Lecture 15: Suffix Arrays 2

Sam McCauley

November 7, 2025

Williams College

Admin

- Suffix Array “checkin” out
    - By Thursday at 10pm: Submit one function (suffix array construction)
    - I would *highly* recommend: before Thursday, go through the C++ code linked on the website, and map each step to what we do today on the board/slides
- Any questions?

From Last Time: Radix Sort

- Radix sort: can sort n k-character strings in time $O(nk)$ (so long as there are at most n characters)

From Last Time: Radix Sort

- Radix sort: can sort n k-character strings in time $O(nk)$ (so long as there are at most n characters)

- Idea: sort one character at a time using $O(n)$ time counting sort

From Last Time: Radix Sort

- Radix sort: can sort n k-character strings in time $O(nk)$ (so long as there are at most n characters)
- Idea: sort one character at a time using $O(n)$ time counting sort
- Specifically, proceed *backwards*: sort by last character, then (stable) sort by second to last character, and so on

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co
- Now, assume we have sorted the prefixes by their first 2^k charac k. Goal: sort them by the first 2^{k+1} characters

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co
- Now, assume we have sorted the prefixes by their first 2^k charac
  k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a "class" based on where it is in sorted order

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using c
- Now, assume we have sorted the prefixes by their first 2^k charac k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a “class” based on where it is in sorted order
  - First 2^{k+1} characters is the same as the first 2^k characters, then the second 2^k characters

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co
- Now, assume we have sorted the prefixes by their first 2^k charac
  k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a "class" based on where it is in sorted order
  - First 2^{k+1} characters is the same as the first 2^k characters, then the second 2^k characters
  - Radix sort by: (class of first 2^k characters, class of second 2^k characters)

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co
- Now, assume we have sorted the prefixes by their first 2^k charac k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a “class” based on where it is in sorted order
  - First 2^{k+1} characters is the same as the first 2^k characters, then the second 2^k characters
  - Radix sort by: (class of first 2^k characters, class of second 2^k characters)
  - Same as sorting by first 2^{k+1} characters! Then we increment k and continue, until we are sorted

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using cc
- Now, assume we have sorted the prefixes by their first 2^k charac
  k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a "class" based on where it is in sorted order
  - First 2^{k+1} characters is the same as the first 2^k characters, then the second 2^k characters
  - Radix sort by: (class of first 2^k characters, class of second 2^k characters)
  - Same as sorting by first 2^{k+1} characters! Then we increment k and continue, until we are sorted
- Let's do this for `CACATACACAGACACAC$`

Prefix Doubling: Main Idea

- We can sort strings by their *first* character in $O(n)$ time using co
- Now, assume we have sorted the prefixes by their first 2^k charac k. Goal: sort them by the first 2^{k+1} characters
  - For every prefix, we know the ordering of the first 2^k characters. Assign each prefix a “class” based on where it is in sorted order
  - First 2^{k+1} characters is the same as the first 2^k characters, then the second 2^k characters
  - Radix sort by: (class of first 2^k characters, class of second 2^k characters)
  - Same as sorting by first 2^{k+1} characters! Then we increment k and continue, until we are sorted
- Let’s do this for `CACATACACAGACACAC$`
- Correct answer:
  - `17 15 13 11 5 7 1 9 3 16 14 12 6 0 8 2 10 4`

Suffix Array Construction: Detailed Description

What we Store

- Two arrays: $p[]$ stores order of the suffixes sorted so far; c stores the class of the 2^k-length prefix of the suffix

What we Store

- Two arrays: $p[]$ stores order of the suffixes sorted so far; c stores the class of the 2^k-length prefix of the suffix
- Goal: after we are done, $p[]$ will store our suffix array (the suffixes will be entirely sorted)

What we Store

- Two arrays: $p[]$ stores order of the suffixes sorted so far; c stores the class of the 2^k-length prefix of the suffix

- Goal: after we are done, $p[]$ will store our suffix array (the suffixes will be entirely sorted)

- Will also use two temporary arrays $pn[]$, $cn[]$ as we move items around, and a temporary array $cnt[]$ to count items in the counting sort.

Step 1

First, sort all suffixes by their first character using counting sort.

- Count the number of occurrences of each first character using *cnt*

Step 1

First, sort all suffixes by their first character using counting sort.

- Count the number of occurrences of each first character using *cnt*
- Go through each suffix one at a time; look at its first character and place it into *p* using *cnt*

Reminder of prefix doubling idea

- Remember: we have sorted all suffixes by their first 2^k characters. Our goal is to sort them by their first 2^{k+1} characters

Reminder of prefix doubling idea

- Remember: we have sorted all suffixes by their first 2^k characters. Our goal is to sort them by their first 2^{k+1} characters

- The first 2^{k+1} characters can be viewed as a pair: (1^{st} 2^k chars, 2^{nd} 2^k characters)

Reminder of prefix doubling idea

- Remember: we have sorted all suffixes by their first 2^k characters. Our goal is to sort them by their first 2^{k+1} characters
- The first 2^{k+1} characters can be viewed as a pair: (1^{st} 2^k chars, 2^{nd} 2^k characters)
- We can radix sort this pair in $O(n)$ time: first counting sort all suffixes by the second 2^k characters, then counting sort by the first 2^k characters

Reminder of prefix doubling idea

- Remember: we have sorted all suffixes by their first 2^k characters. Our goal is to sort them by their first 2^{k+1} characters
- The first 2^{k+1} characters can be viewed as a pair: (1^{st} 2^k chars, 2^{nd} 2^k characters)
- We can radix sort this pair in $O(n)$ time: first counting sort all suffixes by the second 2^k characters, then counting sort by the first 2^k characters
- Then we're sorted by 2^{k+1}! If $2^{k+1} < n$, then increment k and go again; otherwise we're done

Doubling Step: First, sort by second 2^k characters

Our current status: $p[]$ contains all suffixes sorted by their first 2^k characters. We want to sort them by the *second* 2^k characters

- Let's say the ith prefix of the string is in position j

Doubling Step: First, sort by second 2^k characters

Our current status: $p[]$ contains all suffixes sorted by their first 2^k characters. We want to sort them by the *second* 2^k characters

- Let's say the ith prefix of the string is in position j
- Then the $i - 2^k$th prefix of the string should be in position j after we sort!

Doubling Step: First, sort by second 2^k characters

Our current status: $p[]$ contains all suffixes sorted by their first 2^k characters. We want to sort them by the *second* 2^k characters

- Let's say the ith prefix of the string is in position j
- Then the $i - 2^k$th prefix of the string should be in position j after we sort!
- This step can be accomplished as follows: for each entry of p, subtract 2^k; store the result in $pn[]$

Doubling Step: First, sort by second 2^k characters

Our current status: $p[]$ contains all suffixes sorted by their first 2^k characters. We want to sort them by the *second* 2^k characters

- Let's say the ith prefix of the string is in position j
- Then the $i - 2^k$th prefix of the string should be in position j after we sort!
- This step can be accomplished as follows: for each entry of p, subtract 2^k; store the result in $pn[]$
- Can't be less than 0; wrap around by adding n if so

Doubling Step: First, sort by second 2^k characters

Our current status: $p[]$ contains all suffixes sorted by their first 2^k characters. We want to sort them by the *second* 2^k characters

- Let's say the ith prefix of the string is in position j
- Then the $i - 2^k$th prefix of the string should be in position j after we sort!
- This step can be accomplished as follows: for each entry of p, subtract 2^k; store the result in $pn[]$
- Can't be less than 0; wrap around by adding n if so
- (I love this step)

Doubling Step: Then, Sort by *first* 2^k characters

Our current status: $pn[]$ contains all suffixes sorted by their *second* 2^k characters. We want to sort them by the *first* 2^k characters. We have that $c[]$ contains, for each suffix, “which class” it is in

- Use counting sort to place each suffix of $pn[]$ in the correct place by its first 2^k characters:

Doubling Step: Then, Sort by *first* 2^k characters

Our current status: $pn[]$ contains all suffixes sorted by their *second* 2^k characters. We want to sort them by the *first* 2^k characters. We have that $c[]$ contains, for each suffix, “which class” it is in

- Use counting sort to place each suffix of $pn[]$ in the correct place by its first 2^k characters:
  - Use $cnt[]$ to count the number of suffixes strictly before each class

Doubling Step: Then, Sort by *first* 2^k characters

Our current status: *pn*[] contains all suffixes sorted by their *second* 2^k characters. We want to sort them by the *first* 2^k characters. We have that *c*[] contains, for each suffix, "which class" it is in

- Use counting sort to place each suffix of *pn*[] in the correct place by its first 2^k characters:
  - Use *cnt*[] to count the number of suffixes strictly before each class
  - Go through each suffix in *pn*[]; *c*[] gives its class. Look up the class in *cnt*[], decrement it, and place it in *p*[].

Doubling Step: finally, update $c[]$

Our current status: $p[]$ contains all suffixes sorted by their first 2^{k+1} characters. We want to update $c[]$ so we can use it in the next iteration. We'll place the new values into $cn[]$; then swap $c[]$ and $cn[]$. To begin, $cn[p[0]] = 0$: the first prefix in alphabetical order has class 0.

- Remember that we sorted by: $c[i], c[i+2^k]$

Doubling Step: finally, update $c[]$

Our current status: $p[]$ contains all suffixes sorted by their first 2^{k+1} characters. We want to update $c[]$ so we can use it in the next iteration. We'll place the new values into $cn[]$; then swap $c[]$ and $cn[]$. To begin, $cn[p[0]] = 0$: the first prefix in alphabetical order has class 0.

- Remember that we sorted by: $c[i], c[i + 2^k]$
- Go through each suffix j in order of p; let j' be the next suffix in p (so $j = p[i]$ and $j' = p[i + 1]$). If $c[j] = c[j']$ and $c[j' + 2^k] = c[j' + 2^k]$, then we keep $cn[j'] = cn[j]$; otherwise, $cn[j] = cn[j'] + 1$.

Putting it all Together

First, counting sort by first letter.

For $k = 1$ to $\lceil \log_2 n \rceil$:

1. Place suffixes from $p[]$ into $pn[]$, sorted by *second* set of 2^k characters. [Performed using subtraction]

Putting it all Together

First, counting sort by first letter.

For $k = 1$ to $\lceil \log_2 n \rceil$:

1. Place suffixes from $p[]$ into $pn[]$, sorted by *second* set of 2^k characters.
   [Performed using subtraction]

2. Place suffixes from $pn[]$ into $p[]$, sorted by *first* set of 2^k characters.
   [Performed using counting sort]

Putting it all Together

First, counting sort by first letter.

For $k = 1$ to $\lceil \log_2 n \rceil$:

1. Place suffixes from $p[]$ into $pn[]$, sorted by *second* set of 2^k characters. [Performed using subtraction]

2. Place suffixes from $pn[]$ into $p[]$, sorted by *first* set of 2^k characters. [Performed using counting sort]

3. Store classes of length 2^{k+1} in $cn[]$ by iterating through $p[]$ and comparing successive classes. This is the new $c[]$ for the next loop

Each inner step requires $O(1)$ array scans; $O(n \log n)$ time!

Suffix Array Conclusion

Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the “prefix-doubling approach”

Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the "prefix-doubling approach"
- Is this good/bad? Can we do better? What is known?

Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the "prefix-doubling approach"
- Is this good/bad? Can we do better? What is known?
- Can find the suffix array in $O(n)$ time

Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the "prefix-doubling approach"
- Is this good/bad? Can we do better? What is known?
- Can find the suffix array in $O(n)$ time
- Two ways to do this:

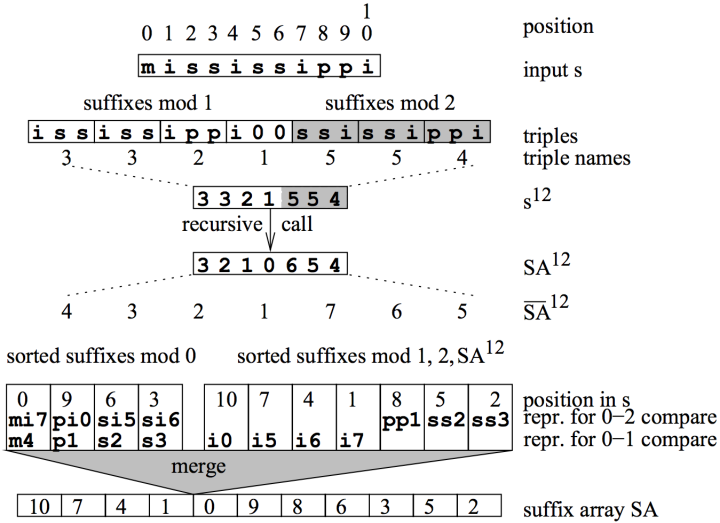
Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the “prefix-doubling approach”
- Is this good/bad? Can we do better? What is known?
- Can find the suffix array in $O(n)$ time
- Two ways to do this:
  - Using suffix links, and

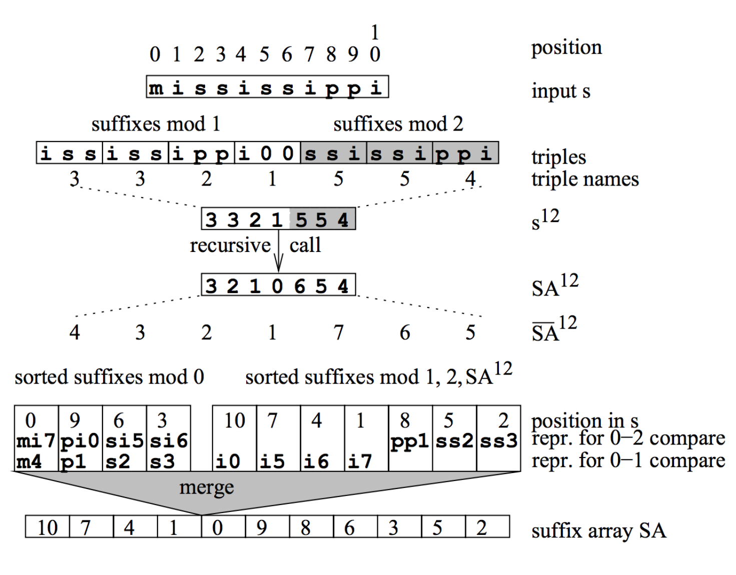
Suffix Array Construction

- This sorts the suffixes in $O(n \log n)$ time; called the “prefix-doubling approach”
- Is this good/bad? Can we do better? What is known?
- Can find the suffix array in $O(n)$ time
- Two ways to do this:
  - Using suffix links, and
  - using recursion

Linear-time Algorithm Diagram



Linear-time Algorithm Diagram



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

Recursive Suffix Array Construction: Basic Idea [Farach-Colton 97]

- 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.)

Recursive Suffix Array Construction: Basic Idea [Farach-Colton 97]

- 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.

Recursive Suffix Array Construction: Basic Idea [Farach-Colton 97]

- 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.

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 first character, the odd-numbered suffixes determine the rest of their ordering.

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 first character, the odd-numbered suffixes determine the rest of their ordering.

- It’s possible in $O(n)$ time with some bookkeeping!

SA-IS Algorithm

- An extremely practical $O(n)$ time suffix array construction algorithm (far faster than version we did)

SA-IS Algorithm

- An extremely practical $O(n)$ time suffix array construction algorithm (far faster than version we did)
    - Your algorithm will run in about `40`s on `timeData.txt`; SA-IS runs in about 2s.

SA-IS Algorithm

- An extremely practical $O(n)$ time suffix array construction algorithm (far faster than version we did)
  - Your algorithm will run in about 40s on `timeData.txt`; SA-IS runs in about 2s.
- I posted a writeup on the website; this is also how your code will be tested for correctness.

SA-IS Algorithm

- An extremely practical $O(n)$ time suffix array construction algorithm (far faster than version we did)
  - Your algorithm will run in about 40s on `timeData.txt`; SA-IS runs in about 2s.
- I posted a writeup on the website; this is also how your code will be tested for correctness.
- Incredibly short, clean, and unintuitive

Suffix Array Uses

Pattern matching

- One of the most ubiquitous string problems involves searching for occurrences of one string in another.

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.

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?

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?
- First: compute the suffix array on T

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?
- First: compute the suffix array on T
- Then: binary search for P in the suffix array

Pattern Matching Analysis

- Binary search requires $O(\log n)$ comparisons

Pattern Matching Analysis

- Binary search requires $O(\log n)$ comparisons
- Each comparison takes $O(m)$ time (may need to walk through whole pattern)

Pattern Matching Analysis

- Binary search requires $O(\log n)$ comparisons
- Each comparison takes $O(m)$ time (may need to walk through whole pattern)
- Gives $O(m \log n)$ time overall

Pattern Matching Analysis

- Binary search requires $O(\log n)$ comparisons
- Each comparison takes $O(m)$ time (may need to walk through whole pattern)
- Gives $O(m \log n)$ time overall
- Can be improved to $O(m)$ time—can search for all occurrences of a pattern in the time it takes to read the pattern!

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
  - Also improvable to $O(m)$

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
  - Also improvable to $O(m)$
- List all locations of P in T

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
  - Also improvable to $O(m)$
- List all locations of P in T
- Useful for calculating BWT

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
  - Also improvable to $O(m)$
- 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

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
    - Also improvable to $O(m)$
- 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

Other Suffix Array Uses

- Already: Can count the # of occurrences of P in $O(m \log n)$ total time (how?)
    - Also improvable to $O(m)$
- 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