Lecture 21: Induced Sorting Suffix Array (SA-IS) Part 1

Sam McCauley

December 3, 2021

Williams College

Admin

- Assignment 8 released soon (last one!)
- Mostly coding
- Interest in leaderboard?
- I'll be using both boards (on other side of the screen) today so make sure you can see all of them; let me know if you can't read something

Plan for today

- Algorithm engineering is a bit more concrete than what we've seen in this class
- Many results are not just high-level ideas, but have detailed instructions of what data structures to use and how to use them
- This topic: let's look at a state of the art approach to solving a complicated and computationally difficult problem

SAIS Algorithm

- We're going to spend two days on this
- Number of lines of code is actually not bad at all. (Doesn't have lots of crazy pointer stuff either, and not too many off-by-one issues.)
- Tricky part: this algorithm is magic on top of magic. I want to pull back the curtain and have it be understandable
- Plan of how to do this:
  - Lots of examples! Lots of intuition as to how we're doing things. I'm going to try to give lots of resources.
  - Ask questions!
  - There are going to be some "if this character is smaller than this character, than we know this character is smaller than..." pieces (like we saw in BWT). Try to stay awake! We're going to wind up in a very cool spot at the end.

Behind the scenes

- Original plan for this course was the DC3 algorithm: a bit less magic, but a pain to implement, and not that fast
- Earlier in the semester I found the SA-IS algorithm. It's much faster AND much simpler.
- So far my SA-IS implementation is 10x faster than my "simpler" suffix array implementation on large strings (and about the same length).
- Only downside: conceptually not obvious
- (BTW: make sure you set aside time to come to office hours if necessary. Also, we didn't do SA-IS last time; while I'll go over this with our wonderful TA Chris he may be less experienced compared to other topics.)

Some String Preliminaries

Notation

- A string is a sequence of characters. Each character of a string must be from some *alphabet.* Oftentimes, especially in `C`, this alphabet is the set of all 256 ASCII characters (though we'll need to consider more general alphabets today).
- We'll use array indexing to refer to specific characters in a string, and assume that they are 0-indexed. So if string S stores `hello`, we have $S[0] = \texttt{h}$ and $S[1] = \texttt{e}$.
- A *substring* of a string S is a sequence of consecutive characters of S. If a substring starts at character 0 it is a *prefix* of S; if it ends at the last character it is a *suffix* of S. We call the suffix starting at character i the *ith suffix* of S. (So, for example, the 0th suffix of S is S itself; the 1st suffix consists of S with its first character removed.)

Terminal character

- Just like in BWT transform! Will be both useful and important for our algorithm
- We assume that the character $ does not appear in the alphabet; furthermore we assume that all strings end with $.
- (Of course, this assumption is easy to maintain by appending $ (or some other non-alphabet character) to any string that does not satisfy the assumption.)
- We consider $ to be before any other character in sorted (i.e. lexicographic) order.
- On your assignment you will be given strings that end with character `\0`, so this assumption will be already satisfied for all strings you are given in your code.

Suffix Arrays

Pattern matching

- One of the most ubiquitous string problems involves searching for occurrences of one string in another.
- In particular, let's say you have a large text T. You want to preprocess T (using at most $O(|T|)$ space) so that for any pattern P, you can quickly determine if P is a substring of T.
- Let's say that $|T| = n$ and $|P| = m$. How fast can we solve this?
- Here's one way we could solve this problem. Let's sort all the *suffixes* of T and store them in an array.
- So if T was `banana`, our suffixes would be, in sorted order: `a`, `ana`, `anana`, `banana`, `na`, `nana`.
- The bad news is that this requires $O(n^2)$ space—but the good news is that we can use binary search to answer queries very quickly!

Pattern matching using suffixes

- Let's say P occurs at character i of T—so $T[i] = P[0]$, $T[i+1] = P[1]$, and so on until $T[i+m-1] = P[m-1]$. But this means that the first m characters of suffix i of T match P. That is to say: P occurs in T if and only if P is a prefix of some suffix of T.

- Therefore, we want to find some suffix of T that starts with P (i.e. has P has a prefix).

Pattern matching example

```
b a n a n a $
  a n a n a $
    n a n a $
      a n a $
        n a $
          a $
            $
```

Let's say P is `ana` and T is `banana$`. P does appear in T since it is the prefix of a suffix of T—in particular, `ana` is a prefix of `anana$`. In fact, P appears twice in `banana`, and indeed `ana` is a prefix of another suffix of `banana`: `ana$`.

Pattern matching using a suffix array

```
            $
          a $
      a n a $
  a n a n a $
b a n a n a $
        n a $
    n a n a $
```

- With a suffix array, we can find such a suffix using binary search.
- Each iteration of the binary search involves a comparison between P and a suffix of T: this requires $O(m)$ time in the worst case.

Pattern matching using a suffix array

						$
					a	$
			a	n	a	$
	a	n	a	n	a	$
b	a	n	a	n	a	$
				n	a	$
		n	a	n	a	$

- We are binary searching over n suffixes, so we require $O(\log n)$ iterations for the binary search to complete. Overall, our table allows us to determine if P is a substring of T in $O(m \log n)$ time—not bad!

Improving this result.

- Of course, $O(n^2)$ space is not very good.
- Instead of storing the suffixes, store the index of each suffix in sorted order
- Then I can look up the suffix itself by looking at the appropriate index in T.
- For `banana`, we would store the indices: 6, 5, 3, 1, 0, 4, 2.
- This data structure is called a `suffix array`.
- $O(n)$ space; $O(m \log n)$ pattern matching!
- Can extend to achieve $O(m)$ time pattern matching

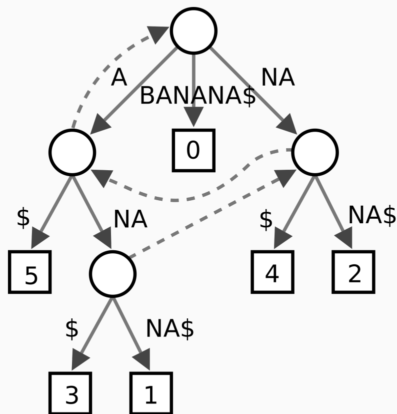
Other Suffix Array Uses

- Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
- List all locations of P in T
- Useful for calculating BWT
- Find the longest common substring between two strings S_1 and S_2 in $O(S_1 + S_2)$ time
- Find the longest palindrome in S in $O(|S|)$ time
- With more work: searching for P in T with errors

Goal for Today

- How fast can we build a suffix array?
- From last time: claimed that we can do $O(n)$ time
- Faster than sorting!
- Only assumption: alphabet is of size $O(n)$.

Beginning: suffix tree



- Similar functionality to suffix array
- We won't talk about (except on this slide)
- Invented in 1976 by McCreight, improved by Ukkonen in 1995
- Construct in $O(n)$ time for constant-sized alphabet using suffix links

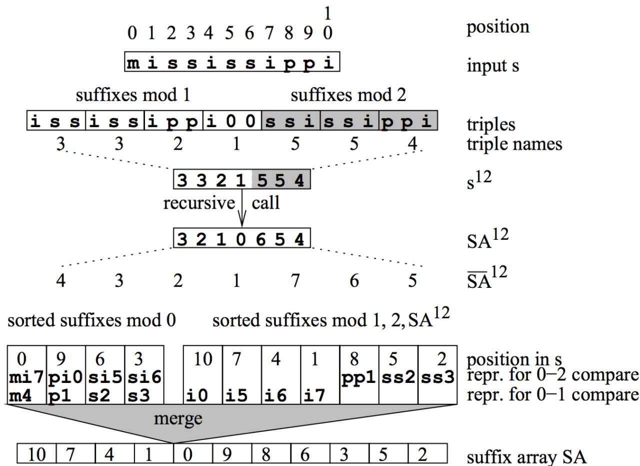
Suffix Array

- Invented by Manber and Myers in 1990
- "3 to 5 times more space-efficient" than a suffix tree
- For constant sized alphabets can be constructed in $O(n)$ expected time
- Can we get to $O(n)$ construction: deterministic and for larger alphabets?
    - Why is this important?

Linear-time Algorithms Through Recursion

- 1997: suffix tree construction algorithm of Farach-Colton
- $O(n)$ time for any alphabet of size at most n
- Instead of suffix links: use recursion!

Linear-time Algorithm Diagram



(There's a reason why we're not going to use this algorithm. But, let's go over the basic idea!)

Recursive Suffix Array Construction: Basic Idea

- Let's sort the odd-numbered suffixes recursively. (The base case is when there's just one odd-numbered suffix, which is trivial to sort.)
- Then: we can assume (recursively) that the odd-numbered suffixes are sorted.
- Use the ordering of the odd-numbered suffixes to build the suffix tree for the even-numbered suffixes

Recursive Suffix Array Construction: Combining

- On the one hand, this seems impossible: the even-numbered suffixes all start with entirely different characters than the odd-numbered suffixes, so the ordering is, right off the bat, 100% different.
- On the other hand, this first character is the *only* difference: once we sort the even-numbered suffixes by their direct character, the odd-numbered suffixes determine the rest of their ordering.
- It's possible in $O(n)$ time with some bookkeeping!
- The SA-IS algorithm has, at its heart, exactly the same structure

Recursive Suffix Array Construction: Running time

- Write the recurrence: $T(n)$ time for a string of length n.
- If $n = 1$, $T(n) = O(1)$.
- Otherwise, $T(n) = T(n/2) + O(n)$
- Solves to $O(n)$!

Improvements since the 90s

- Recursive approach quickly became the most popular
- Long line of research improving these algorithms
- Generally: could simplify the algorithm conceptually at a cost of speed; or could make a fast algorithm with more bookkeeping

The SA-IS algorithm

- Created by Nong, Zhang, and Chan in 2009.
- This algorithm:
  - Uses only arrays to calculate the suffix array—in fact, it only needs the suffix array itself along with a couple of temporary arrays
  - Decreases the input size by 2 with each recursive call in the *worst case*—in practice the size often decreases by 3-4
  - Is quite simple (though precise)—it depends on a few straightforward subroutines and can be written in a couple hundred lines of code.
  - Has (in short) the best practical performance of any algorithm
- There have been improvements since 2009; we'll ignore those.

The SA-IS Algorithm

Plan of attack

- SA-IS is fairly simple in terms of bookkeeping
- But, has a number of different parts
- Each part is short to implement, but may not make sense until they're all put together at the end.

How we'll discuss the algorithm

- First, we'll talk about some basic principles and vocabulary that we'll be using in the algorithm.
- Then, we'll skip the recursive call and go right to obtaining the final solution—we'll talk about how if we obtain the sorted order of special subset of the suffixes (the LMS suffixes), we can finish finding the suffix array in $O(n)$ time. This step will be called an *inductive sort*.
- Finally (on Thursday), we'll talk about the recursive call: how can we sort the LMS suffixes?
- That's a lot of explanation for a (supposedly simple) 100 lines of code! In short: the SA-IS algorithm is fairly simple in terms of methodology, but we'll be going over *how* it works.

Basic Principles: L-type, S-type, and LMS suffixes

- From now on, assume that we want to find the suffix array of a string S of length n, storing the result in a suffix array SA.
- Now: basic building block of the SA-IS algorithm is S-type and L-type suffixes.

Definition 1

A suffix i is S-type if it is *smaller* in lexicographic order than suffix $i+1$; it is L-type if it is *larger* in lexicographic order than suffix $i+1$. The last suffix is always considered S-type.

Labelling types of each suffix

- The first step of our algorithm is going to be labelling every suffix as S-type or L-type.
- How can we do that in linear time?
- Let's look at two different cases.

Labelling types of each suffix

- First, let's say the second character of the ith suffix is different from the first; i.e. $S[i] \neq S[i+1]$. If $S[i] < S[i+1]$ then the ith suffix is S-type; if $S[i] > S[i+1]$ then the ith suffix is L-type.
- If the second character of the ith suffix is equal to the first, then the type is determined by the next character not equal to $S[i]$. Here's one way we can simplify this: if $S[i] = S[i+1]$, then the type of the ith suffix is equal to the type of the $i+1$st suffix.

Labelling types of each suffix: proof idea

b b b b b c

i

b b b b c

$i+1$

Comparing suffix i to suffix $i+1$ (only a part of each suffix is shown). In this case, $S[i] = S[i+1]$. Because the next character not equal to b is c, suffix i is S-type. If there had been an a instead of a c, suffix i would have been L-type.

Linear-time algorithm for assigning types

- We scan S from back to front.
- The last character is always S type.
- Then, for each character $S[i]$, we compare $S[i]$ and $S[i+1]$. If $S[i] < S[i+1]$ we store that i is S-type; if $S[i] > S[i+1]$ we store that i is L-type; if $S[i] = S[i+1]$ we copy the type of suffix $i+1$ over to i.
- Example on board: `CGACTCCAACAACAAGCT$`.

Final definition: LMS suffixes

- A suffix i is an LMS suffix if i is the last character in S, or if suffix i is of type S and suffix $i - 1$ is of type L.
- (Suffix 0 can never be an LMS suffix.)
- These can easily be identified in linear time given the types.
- Example on board

Labelled LMS suffixes

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

Labelling a string with the type of each suffix (L or S). The LMS suffixes are highlighted in red.

Initial Observations of Suffix Tree Structure

Using suffix types

- What can we say, already, about the final suffix tree of our string?
- Right off the bat we may observe that the sorted suffixes must be grouped by their first letter

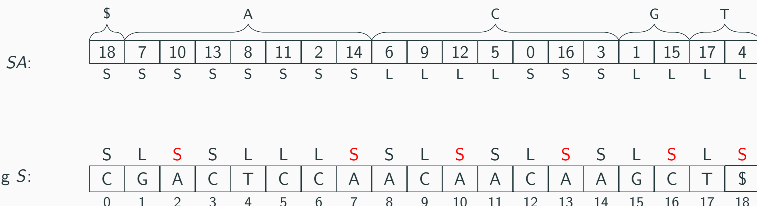
Grouping By First Letter

Let's look at how the *L*-type and *S*-type suffixes wind up being placed in the final suffix array.

	$															G		T	
SA:	18															1	15	17	4

	S	L	S	S	L	L	L	S	S	L	S	S	L	S	S	L	S	L	S
String *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

L-type and S-type suffix grouping



Looking ahead to the final suffix array, the L-type suffixes are always before all S-type suffixes in their bucket.

Formalizing that L-type suffixes are before S-type

Lemma 2

Consider two suffixes i and j of S such that i is L-type, j is S-type, and both i and j start with the same character ($S[i] = S[j]$). Then the ith suffix is before the jth suffix in sorted order.

Why might this be true?

Proof of Lemma 2

Proof.

Let $c = S[i] = S[j]$ be the first character of each string. As discussed when defining L-type strings, suffix i must begin with c repeated 1 or more times, followed by a character $c_i < c$. Similarly, suffix j must begin with c repeated 1 or more times, followed by a character $c_j > c$.

If the number of repeats of c is the same in i and j, then since $c_i < c_j$, i comes before j in sorted order. If i has fewer repeats of c, then i contains c_i at a position where j contains c; since $c_i < c$, i comes earlier in sorted order. Similarly, if i has more repeats of c, then j contains c_j at a position where i contains c; since $c_j > c$, j comes later in sorted order. □

Where we are

With the above, in $O(n)$ time, we can:

- Count the characters to determine where each "bucket" of suffixes that start with the same character begins and ends,
- label each suffix as L-type or S-type and find the LMS suffixes,
- place each suffix in the correct character bucket with the other L-type or S-type suffixes.

Therefore, all we have left to do is sort the suffixes that are of the same type *and* start with the same letter.

Building the Final Suffix Array Using Induced Sorting

Induced Sorting: Plan

- In this section, we skip ahead in terms of the final algorithm.
- **Let's assume that the LMS suffixes are stored in sorted order.**
- How can we use this sorted order to finish sorting the remaining suffixes?
- On Thursday we'll talk about how to sort the LMS suffixes

Formalizing our assumption

- Let P be an array containing the indices of the LMS suffixes so that the suffixes are stored in lexicographicical order.
- The goal of this section is to use P to determine the sorted order of *all* suffixes in $O(n)$ time.
- Let's start small: we have P in sorted order. Can we say *anything* about the order of the other suffixes of S?
- Continuing our example from before: P is 18, 7, 10, 13, 2, 16.

diagram

P:

18	7	10	13	2	16

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 say we're given a sorted array of LMS indices P. What can we say about the remaining suffixes of S?

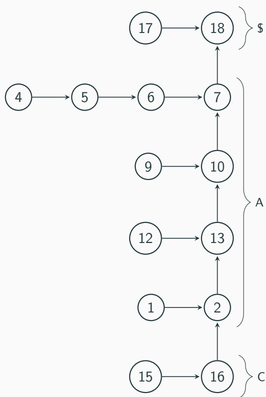
Observations about P

- Each LMS suffix is preceded by an L-type suffix. By definition of L-type suffix, this suffix is before its successive LMS suffix in sorted order.
- In fact we can extend this even further: the L-type suffix before that must be even larger, and so on.
- By definition, any sequence of L-type suffixes is followed by an LMS suffix.
- This means that any L-type suffix can be related to an LMS suffix!

P:

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.



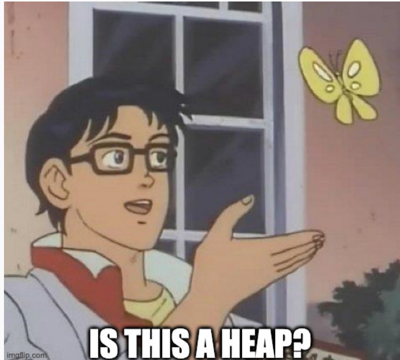
String S:

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

What do we have?

- a big graph of suffixes. But, for the most part,
- don't know very much about the sorted order
- For example, we know nothing about how suffix 12 compares to suffix 5.
- How can we get the sorted order in $O(n)$ time?

Observation



(Out of date meme for today)

- This kind of looks like a min heap
- We know the smallest item, and we know the two next smallest items
- Can we do heap-like operations on this?
- Let's try

Problem with heap

- The number of suffixes keep increasing
- Even comparing *two* suffixes may require $\Theta(n)$ time! We want to use that much time to place *all* of the L-type suffixes
- Can we do something without this comparison?
- Idea: rather than comparing, let's put each suffix we're unsure about in the right place for later.

Idea: use the first character and type

- Look at the first character of the suffix, and the type of the suffix, to determine where it will go
- We will place *all* LMS suffixes and L-type suffixes in sorted order without doing any comparisons between suffixes at all!
- Basic idea will still be: repeatedly removing the smallest suffix

Example sort.

Example

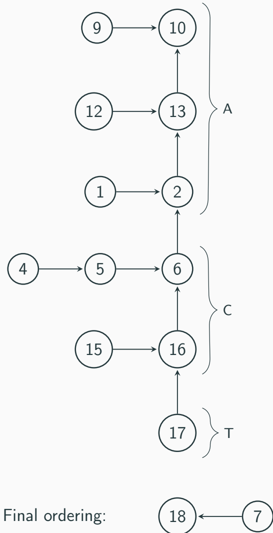
- Let's actually sort the suffixes for the example we've been using
- This example will help motivate our final algorithm.
- Let's start: do we know the smallest suffix?
- Now, what do we do with suffix 17?

First case

- Suffix 17 is easy: it's the only suffix so far starting with T
- We can place it below all the other suffixes in the right column.
- Now what?
- Suffix 7 is next smallest. Where do we put suffix 6?

Second case

- Looking at $S[6]$, we see that suffix 6 starts with C.
- But, how does it compare to suffix 16?
- Since suffix 16 is an LMS suffix, it is S-type.
- Suffix 6, however, is L-type.
- Therefore, we know by Lemma 2 that suffix 6 comes before suffix 16.



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

Third case

- Suffix 10 can, then, be written to the final ordering directly.
- Now we need to place suffix 9. Suffix 9 starts with C and is L-type.
- We know that suffix 9 is before suffix 16 (because 9 is L-type and 16 is S-type). How does suffix 9 compare to suffix 6?

Looking a bit deeper in the third case

- To answer this question, we need to think a little bit about what these suffixes really are.
- Suffix 9 consists of a C followed by suffix 10. Suffix 6 consists of a C followed by suffix 7.
- So the ordering of suffix 9 and suffix 6 is the same as the ordering of suffix 10 and suffix 7.
- But we already know this ordering from the final ordering—suffix 7 comes first!
- Therefore, suffix 9 goes behind suffix 6.

What were all these cases again?

(Note: there aren't actually three cases here...)

- In case 1, we placed the suffix as the only suffix of its character
- In case 2, we placed the string after all S-type suffixes with the same character
- In case 3, we placed the string after all S-type suffixes and before all L-type suffixes with the same character

Simplifying the algorithm

- Overall: seems like we can, in all cases, place each new suffix in the rightmost column after all S-type suffixes and before all L-type suffixes with the same character.
- Uses *no string comparisons whatsoever*
- Each iteration places a new suffix in the correct final ordering
- Any suffix to its left is placed in the rightmost column
- Let's finish sorting all of our L-type and AMS suffixes

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.