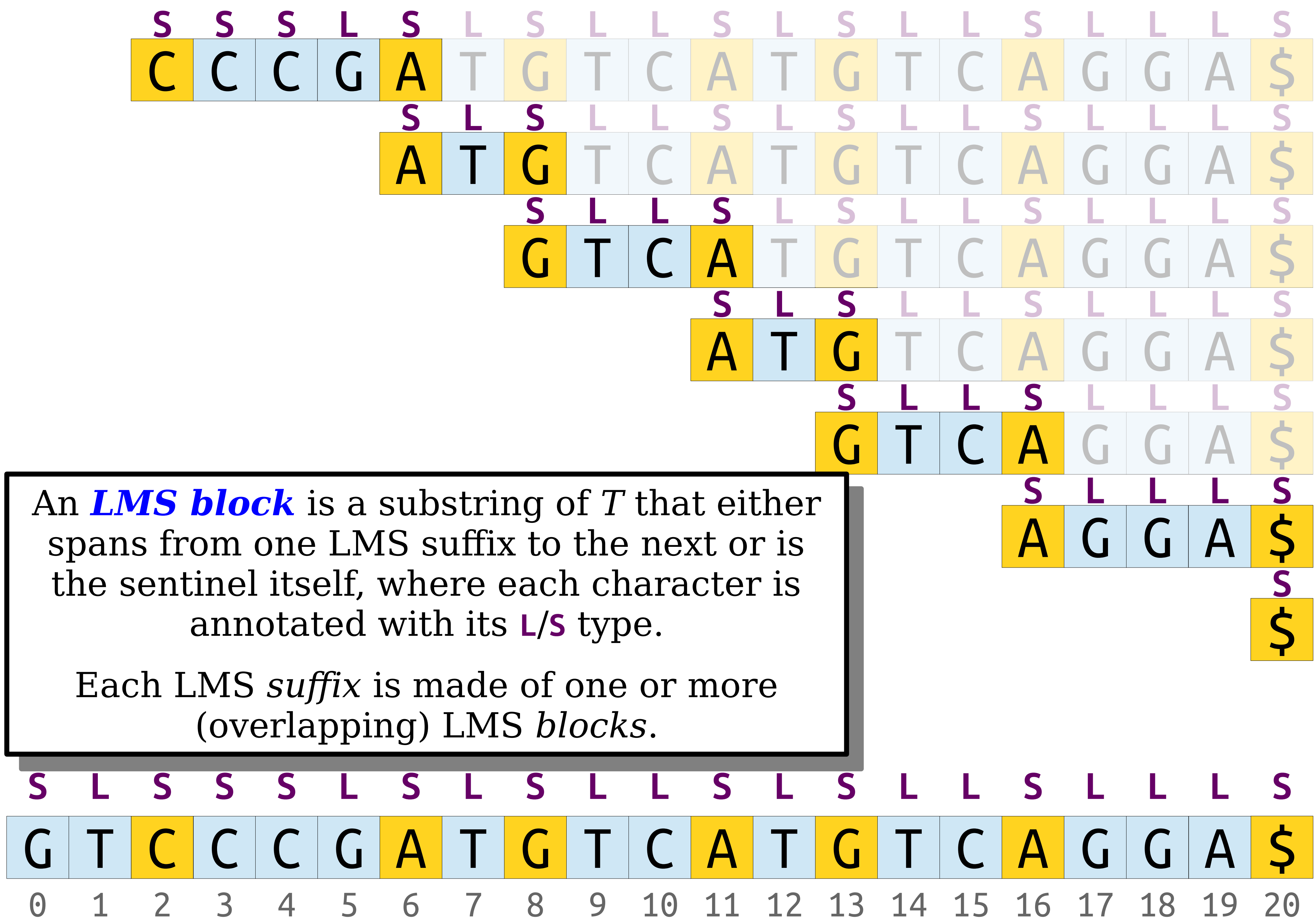
A G G A $

$

Observation: The comparisons between these strings seem to be boiling down to comparisons between their blocks.

Why is that?

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



An ***LMS block*** is a substring of T that either spans from one LMS suffix to the next or is the sentinel itself, where each character is annotated with its **L**/**S** type.

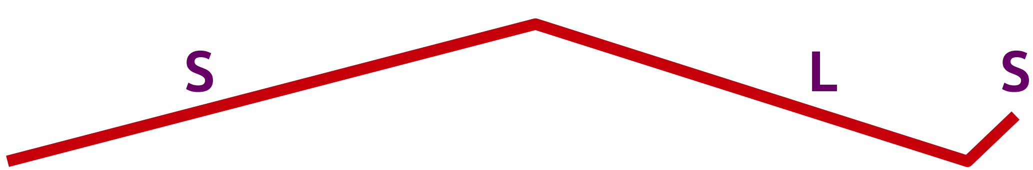
Each LMS *suffix* is made of one or more (overlapping) LMS *blocks*.

Theorem: Treat each character in an LMS block as a pair of the character itself and its L/S type. Then no LMS block is a prefix of another LMS block.

Corollary: If two different LMS blocks are compared factoring in L/S types, a mismatch will be found somewhere inside the blocks.

Claim 1: Every suffix starting at an LMS character is a local minimum among the suffixes near it in the original string.

Claim 2: With the exception of the sentinel, the types of the characters in an LMS block match the regex S^+L^+S.



S	S	S	L	S
C	C	C	G	A

S	L	S
A	T	G

S	L	L	S
G	T	C	A

S	L	S
A	T	G

S	L	L	S
G	T	C	A

S	L	L	L	S
A	G	G	A	$

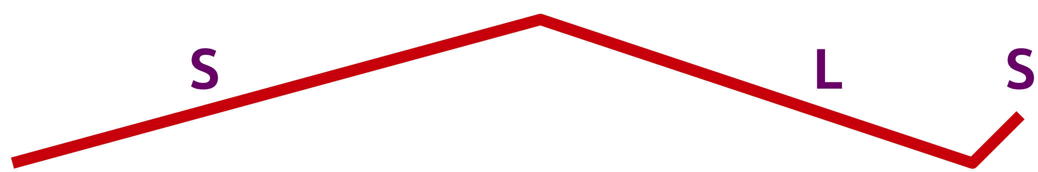
S
$

S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Theorem: Treat each character in an LMS block as a pair of the character itself and its **L**/**S** type. Then no LMS block is a prefix of another LMS block.

Corollary: If two different LMS blocks are compared factoring in **L**/**S** types, a mismatch will be found somewhere inside the blocks.

Claim 2: With the exception of the sentinel, the types of the characters in an LMS block match the regex S^+L^+S.



Proof: A comparison of two different LMS blocks will result in a mismatch no later than the first occurrence of **LS**.

S	S	S	L	S
C	C	C	G	A

S	L	S
A	T	G

S	L	L	S
G	T	C	A

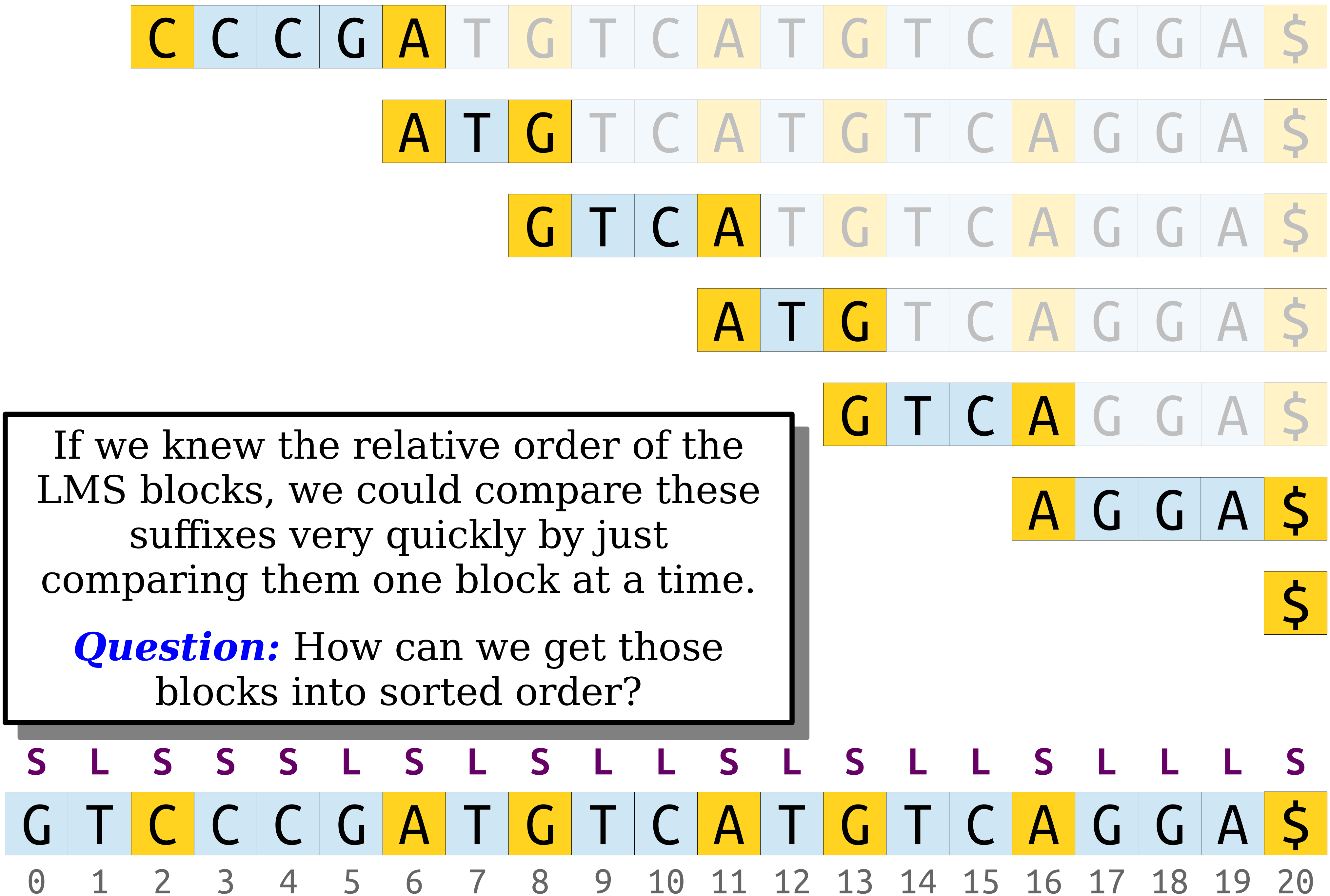
S	L	S
A	T	G

S	L	L	S
G	T	C	A

S	L	L	L	S
A	G	G	A	$

S
$

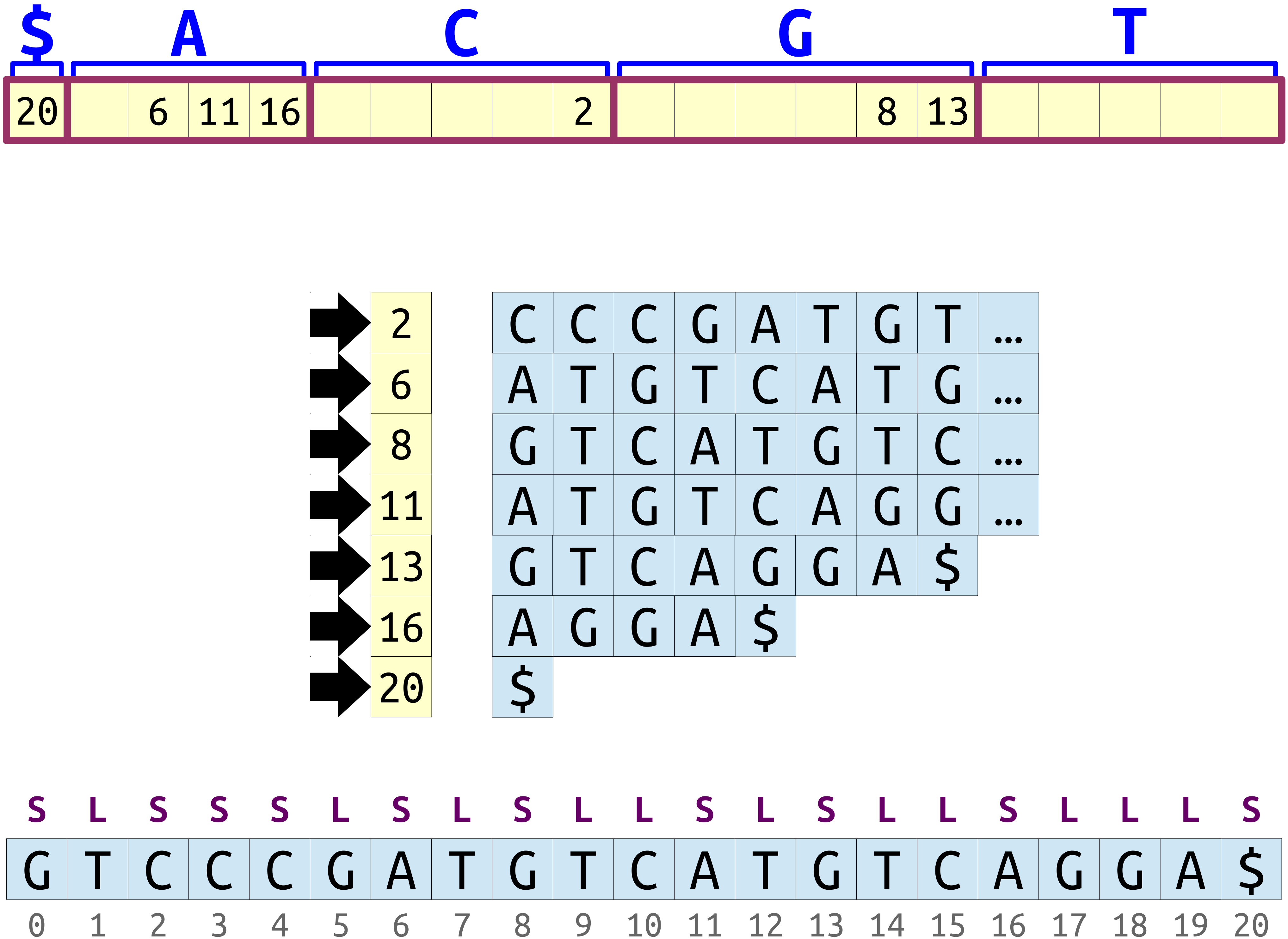
S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



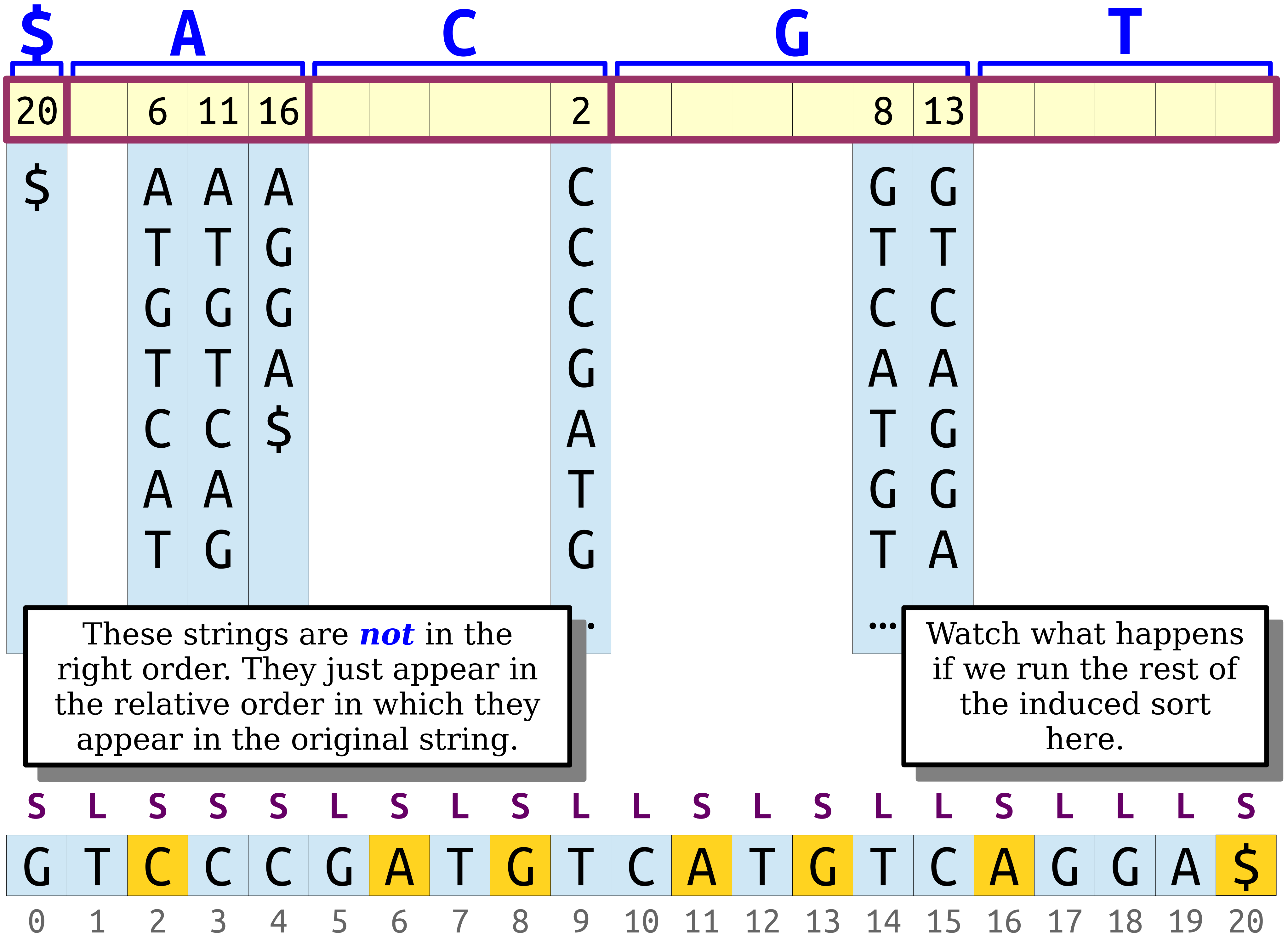
If we knew the relative order of the LMS blocks, we could compare these suffixes very quickly by just comparing them one block at a time.

Question: How can we get those blocks into sorted order?

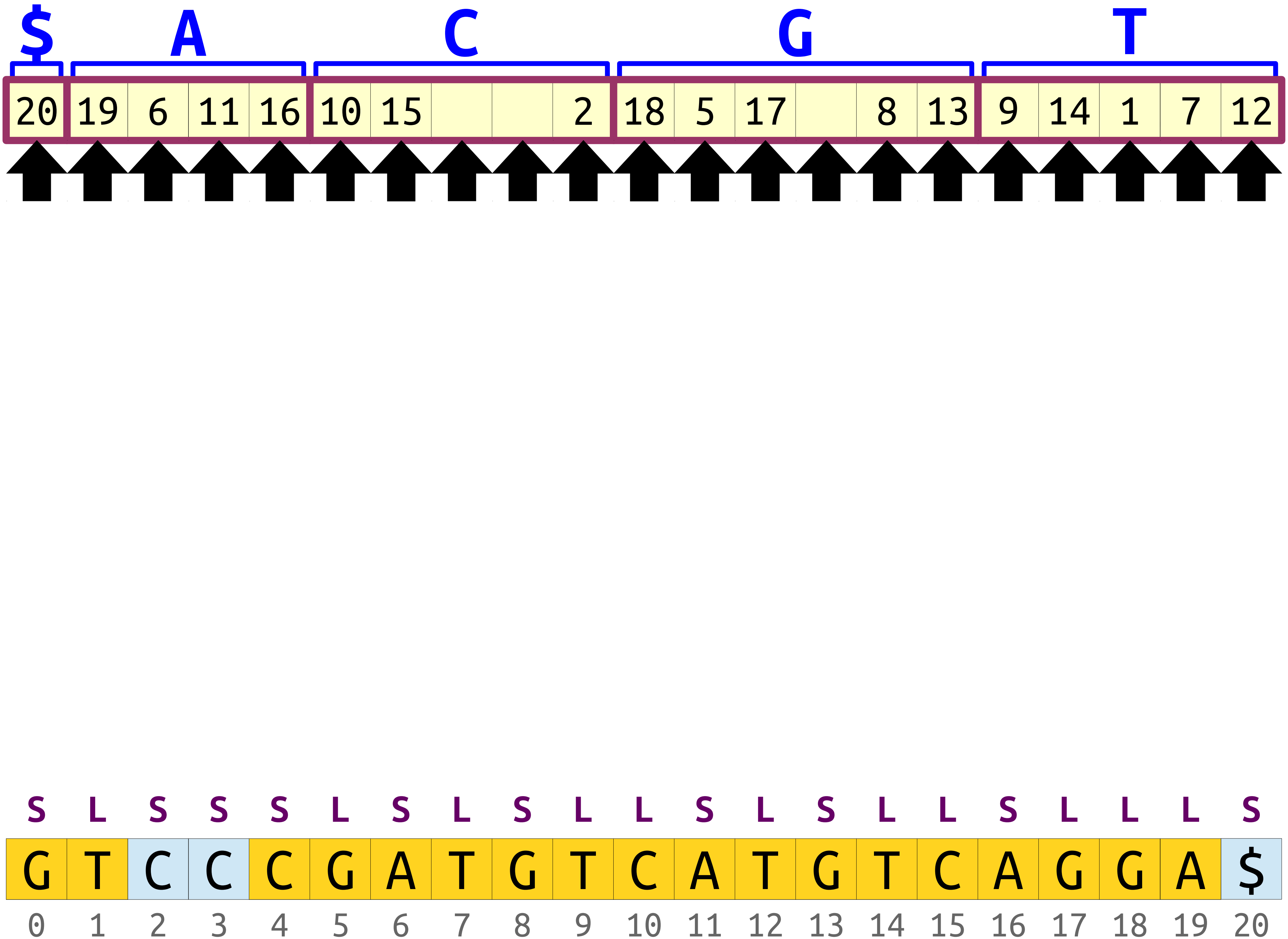
This next bit is totally brilliant.
A huge shoutout to Nong, Zhang, and Chan
for figuring this one out.

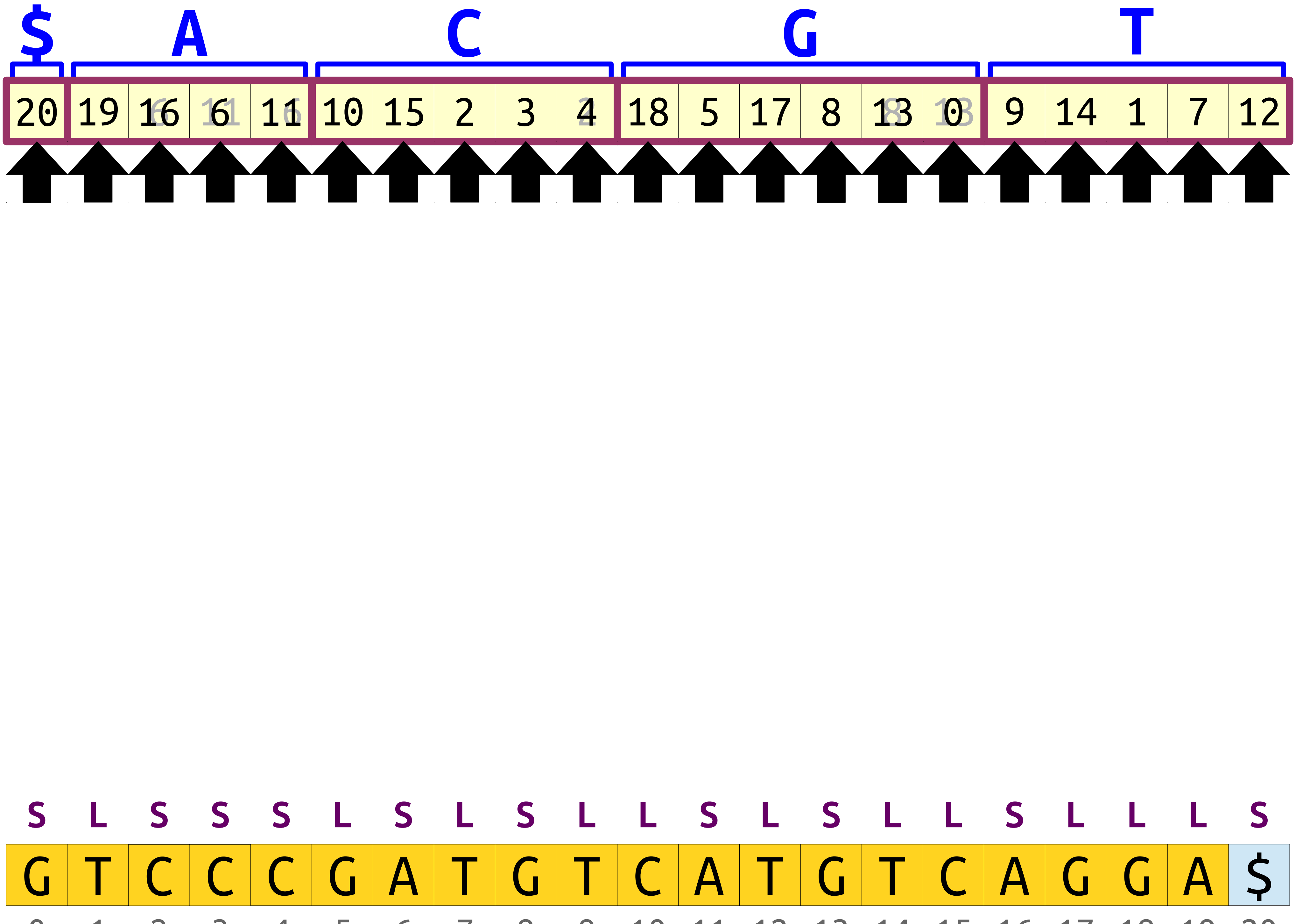


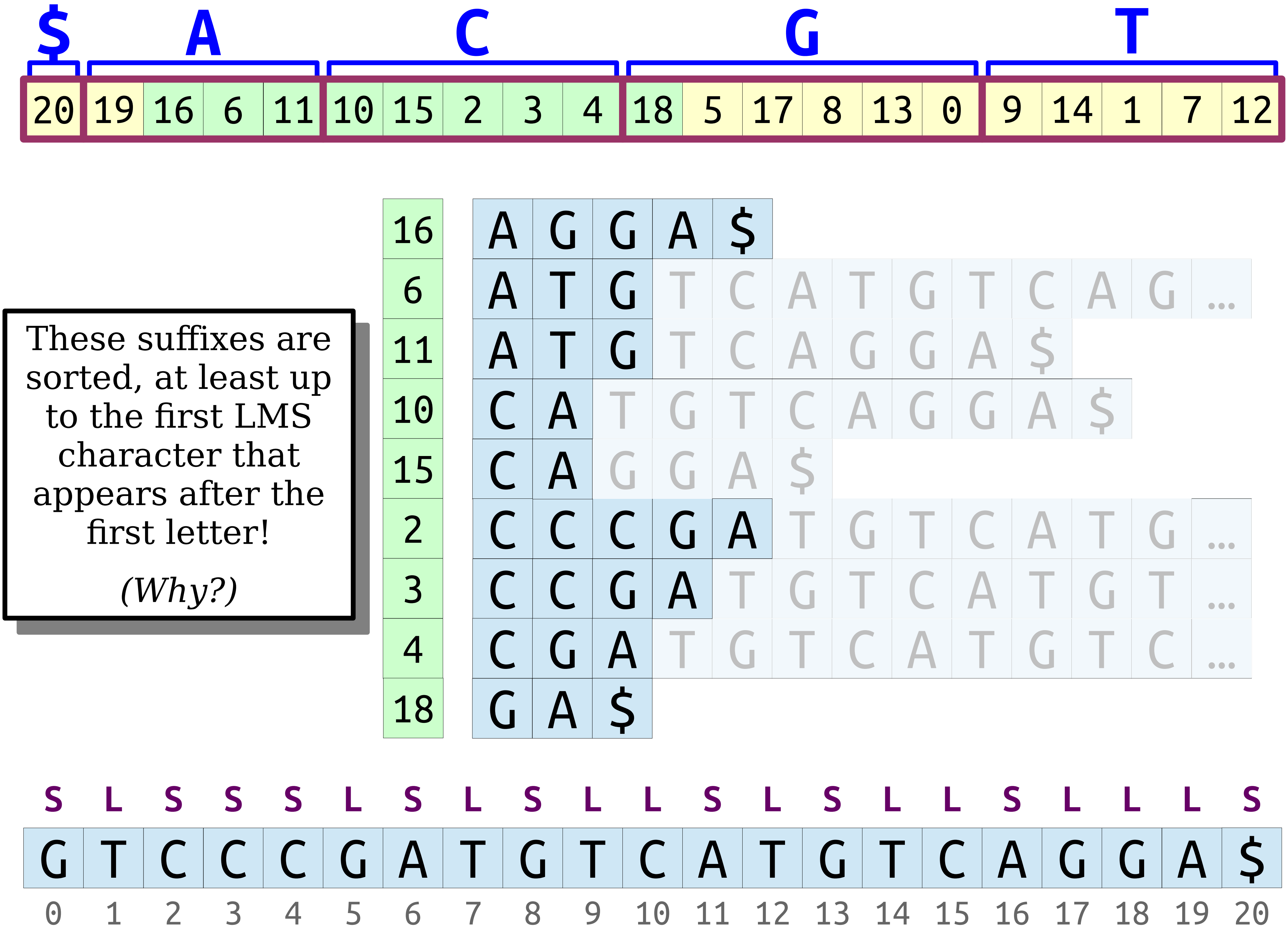
$
A
C
G
T
20
6 11 16
2
8 13
2
6
8
11
13
16
20
C C C G A T G T ...
A T G T C A T G ...
G T C A T G T C ...
A T G T C A G G ...
G T C A G G A $
A G G A $
$
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



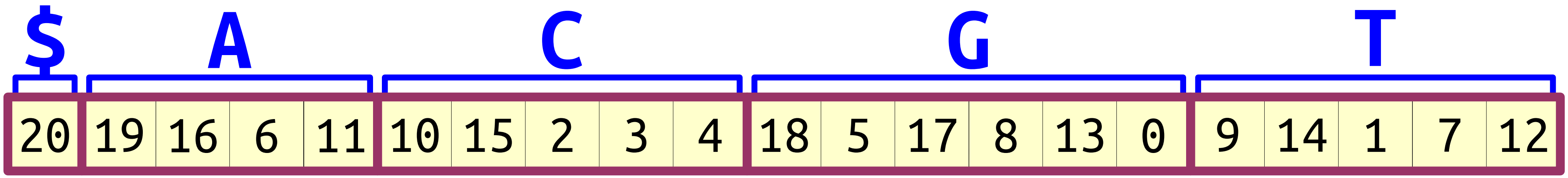
$
A
C
G
T
20
6
11
16
2
8
13
$
ATGTCAT
ATGTCAG
AGGA$
CCCGATG
GTCATGT
GTCAGGA
...
...
...
These strings are *not* in the right order. They just appear in the relative order in which they appear in the original string.
Watch what happens if we run the rest of the induced sort here.
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

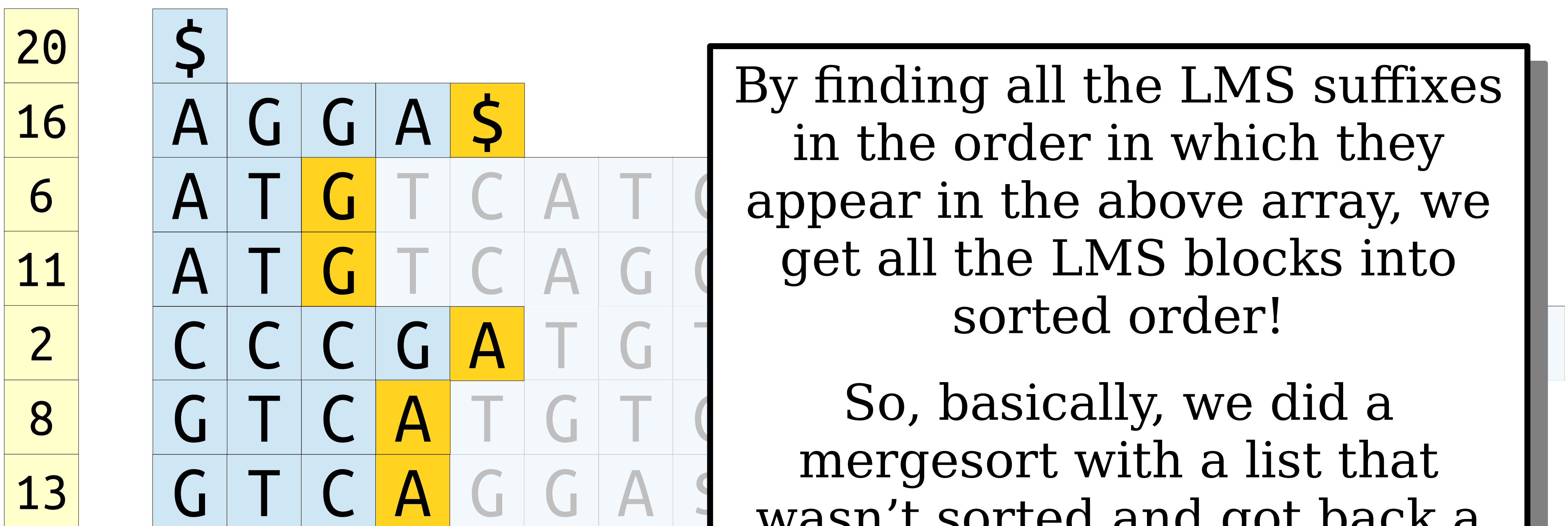

$ A C G T
20 19 6 11 16 10 15 2 18 5 17 8 13 9 14 1 7 12
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20


$ A C G T
20 19 16 6 11 10 15 2 3 4 18 5 17 8 13 0 9 14 1 7 12
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



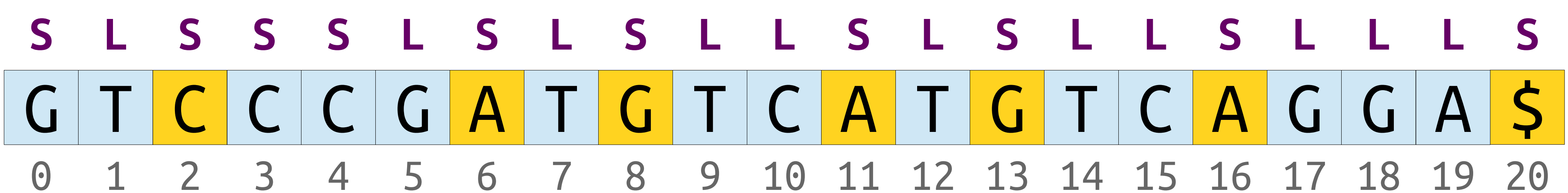
$ A C G T
20 19 16 6 11 10 15 2 3 4 18 5 17 8 13 0 9 14 1 7 12
16 AGGA$
6 ATGTCATGTCAG...
11 ATGTCAGGA$
10 CATGTCAGGA$
15 CAGGA$
2 CCCGATGTCATG...
3 CCGATGTCATGT...
4 CGATGTCATGTC...
18 GA$
These suffixes are sorted, at least up to the first LMS character that appears after the first letter!
(Why?)
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20





By finding all the LMS suffixes in the order in which they appear in the above array, we get all the LMS blocks into sorted order!

So, basically, we did a mergesort with a list that wasn't sorted and got back a list that is. Kinda.



To Recap

- The relative order of the LMS suffixes depends purely on the relative order of the LMS blocks.
- The order of the LMS blocks can be found by running the induced sorting algorithm on a list of all the LMS suffixes in any order we'd like!
- We're almost done!

SA-IS at a Glance

- There are three core insights that collectively give us the SA-IS algorithm.
- First:

There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

- Second:

Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

- Third:

With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.

SA-IS at a Glance

- There are three core insights that collectively give us the SA-IS algorithm.
- First:

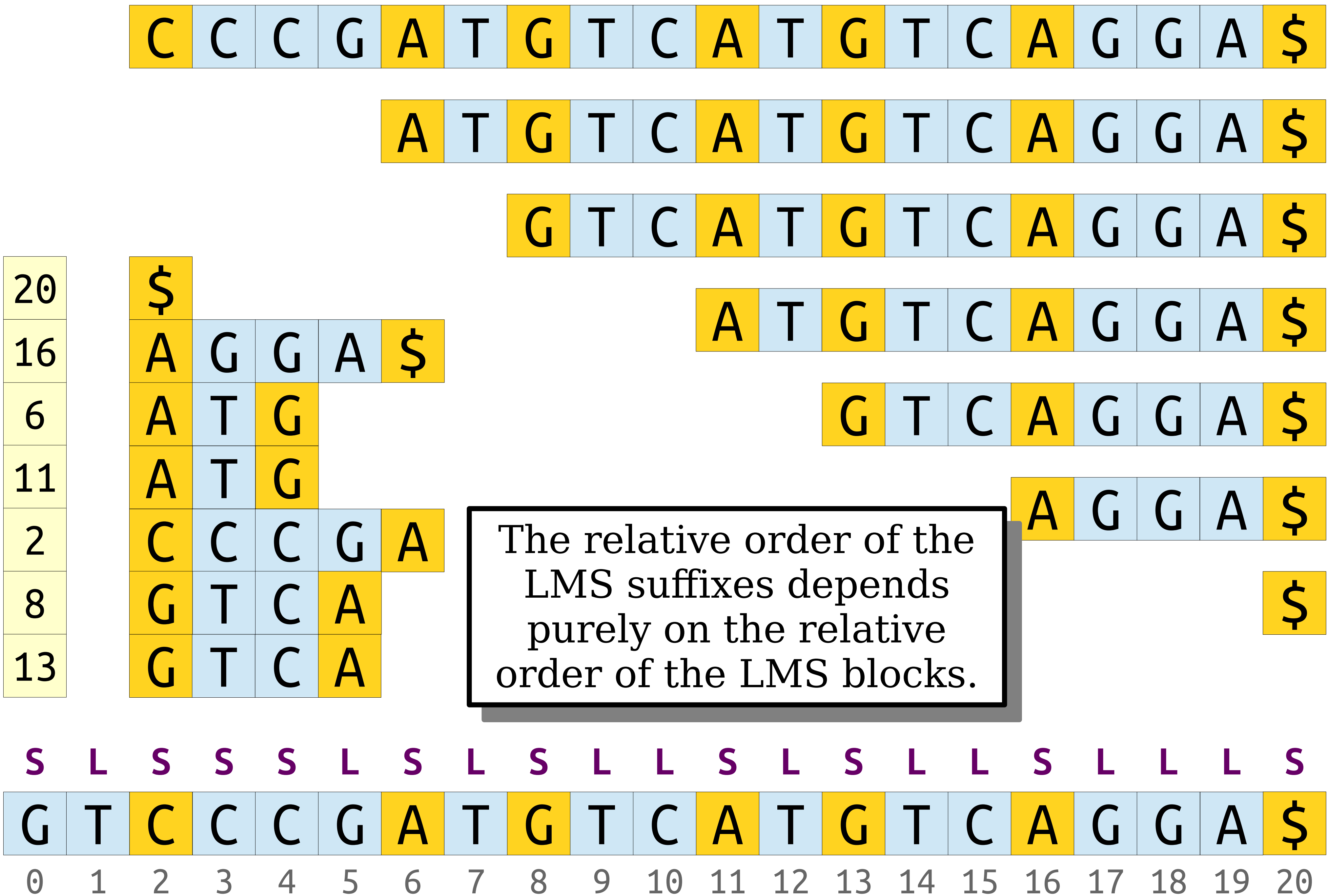
  There is a proper subset of the suffixes that, if sorted, can be used to recover the order of all the remaining suffixes.

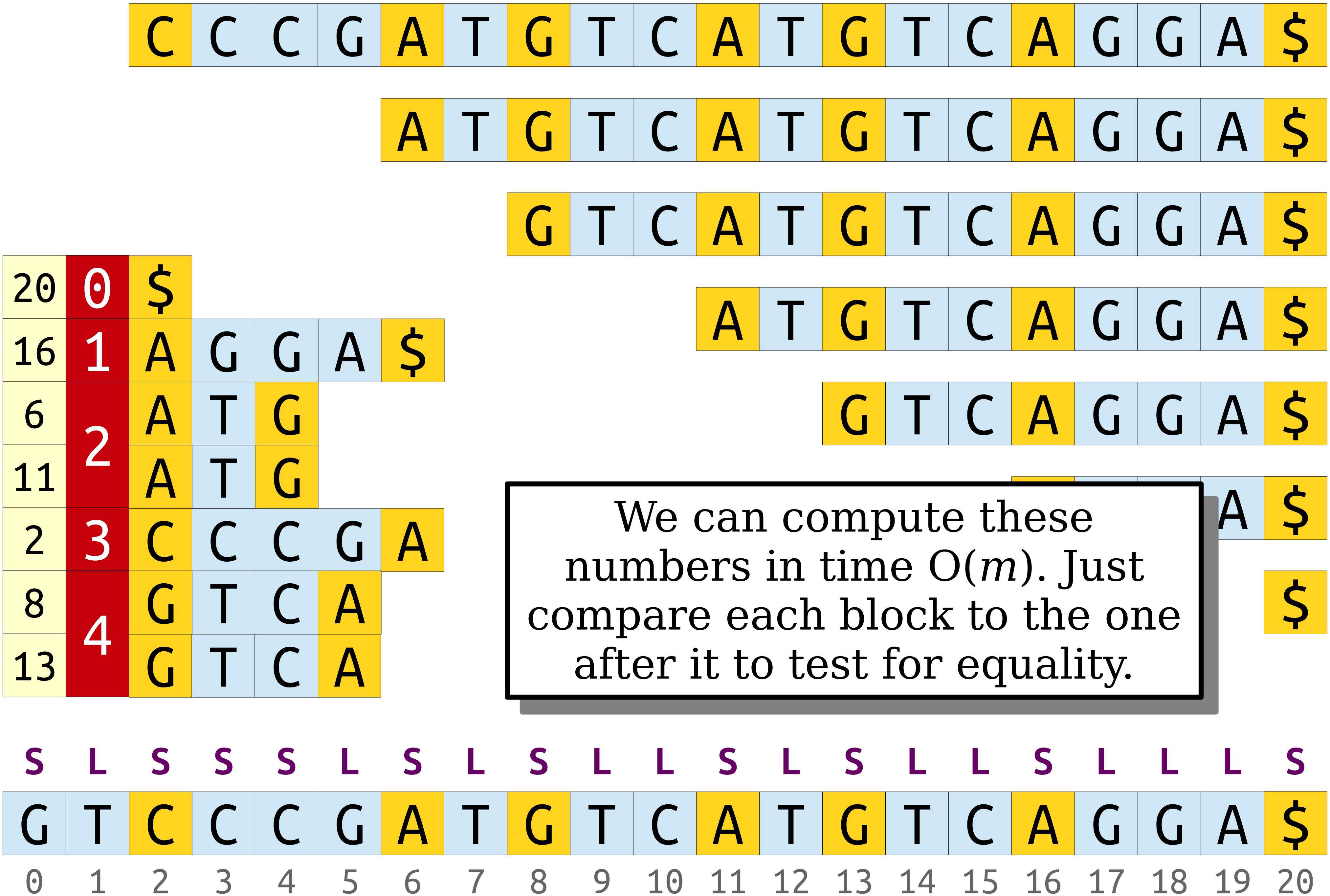
- Second:

  Those suffixes can be broken apart into blocks of characters such that the order of the suffixes depends purely on the order of the blocks.

- Third:

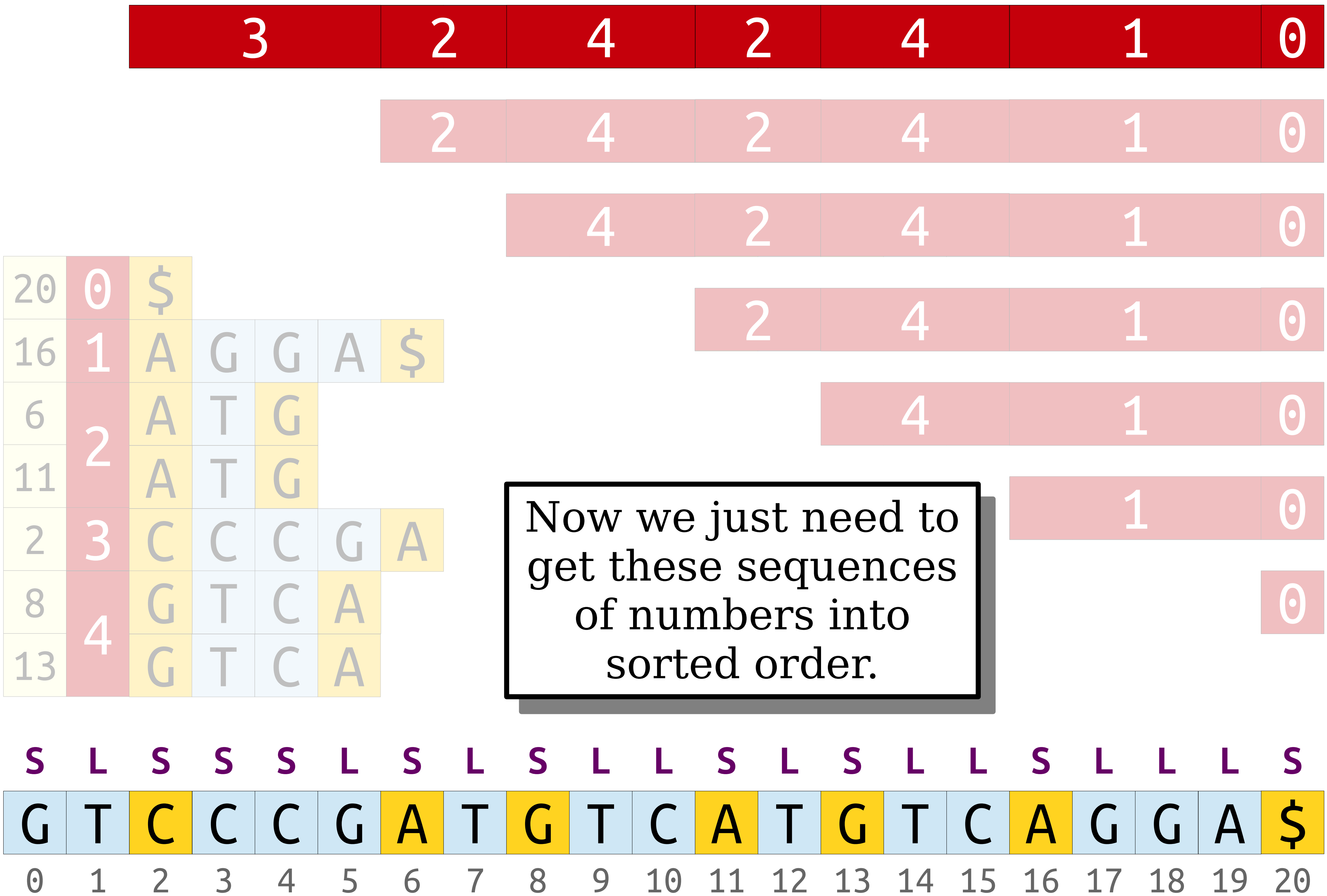
  With the proper preprocessing, those suffixes can be sorted via a recursive call on a smaller input string.


The relative order of the LMS suffixes depends purely on the relative order of the LMS blocks.
20
16
6
11
2
8
13
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



We can compute these numbers in time O(*m*). Just compare each block to the one after it to test for equality.

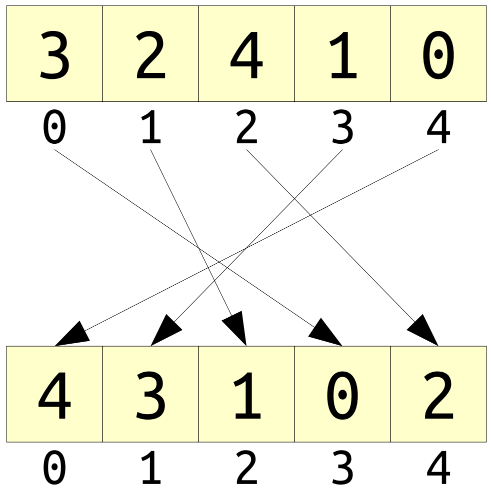
We need a suffix array for this reduced string!



Now we just need to get these sequences of numbers into sorted order.

Recursion to the Rescue

- The SA-IS algorithm handles this step recursively, with a very cleverly-chosen base case.
- ***Base Case:*** If all blocks are unique, the suffix array can be computed manually in time O(m) by writing down the indices of 0, 1, 2, …, k.

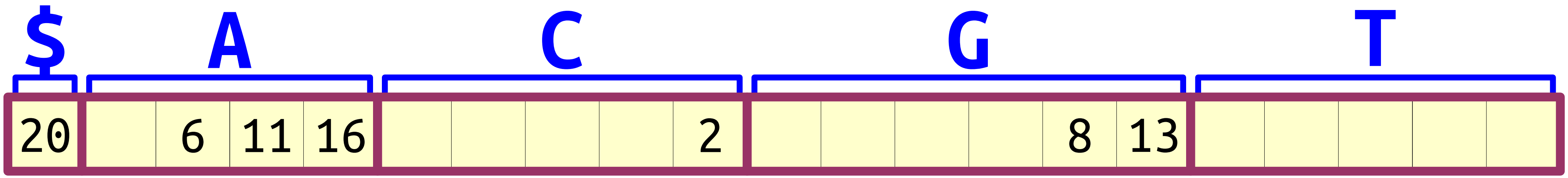


- ***Recursive Case:*** Otherwise, recursively invoke SA-IS to get the suffix array!

The Whole Algorithm, End-to-End

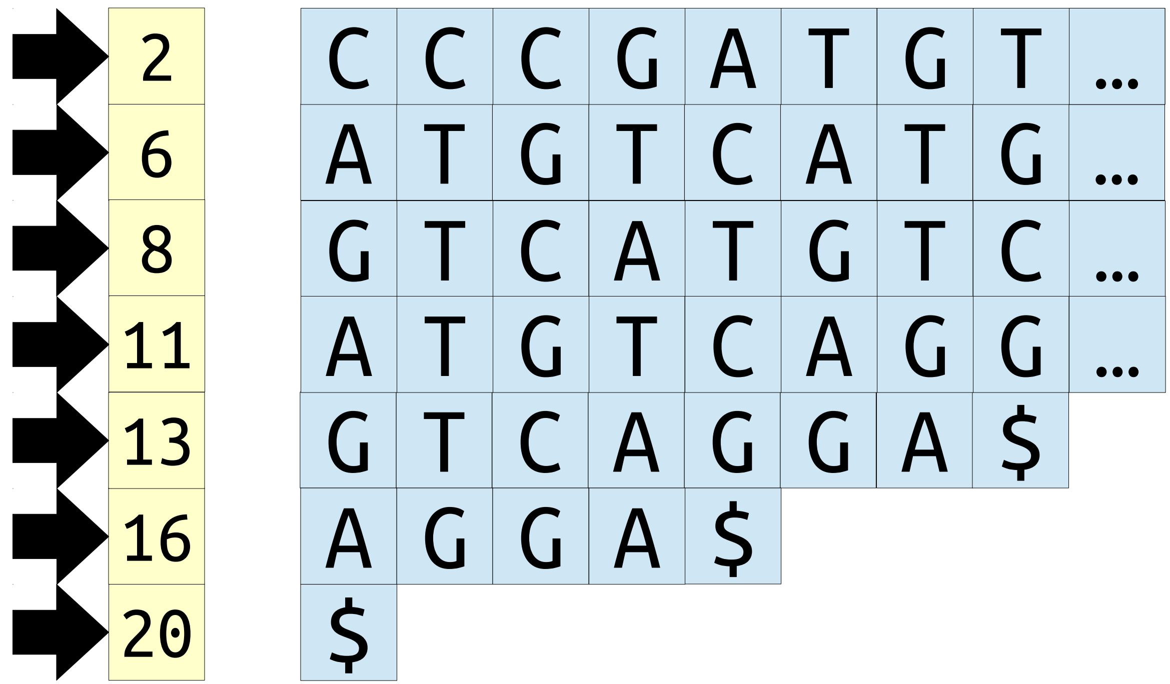
Step One: Scan the array from right-to-left to label each suffix as *S*-type or *L*-type.

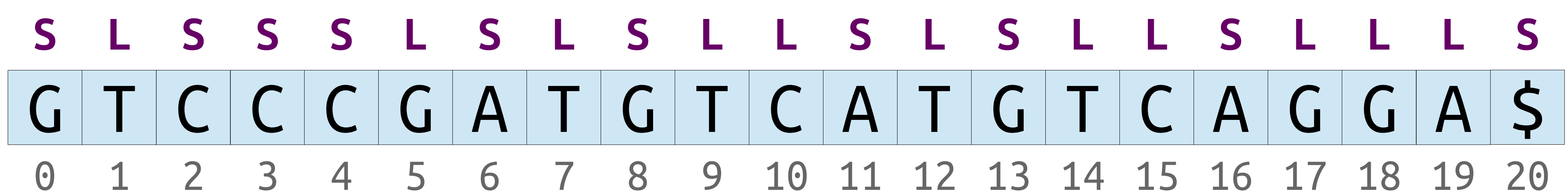
S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

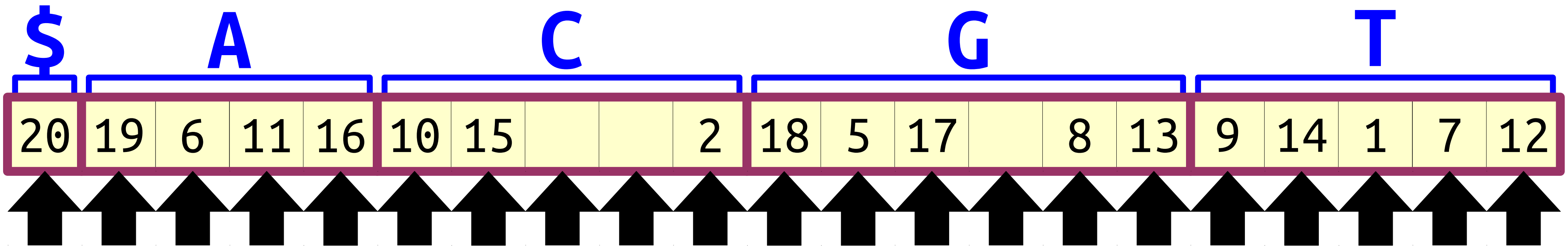


Step Two: Run an induced sorting step on the LMS suffixes in the order they appear. This will get the LMS *blocks* (not the LMS *suffixes*) into sorted order.

Pass One: Place the LMS suffixes at the ends of their buckets.



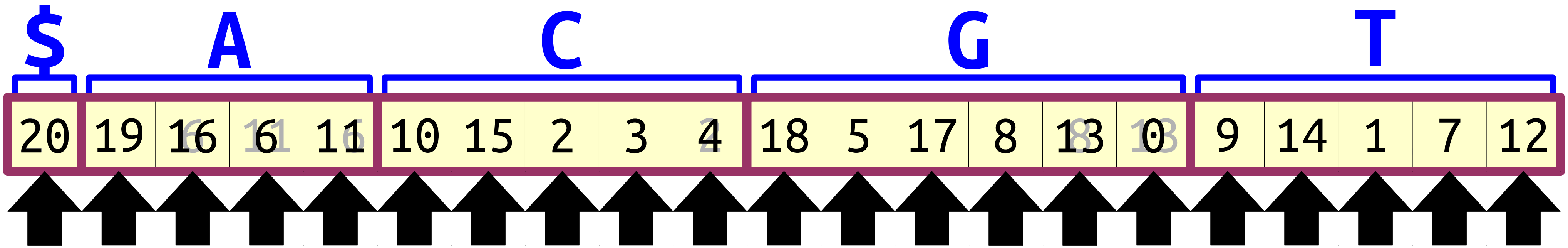




Step Two: Run an induced sorting step on the LMS suffixes in the order they appear. This will get the LMS *blocks* (not the LMS *suffixes*) into sorted order.

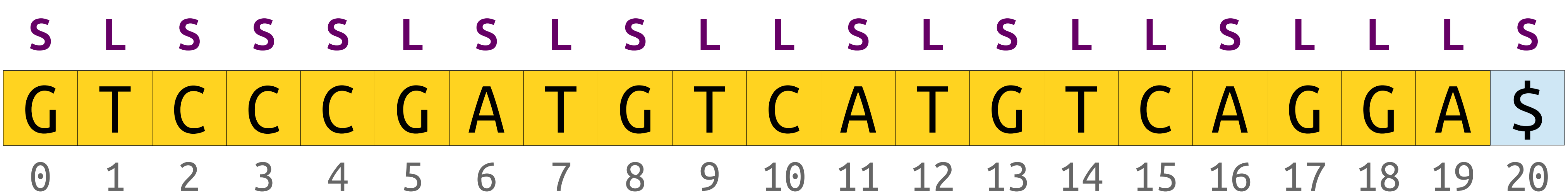
Pass Two: Place the *L*-type suffixes at the fronts of their buckets.

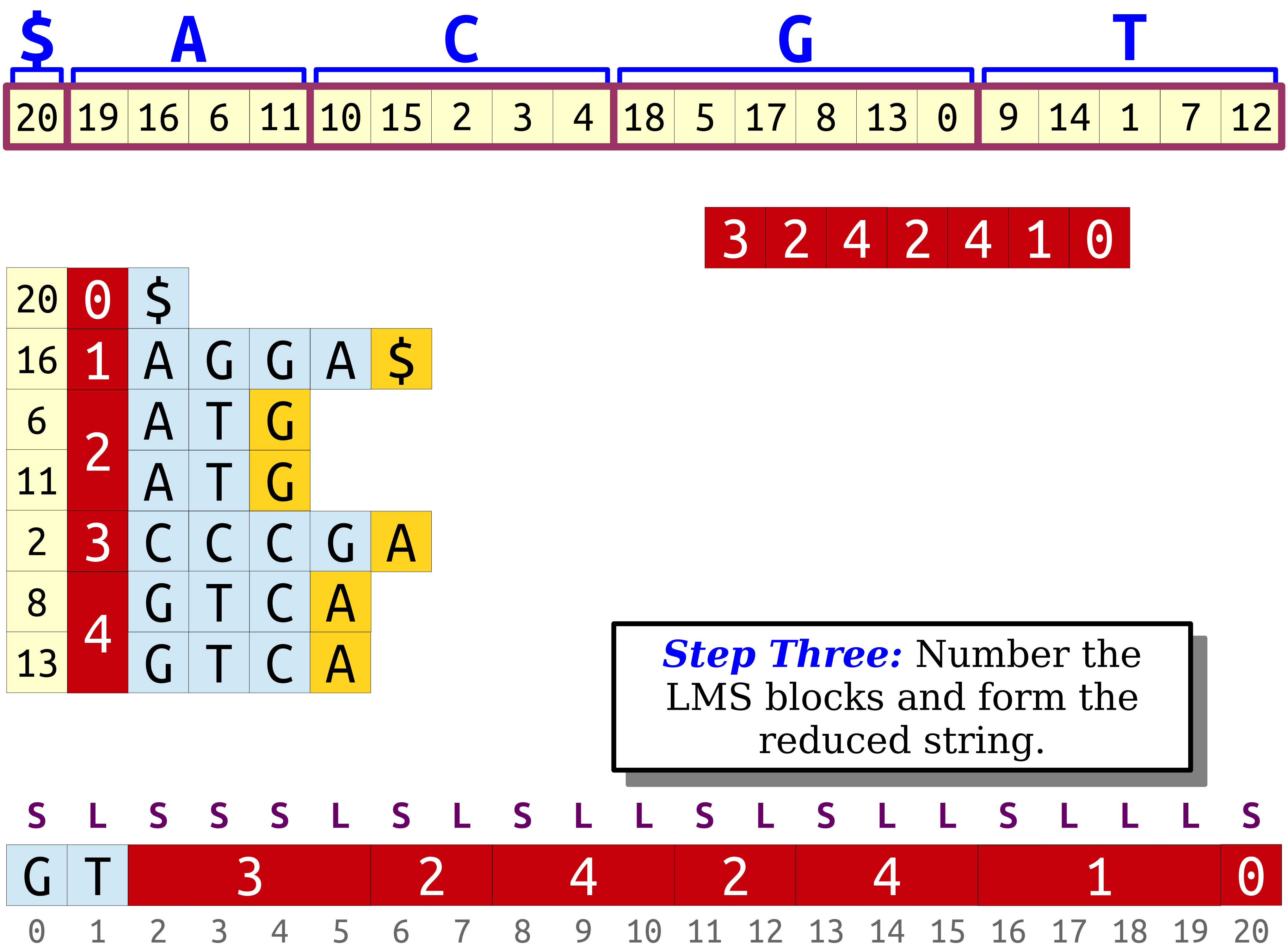
S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



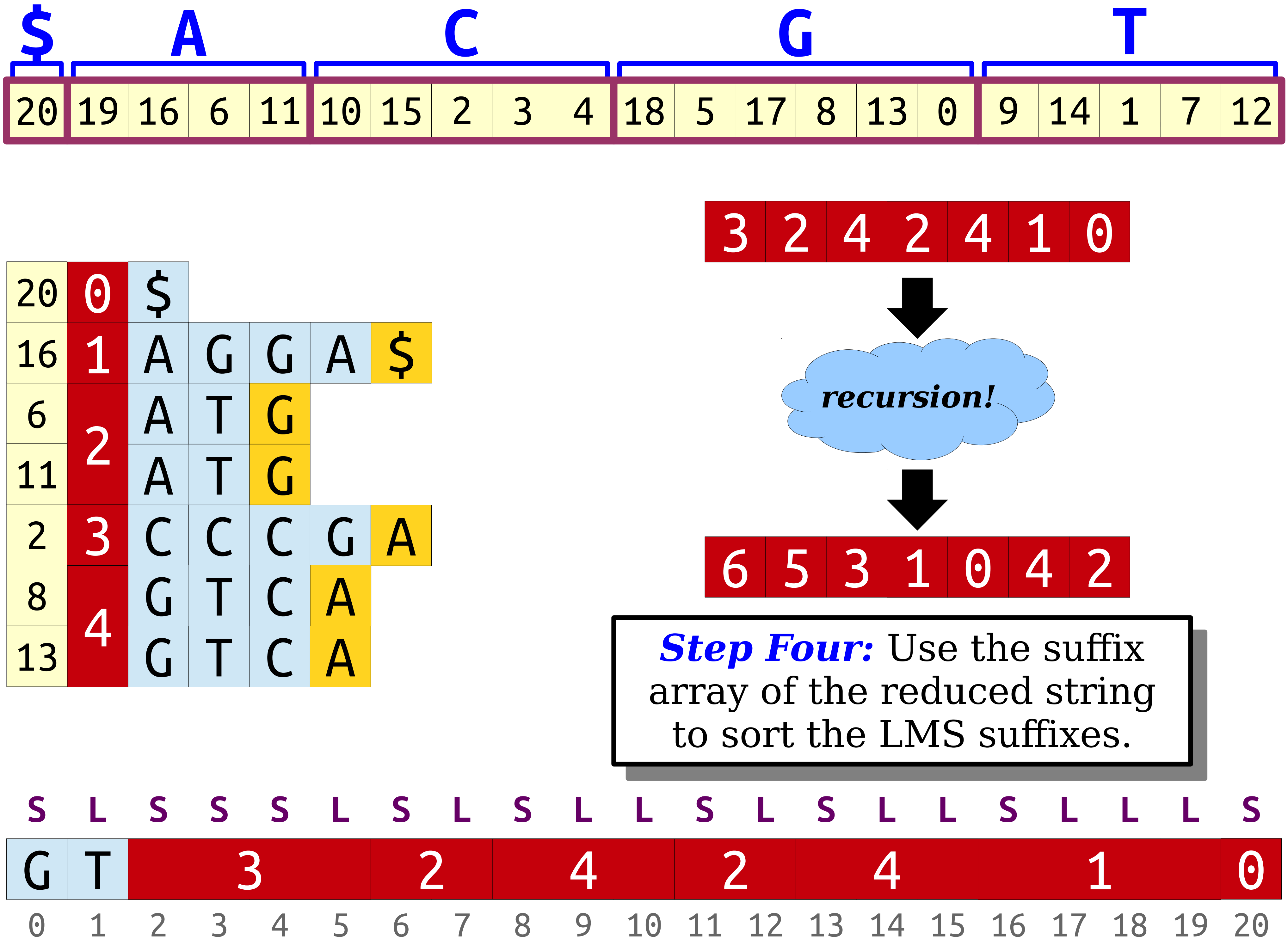
Step Two: Run an induced sorting step on the LMS suffixes in the order they appear. This will get the LMS *blocks* (not the LMS *suffixes*) into sorted order.

Pass Three: Place the *S*-type suffixes at the ends of their buckets.

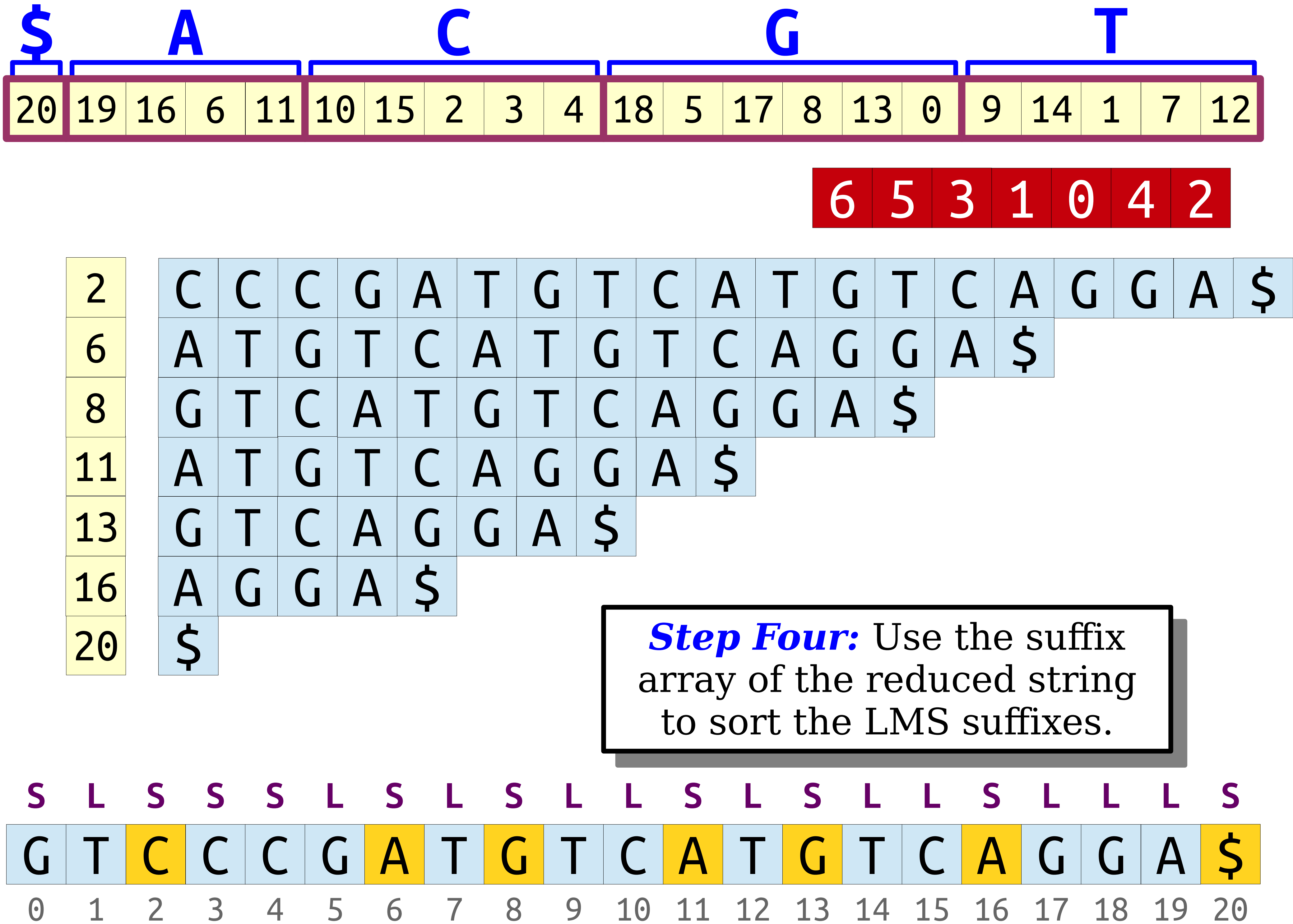




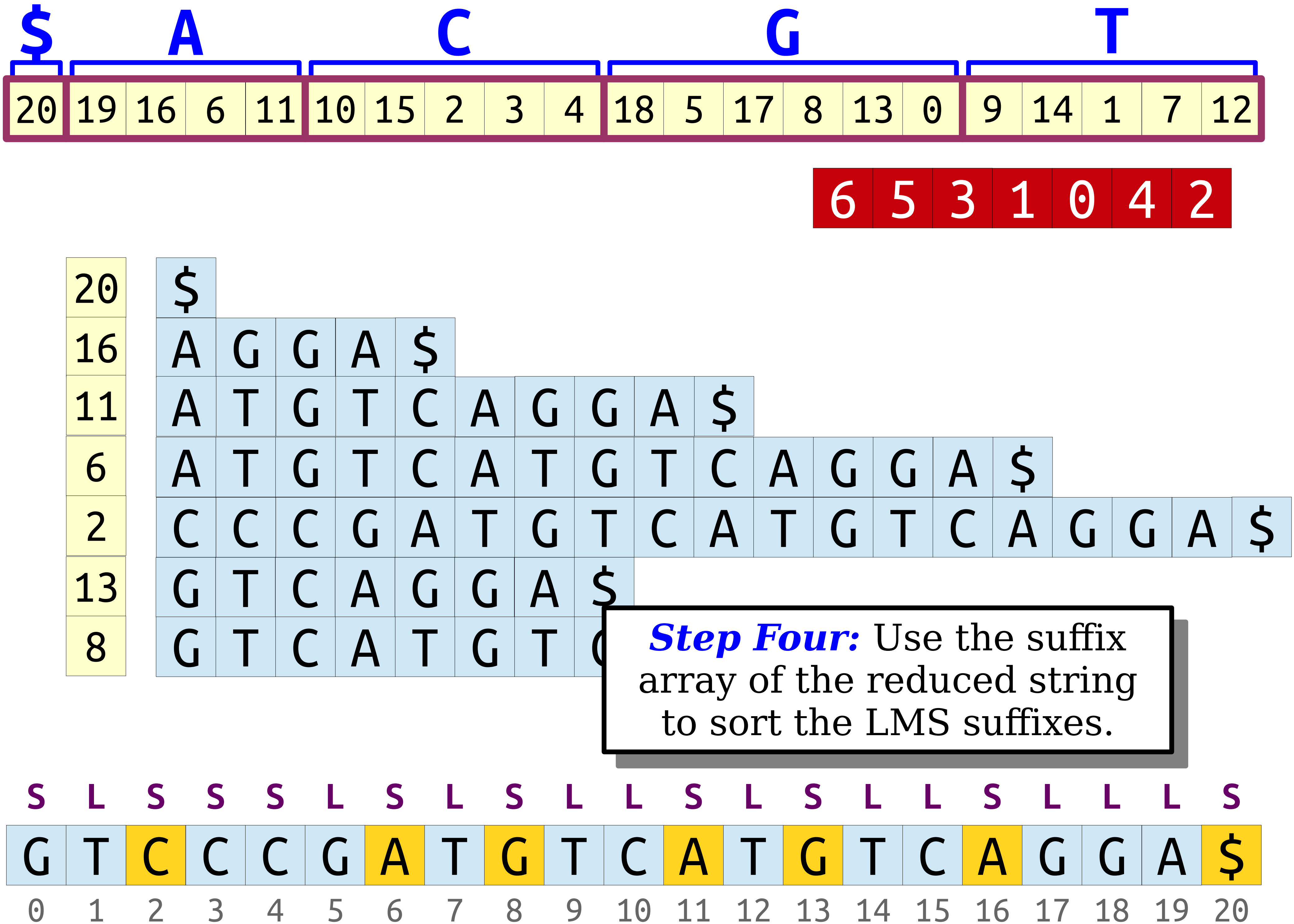
Step Three: Number the LMS blocks and form the reduced string.

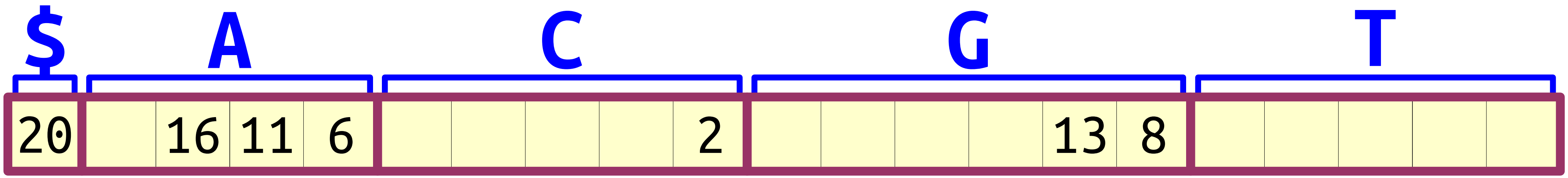


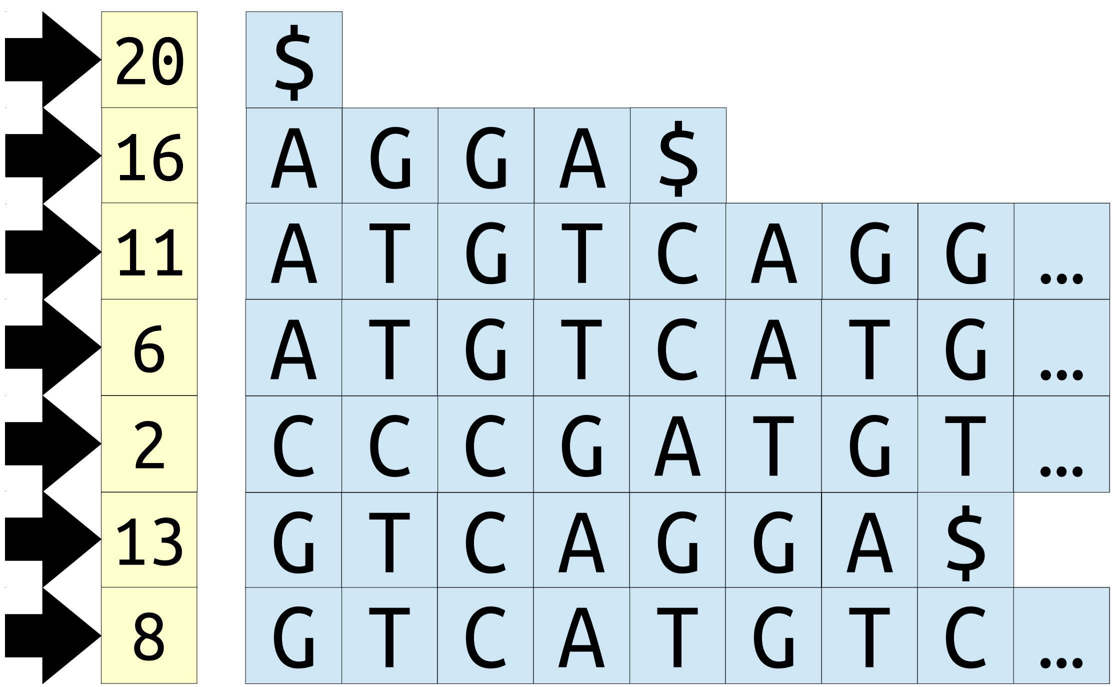
Step Four: Use the suffix array of the reduced string to sort the LMS suffixes.



$ A C G T
20 19 16 6 11 10 15 2 3 4 18 5 17 8 13 0 9 14 1 7 12
6 5 3 1 0 4 2
2 C C C G A T G T C A T G T C A G G A $
6 A T G T C A T G T C A G G A $
8 G T C A T G T C A G G A $
11 A T G T C A G G A $
13 G T C A G G A $
16 A G G A $
20 $
Step Four: Use the suffix array of the reduced string to sort the LMS suffixes.
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

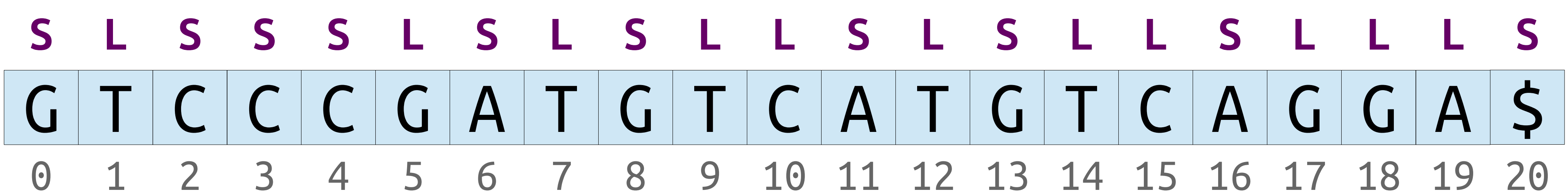

$ A C G T
20 19 16 6 11 10 15 2 3 4 18 5 17 8 13 0 9 14 1 7 12
6 5 3 1 0 4 2
20 $
16 A G G A $
11 A T G T C A G G A $
6 A T G T C A T G T C A G G A $
2 C C C G A T G T C A T G T C A G G A $
13 G T C A G G A $
8 G T C A T G T C
Step Four: Use the suffix array of the reduced string to sort the LMS suffixes.
S L S S S L S L S L L S L S L L S L L L S
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

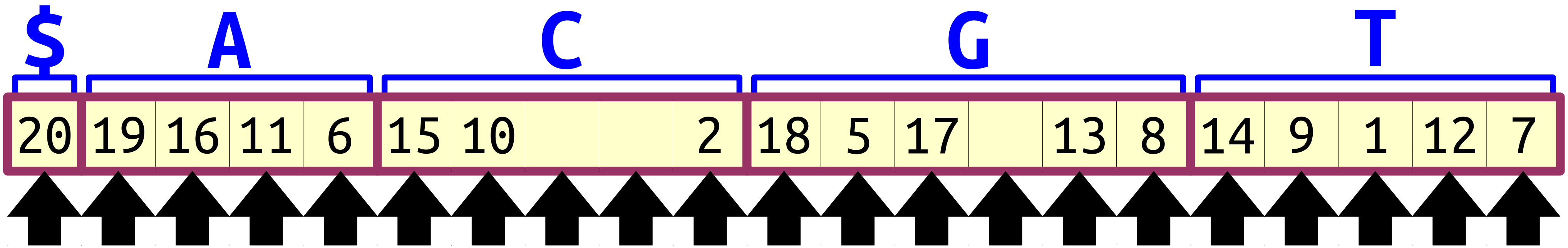




Step Five: Run an induced sorting step on the sorted LMS suffixes to produce the overall suffix array.

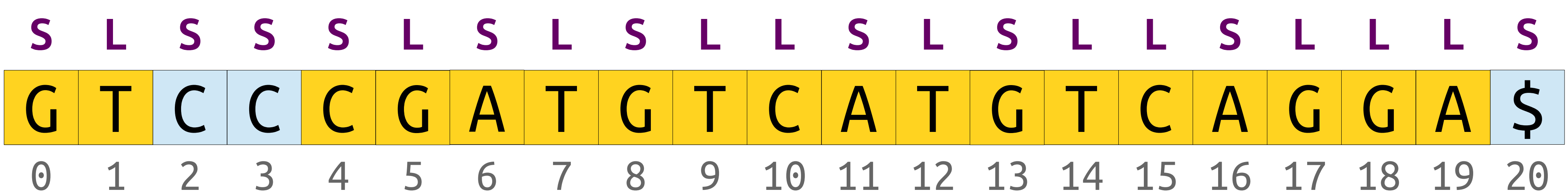
Pass One: Place the sorted LMS suffixes at the ends of their buckets.





Step Five: Run an induced sorting step on the sorted LMS suffixes to produce the overall suffix array.

Pass Two: Place the L-type suffixes at the fronts of their blocks.

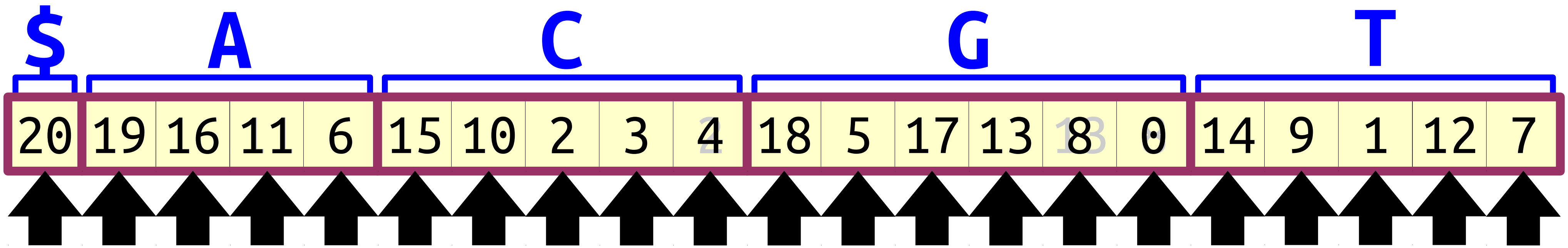


$	A				C					G						T				
20	19	16	11	6	15	10			2	18	5	17		13	8	14	9	1	12	7

Step Five: Run an induced sorting step on the sorted LMS suffixes to produce the overall suffix array.

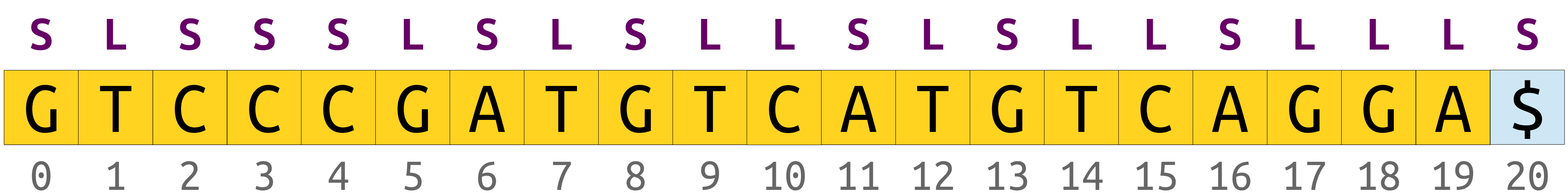
Pass Three: Place the *S*-type suffixes at the ends of their blocks.

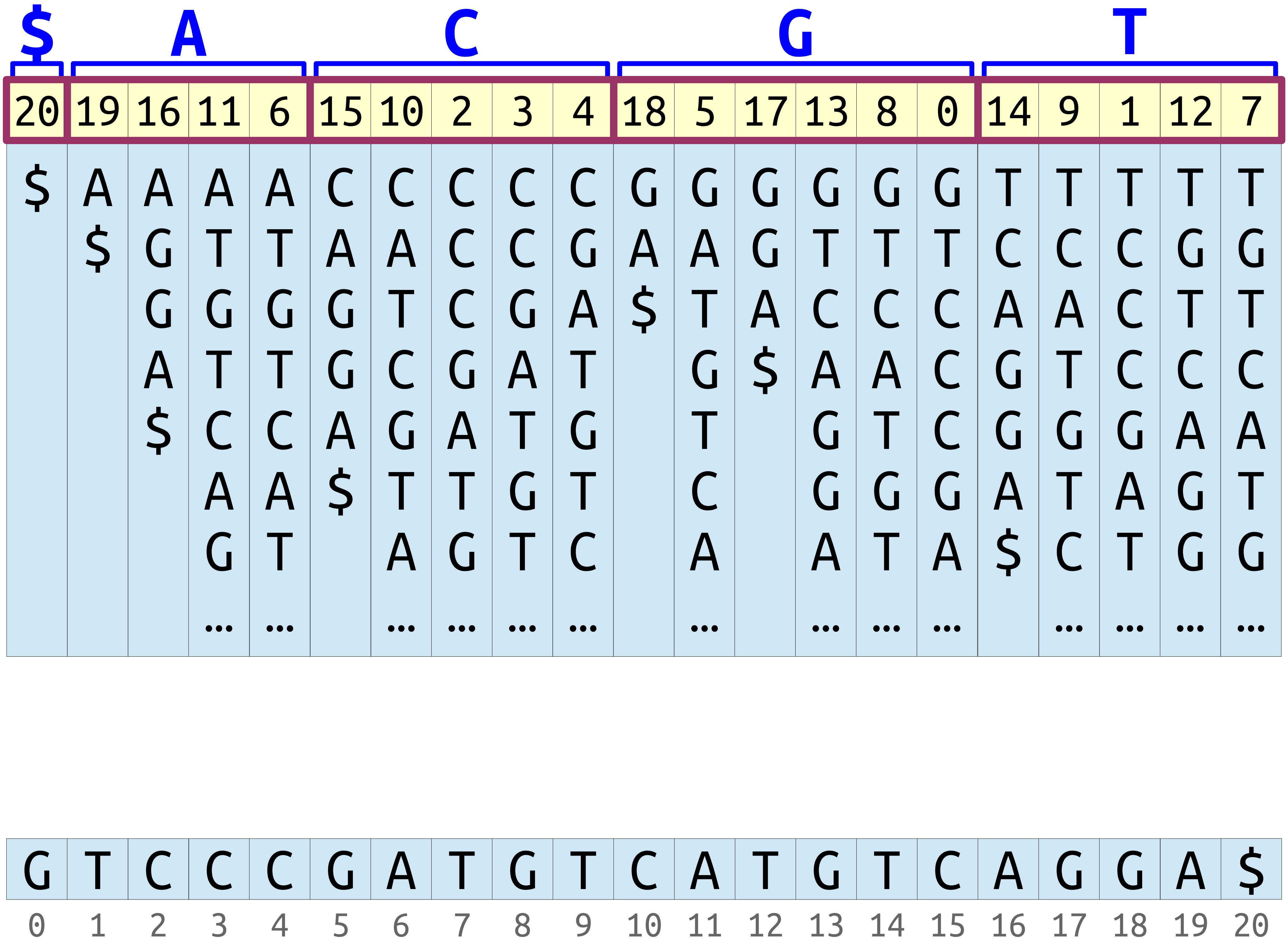
S	L	S	S	S	L	S	L	S	L	L	S	L	S	L	L	S	L	L	L	S
G	T	C	C	C	G	A	T	G	T	C	A	T	G	T	C	A	G	G	A	$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20



Step Five: Run an induced sorting step on the sorted LMS suffixes to produce the overall suffix array.

Pass Three: Place the *S*-type suffixes at the ends of their blocks.




$ A C G T
20 19 16 11 6 15 10 2 3 4 18 5 17 13 8 0 14 9 1 12 7
G T C C C G A T G T C A T G T C A G G A $
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

SA-IS, End-to-End

SA-IS(T):

Scan T from right-to-left to mark each character as S-type or L-type.
Identify all the LMS suffixes of T.

Run induced sorting using the LMS suffixes in the order they appear in in T.

Scan the result, gathering LMS suffixes in the order they ended up in.
Number the LMS blocks, assigning duplicate blocks the same number.
Form the reduced string T' from the block numbers.

If all blocks are unique, get a suffix array for T' by directly inverting T'.
Otherwise, get a suffix array for T' by calling SA-IS(T').

Use the suffix array for T' to sort the LMS suffixes of T.
Do a second induced sorting pass of T using the LMS suffixes in sorted order.

The Overall Runtime

- The SA-IS algorithm does $O(m)$ work, then (optionally) makes a recursive call on the reduced string.
- The size of the reduced string is equal to the number of LMS characters.
- ***Claim:*** There are at most $m/2$ LMS characters.
  - Each LMS character appears when an L-type suffix is followed by an S-type suffix, and in the worst case the suffix types alternate between L-type and S-type.
- Recurrence relation is

$$T(m) \leq T(m/2) + O(m).$$

- Applying the Master Theorem, this solves to $\mathbf{O(m)}$ total work!

Wow! What a nifty algorithm!

In Practice

- SA-IS is extremely fast in both theory and in practice.
  - Excellent locality of reference in the induced sorting and block numbering steps.
  - Recursive step usually has a great compression ratio.
- With a creative implementation, the memory overhead is minimal.
  - There's further work beyond what's shown here about reducing the total memory usage by being clever and recycling space.
- The current fastest suffix array construction algorithm, DivSufSort, is essentially a highly optimized version of SA-IS using a slightly different approach to sorting LMS suffixes.

Why Study SA-IS?

- ***Explore the theoretical structure of suffix arrays.***
  - The relative ordering of L-type and S-type suffixes, the idea of induced sorting, and the bit about LMS blocks are all really beautiful theoretical results.
- ***See the idea of simulating one algorithm with another.***
  - Induced sorting is basically a multiway merge sort implemented really well, yet there's little evidence of this in the final code!
- ***Look at a really, really clever divide-and-conquer algorithm.***
  - Did you expect to see the suffix array reduced that way?
- ***Probe the interface between theory and practice.***
  - This algorithm has an asymptotically optimal runtime, *and* it's really fast in practice!

More to Explore

- ***Constructing LCP using induced sorting***. (Fischer and Kurpicz, 2011)
  - Kasai's LCP algorithm was the first linear-time LCP algorithm. Turns out you can augment SA-IS to produce both the suffix array and the LCP array much, much faster than this.
- ***Reducing SA-IS memory usage***. (Nong, 2013)
  - A variation of SA-IS (by one of its original authors!) that cuts down on the memory usage and improves performance.

Next Time

- ***Balanced Trees***
  - Fast, flexible data structures for sorted sequences.
- ***B-Trees***
  - Built for databases, now popular in RAM!
- ***2-3-4 Trees***
  - One of the simplest balanced trees around.
- ***Red/Black Trees***
  - Where do they come from?