Lecture 22: Induced Sorting Suffix Array (SA-IS) Part 2 (Updated after class)

Sam McCauley

December 3, 2021

Williams College

Admin

- Assignment 8 ready; I added a bunch more scaffolding than previous assignments
- Anything I give in `sais.c` is optional.
- SA-IS lecture notes on the website
  - I tried very hard to get rid of typos, but please let me know about any you find!
- May have to go a bit quicker today to make sure we get through everything
- I'll come back to each topic to help prove why it works and give examples. (May do some of this on Monday, but I want to at least get through the algorithm today.)

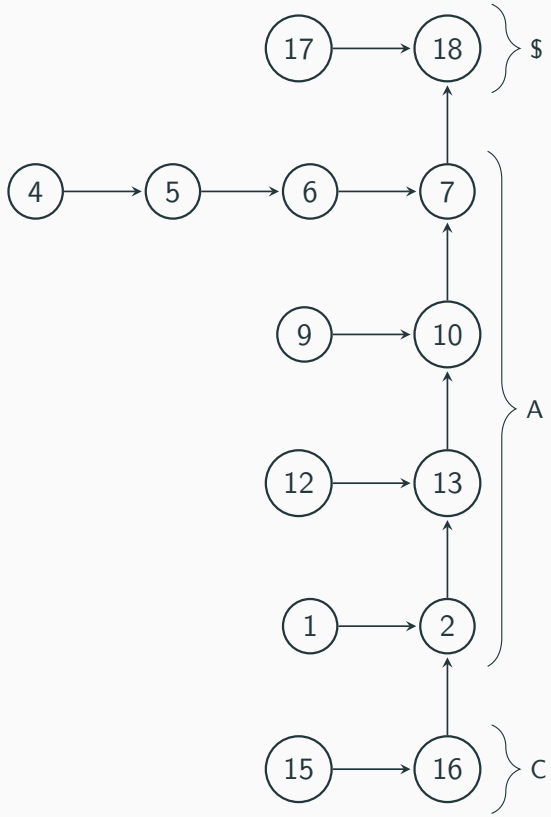
Where we were

- L-type and S-type suffixes; L-type is LARGER than next suffix
- LMS suffixes are S-type suffixes preceded by L-type suffix
- Goal for the moment: if we have a sorted list of LMS suffixes, we can sort L-type suffixes
- Recall that suffixes are in the final suffix array in order of their first character.

sortedP:

18	7	10	13	2	16

We know each *L*-type suffix is larger than the suffix that comes after it. We can represent all of these dependencies graphically. The rightmost column of nodes consists of the LMS suffixes.



String *S*:

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

Idea: use the first character and type

- Look at the first character c of the suffix, and the type of the suffix, to determine where it will go
- From last time, three cases:
  - If there are no L-type suffixes starting with c in the right column, add this one before any S-type suffixes starting with character c
  - If there is an L-type suffix starting with c in the right column, it was added earlier in this process (since our column started with all S-type suffixes in the right column). This means we should place our new L-type suffix lower in the right column (later in lexicographic order)
- In all cases: place our new suffix *before* all S-type suffixes starting with C, and *after* all L-type suffixes starting with C

Final ordering

18	7	10	13	2	6	9	12	5	16	1	15	17	4

The LMS suffixes and *L*-type suffixes in sorted order after we repeatedly perform the operation we discussed: remove the guaranteed minimum node from the top of the right column. If it has any *L*-type predecessors (to the left), place them in the right column using their character and type.

Induced Sorting

Problems with This Approach

- The above gives us an $O(n)$ time method to get the correct order of the L-type suffixes given the ordering of the LMS suffixes.

- Two problems remain.
  - First, we need to also order the S-type suffixes.
  - Second, the above method doesn't really work as-is. How do we insert the suffix? Can we avoid this annoying tree structure?

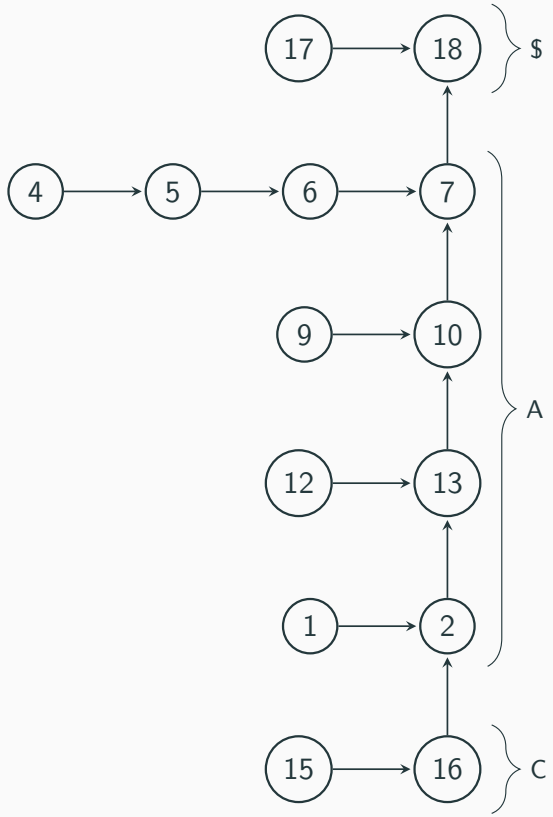
Warmup

- Let's improve this algorithm just a little bit: we'll remove the right column of pointers
- How can we keep track of the right column with just an array?
- Arrays are bad for inserts because we may need to shift items down. How can we avoid all shifts?

sortedP:

18	7	10	13	2	16

We know each *L*-type suffix is larger than the suffix that comes after it. We can represent all of these dependencies graphically. The rightmost column of nodes consists of the LMS suffixes.



String *S*:

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

Warmup Solution

- Let's allocate an array of size n, and keep track of where each bucket (corresponding to a letter) begins
- Each time we insert a new suffix to the right column, add it to the leftmost remaining spot of the corresponding bucket
- And increment the counter
- (We're not done—still need to keep track of the pointers coming from the left. . . . or do we?)

Induced Sorting: Magic of the SA-IS algorithm

- Notice that the right-pointing edges in Figure 8 come immediately from string S and the type of each suffix.
- Let's run exactly the same algorithm—but now we *won't keep track of anything but the final suffix array*.
- Instead, we'll take advantage of the fact that they're placed greedily to put them in the correct place in the suffix array as they come.

Induced Sorting Algorithm: Place LMS Suffixes

Create a suffix array SA, initializing all array elements to -1. Calculate where the "bucket" for each letter begins and ends.

Then, place the sorted LMS suffixes, in sorted order (from our given array $sortedP$), as far to the right as possible in their bucket. For each index i of SA (from left to right):

- If $SA[i] = -1$, ignore this slot
- Otherwise, place the predecessor suffix if it exists and is of type L

Induced Sorting Algorithm: Place L-type suffixes

Iterate through SA from left to right. If the preceding suffix is L-type, place it as far to the *left* as possible in its bucket

For each suffix array index i from 0 to $n-1$:

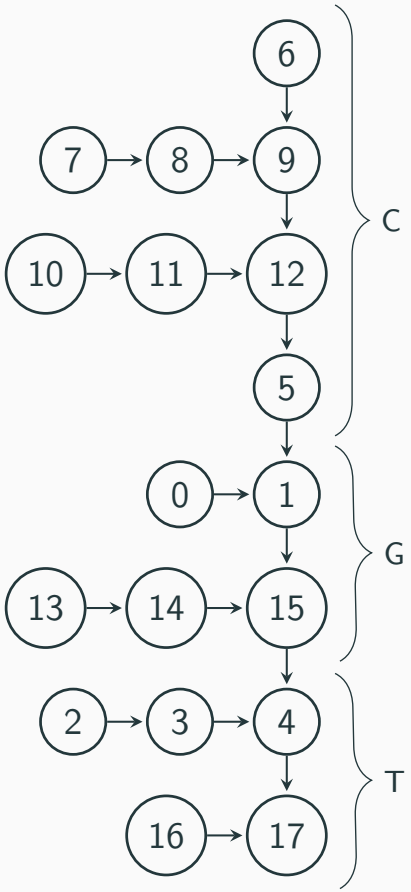
1. if i is not empty (i.e. $SA[i] > 0$) and index $SA[i]-1$ is of type L:
    - Place suffix $SA[i]-1$ in the next (leftmost) empty slot in its bucket (i.e. in bucket $S[SA[i]-1]$.)

Finishing the Remaining Suffixes

- We need to write *S*-type suffixes
- What do we know about each *S*-type suffix?
    - It comes *before* its successor in sorted order
- We know the sorted order of the *L*-type suffixes
- Doesn't that lead to the exact same kind of diagram as before? (But with *L*-type suffixes in the right column, and each *S*-type suffix being *smaller* than the next)

Sorted L and LMS suffixes:

18	7	10	13	2	6	9	12	5	16	1	15	4	17



String S:

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

Let's do the same thing with this diagram

- Take off the node at the *bottom* of the rightmost column; this is the *largest* suffix remaining (these are both reversed from what we did before)
- Can place that suffix at the end of the current suffix array
- May have a left pointer (predecessor). What do we do with that?
    - This suffix is *larger* than any L-type suffix starting with the same character
    - This suffix is *smaller* than any previously-placed S-type suffix starting with the same character

Induced Sorting Algorithm: Place S-type suffixes

Iterate through SA from *right to left*. If the preceding suffix is S-type, place it as far to the *right* as possible in its bucket

For each suffix array index i from $n-1$ to 0:

1. if i is not empty (i.e. $SA[i] > 0$) and index $SA[i]-1$ is of type S:
    - Place suffix $SA[i]-1$ in the next (rightmost) empty slot in its bucket (i.e. in bucket $S[SA[i]-1]$.)

Placing S-type suffixes after L-type suffixes

Two problems to consider with this:

- Some of the S-type suffixes are already in SA. We can remove them before we start.
- But, it turns out we don't need to. Instead: just recalculate the ends of each bucket
- Then we will overwrite the LMS suffixes that were already placed
- Is it a problem if our loop reaches an LMS suffix (with index i) before it's overwritten?
  - No! The preceding suffix must be L-type; so we just ignore it anyway.

Putting it together

- One we place the S-type suffixes, we are done! (Let's see an example)
- So we placed the LMS suffixes. Then, we placed the L-type suffixes. Then, we placed the S-type suffixes. Each took a linear scan.
- Let's look at our final Induced Sort: an $O(n)$ algorithm to obtain the suffix array given the sorted LMS suffixes.

```
InducedSort:
    Create a count of each character in array C
    store the ends of each bucket in E using C
    store the beginning of each bucket in B using C
    for i from sizeOfP − 1 to 0:
        firstChar ← S[sortedP[i]]
        SA[E[firstChar]] = sortedP[i]
        E[firstChar]−−
    for i from 0 to n−1:
      if SA[i] > 0 and t[SA[i]−1] = L:
        prevIndex ← SA[i] − 1
        firstChar ← S[prevIndex]
        SA[B[firstChar]] ← prevIndex
        B[firstChar]++
    store the ends of each bucket in E using C
    for i from n−1 to 0:
      if SA[i] > 0 and t[SA[i]−1] = S:
        prevIndex ← SA[i] − 1
        firstChar ← S[prevIndex]
        SA[E[firstChar]] = sortedP[i]
        E[firstChar]−−
```

Recursively Sorting the LMS Suffixes

Our goal

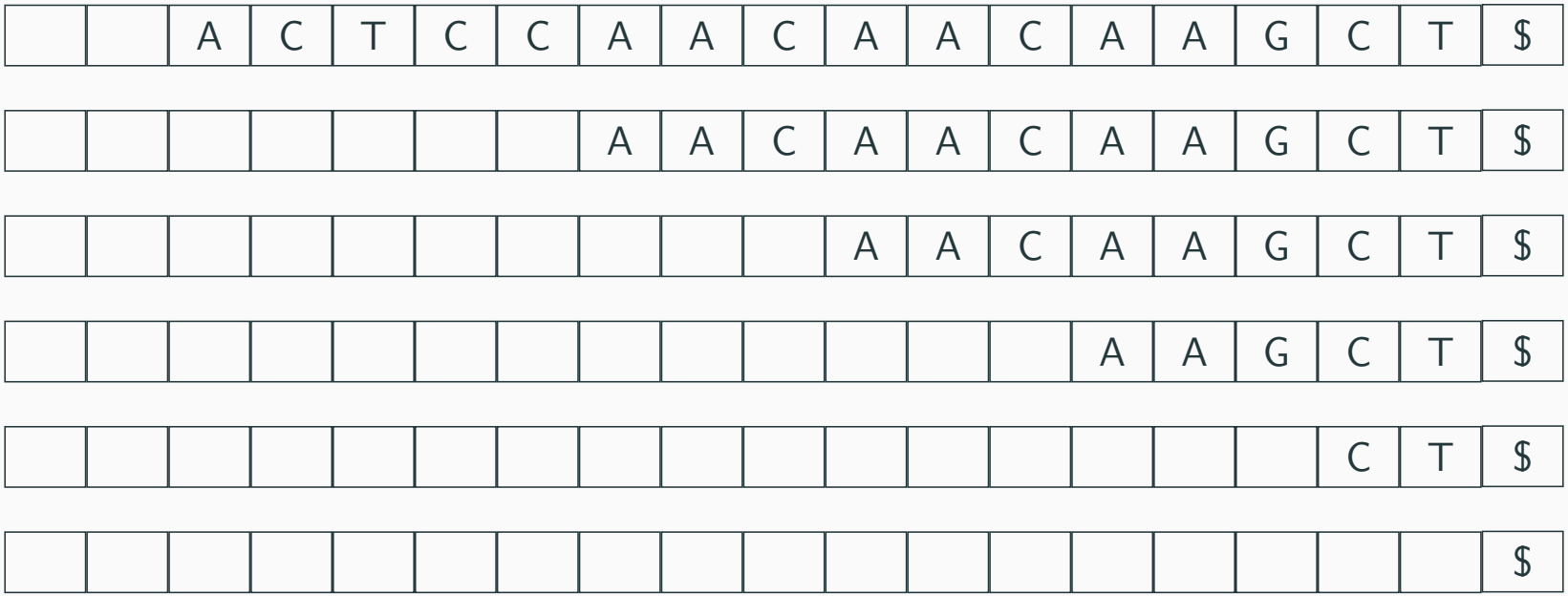
- Our only task now is to sort the LMS suffixes. If we can do that, we are done.

- We can obtain, in $O(n)$ time, the *indices* of the LMS suffixes (not in sorted order): we label the types of the suffixes of S, and then find the LMS suffixes using a linear scan.

- Let's say we store them in an array `P`. So, rephrasing, our goal is to sort the suffixes stored in `P` to obtain `sortedP`.

Using Recursion?

- I've said before that we're going to solve this problem recursively
- A recursive problem would be: "find the suffix array of a ($-terminated) string S' "
- Instead, we want to find the ordering of a subset of the suffixes (the LMS suffixes)


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

Our Suffixes

- These aren't the suffixes of a string S'
- But they are suffixes of each other
- Problem: a *sequence* of characters between each suffix rather than a single character
- Can we replace these sequences with single characters while retaining the string ordering?

Goal: group these sequences of characters into single characters, while retaining the ordering between these strings.

LMS Blocks

Definition 1

We define a *LMS block* to be a substring of S from i to j (i.e. $S[i], S[i+1], \ldots, S[j]$) where i and j are both LMS suffixes. The final character of S, by itself, is also considered to be an LMS block.

Note that the blocks, as defined, overlap.

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

LMS blocks of this string: `ACTCCA`, `AACA`, `AACA`, `AAGC`, `CT$`, `$`.

Plan for sorting LMS Suffixes

- We need to assign characters to the LMS blocks
- But need to make sure that we retain the sorted order (smaller blocks get smaller characters)
- So we need to sort the LMS blocks
  - Just the blocks—not the whole LMS suffixes!
  - We'll have some "ties" that we'll need to resolve with the recursive call

Real Magic of the SA-IS algorithm

- We want to obtain the order of the LMS blocks
- Plan: run Induced Sort on S and P *without assuming the LMS suffixes in P are sorted*
- We saw that running Induced Sort on S and *sortedP* will give the correct suffix array SA
- Claim: running Induced Sort on S and P (where P contains LMS suffixes in the order they appear in S) will lead to an SA such that the LMS *blocks* in SA are in sorted order.

Taking a step back: what are these two Induced Sort calls?

- To take a step back, each recursive call of the SA-IS algorithm is going to use two induced sorts.
  - First, an induced sort to get the LMS blocks in sorted order
  - Then a recursive call on a string S' built using the sorted LMS blocks (haven't discussed this yet)
  - The recursive call will make sure that the LMS suffixes (not just the LMS blocks) are entirely sorted.
  - Once the recursive call completes, the LMS suffixes will be in sorted order, and a new induced sort will give the correct suffix array.

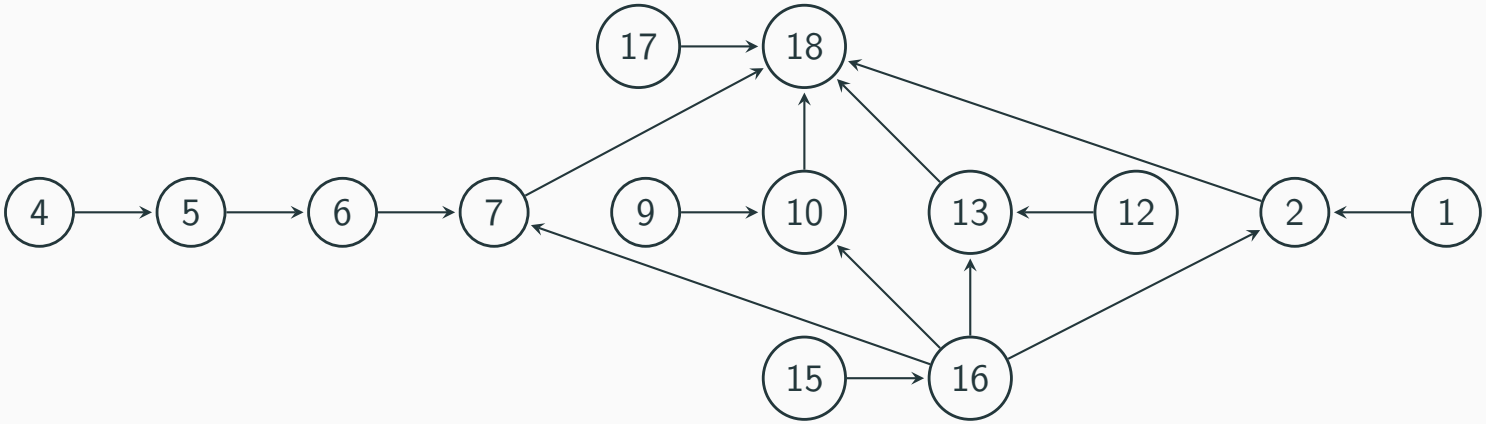
Let's work through an example and discuss

What happens when we run an induced sort using an unsorted set of LMS suffixes P?

Let's start an example on the board. To begin, we place P in its correct bucket.

- Now, suffixes will not be sorted correctly within a bucket
- ...But they're not entirely unsorted. What CAN we say about them?
- They are correctly sorted by their first character. (Only.)
- Can we use this to learn something about the L-type suffixes?

L-type dependencies with unsorted P



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

We have dependencies between the first characters of the LMS suffixes, and the L-type suffix dependencies are exactly as before. But we no longer have a clean "rightmost column"; just a grouping by character.

L-type suffix guarantee (unsorted P)

- Each L-type suffix i is in the correct place based on suffix $i + 1$.
- If $i + 1$ is an LMS suffix, all we know is that the first two characters of L are sorted
- . . . with an induction, the L-type suffixes are sorted up to the next LMS suffix.

Example of placing L-type suffixes (unsorted P)

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

Claim: each L-type suffix is sorted up to the next LMS suffix.

Example: Suffix 6 is `CAAC...`; suffix 12 is `CAAG...`. Suffix 6 winds up earlier than suffix 12, even though it's later in sorted order. But the second character of each of these suffixes is from an LMS suffix—so we only are sorting `CA` from suffix 6 with `CA` from suffix 12.

Finishing Induced Sort with Unsorted P

- Now place the S-type suffixes as before
- (Remember to reset the array ends E!)
- Since L-type suffixes are sorted up to the next LMS suffix...
- S-type suffixes are sorted up to the next LMS suffix
- LMS suffixes are S-type. So the LMS *blocks* are sorted!
- Scan through SA and add each block to `sortedLMS`
- Should get: 18, 7, 10, 13, 2, 16

Creating the Recursive String

Where we are

- We can sort the LMS blocks using Induced Sort
- **We are here:** Using the sorted LMS blocks, can obtain a string S' such that the suffix array of S' gives the sorted order of the LMS suffixes (*sortedP*)
- Using *sortedP*, an Induced Sort gives us the suffix array of S

Goal: group these sequences of characters into single characters, while retaining the ordering between these strings.

Assigning Characters to LMS blocks

- We know the order to assign them (sorted order!)
- First LMS block gets character 0
- Then: if two LMS blocks are the same, want them to have the same character
- Otherwise, want them to have a different character
- Only need to compare consecutive LMS blocks but...
- How can we compare them all in $O(n)$ time?

Observation

- LMS blocks have *total size* $O(n)$
- So we just compare them character by character
- **Note: need to compare the character AND the type of the LMS blocks. If either mismatches they are different.**
- Create array `blockAssignments`, where `blockAssignments[i]` contains the new character for the LMS block stored in `sortedLMS[i]`

Comparing two LMS blocks i and j

Start with $c = 0$. For each c, test (incrementing c after each iteration):

- If $S[i + c] \neq S[j + c]$, the blocks are unequal
- If $t[i] \neq t[j]$, the blocks are unequal
- If we have reached (and tested!) another LMS suffix we are done and the blocks are equal

Creating block assignments

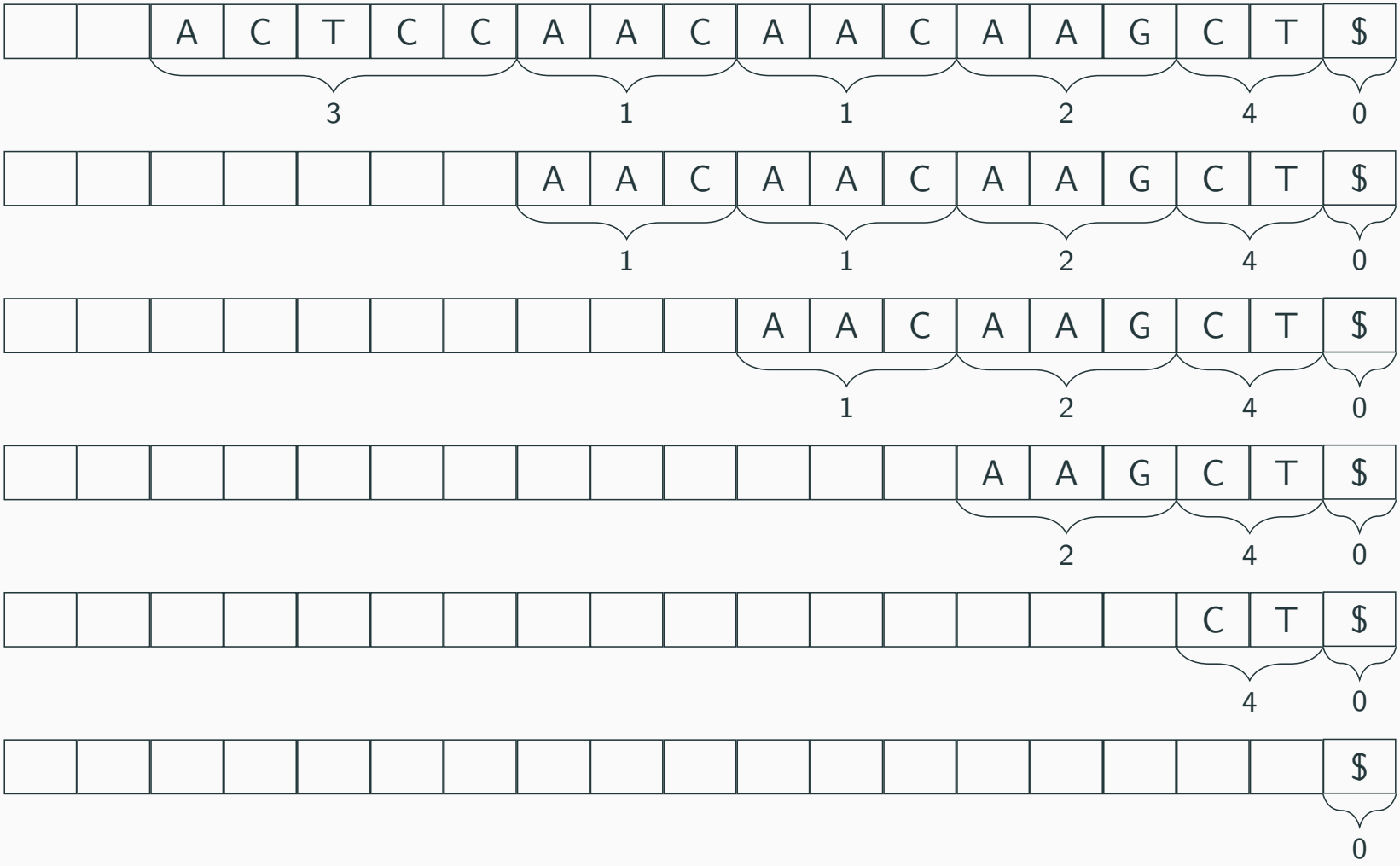
`blockAssignments[0] = 0`

For $i = 1$ to size of `blockAssignments` -1:

- If LMS blocks `sortedLMS[i]` and `sortedLMS[i-1]` are equal, `blockAssignments[i] = blockAssignments[i-1]`
- Otherwise, `blockAssignments[i] = blockAssignments[i-1] + 1`

Where we are

- `sortedLMS`: 18, 7, 10, 13, 2, 16
- `blockAssignments`: 0, 1, 1, 2, 3, 4
- Let's look at what happens (visually) when we replace these LMS blocks with these new characters



Strings to sort: 311240, 11240, 1240, 240, 40, 0

Recursive call?

- We now want to sort the suffixes of 311240
- That's a recursive call!
- Create $S' = 311240$ and find its suffix array
- Size of S'?
  - $|S'|$ is the number of LMS suffixes; this is at most $n/2$ (why)?
- Is the smallest character in S' the last character?
  - Yes! The smallest LMS block is always the last character itself; so that will always be the last block and will always be assigned 0

Making the Recursive Call

Surprisingly, here is the base case of the SA-IS algorithm.

- Let's say that there are no identical LMS blocks. What can we say about the suffix array of S'?
- The characters in S' are $\{0, 1, \ldots, |S'| - 1\}$
- So each character names its index in sorted order
- we can write each character of S' to the correct spot in SA' using a linear scan: $SA'[S'[i]] = i$
- Don't need a recursive call!
- (Why are we guaranteed that this eventually happens?)

Game Plan for sorting LMS Suffixes

- Sort the LMS blocks using an Induced Sort
- Scan the LMS blocks in sorted order, comparing consecutive LMS blocks. Use this to create an assignment of new characters to blocks.
- Use this assignment to create a new string S'
- If the last assigned character was $|S'| - 1$, then $SA' = S'$. Otherwise, calculate SA' recursively.
- Use SA' to sort the LMS suffixes

Constructing S'

- $S'[i]$ corresponds to the ith LMS block (i.e. index $P[i]$)
- We have: for all i, the LMS block at `sortedLMS[i]` should obtain character `blockAssignments[i]`
- One way to do this (maybe not best)?:
  - Create an array `pAssignments` of size $O(n)$
  - For each i, set `pAssignments[sortedLMS[i]] = blockAssignments[i]`
  - Then, for each i, set $S'[i] =$ `pAssignments[p[i]]`

Final step!

Need to use SA' to obtain `sortedP` (the LMS suffixes in sorted order)

- The LMS indices (in order in S) are exactly the suffixes of S'
- So: SA' gives us the ordering!

For each index i of P:

- `sortedP` $= P[SA'[i]]$.

Now that we have `sortedP`, a final InducedSort will give us the correct suffix array.

Let's go over what's happened

- Let's look at all of the pseudocode in the lecture notes, and talk a little bit about what each piece does
- Then, if we have time, let's do an example from beginning to end. Perhaps `baannaannaa`.
- Then talk a bit about the assignment