Outline

1. Motivation
2. Problem Statement
3. Definitions/Setup
4. Analysis
5. Reflection
6. Discussion
Motivation

➔ Suffix Trees and Arrays are relatively well-studied data structures with many applications
Motivation

➔ Suffix Trees and Arrays are relatively well-studied data structures with many applications

- Interchangeable
  - Can be converted between each other relatively quickly
- Handle somewhat different problem scenarios
Motivation

➔ Examples of problems suffix arrays/trees solve
Motivation

- Examples of problems suffix arrays/trees solve
  - Pattern searching
Motivation

➔ Examples of problems suffix arrays/trees solve

- Pattern searching
- Longest repeated substring
Motivation

Examples of problems suffix arrays/trees solve
- Pattern searching
- Longest repeated substring
- Longest common substring (between two strings)
Motivation

➔ Examples of problems suffix arrays/trees solve
  ● Pattern searching
  ● Longest repeated substring
  ● Longest common substring (between two strings)
  ● Longest palindrome in a string
Motivation

➔ Examples of problems suffix arrays/trees solve
  ● Pattern searching
  ● Longest repeated substring
  ● Longest common substring (between two strings)
  ● Longest palindrome in a string
  ● etc.!
Motivation

Applications to real life
Motivation

➔ Applications to real life
  ● Bioinformatics
    ● DNA/RNA sequencing
Motivation

→ Applications to real life
  ● Bioinformatics
    ● DNA/RNA sequencing
  ● Data compression
Motivation

→ Applications to real life
  ● Bioinformatics
    ● DNA/RNA sequencing
  ● Data compression
  ● Engineering interviews
Problem Statement

➔ Given an input string of length n:
Problem Statement

➔ Given an input string of length n:

aladdin (n = 7)
Problem Statement

→ Given an input string of length n:
  aladdin (n = 7)

→ Return a permutation of (0...n)
  ● This permutation designates the sorted order of the string's suffixes
Problem Statement

Given an input string of length n:

aladdin (n = 7)

Return a permutation of (0...n)

- This permutation designates the sorted order of the string's suffixes
- One index (n) corresponds to the empty suffix
  - Treat the string as if it's infinitely extended by "0"s which are lexicographically earliest
A Quick Example

→ Consider "aladdin" as before
A Quick Example

→ Consider "aladdin" as before
→ The list of suffixes is:
  ● "" - 7
  ● "n" - 6
  ● "in" - 5
  ● "din" - 4
  ● "ddin" - 3
  ● "ddin" - 3
  ● "addin" - 2
  ● "laddin" - 1
  ● "aladdin" - 0
A Quick Example

➔ Consider "aladdin" as before
➔ The sorted list of suffixes is:
  ● "" - 7
  ● "addin" - 2
  ● "aladdin" - 0
  ● "ddin" - 3
  ● "din" - 4
  ● "in" - 5
  ● "laddin" - 1
  ● "n" - 6
A Quick Example

➔ Consider "aladdin" as before
➔ The sorted list of suffixes is:
  ● "" - 7
  ● "addin" - 2
  ● "aladdin" - 0
  ● "ddin" - 3
  ● "din" - 4
  ● "in" - 5
  ● "laddin" - 1
  ● "n" - 6
➔ Hence, the suffix array is (7, 2, 0, 3, 4, 5, 1, 6)
Definitions/Setup

> Goal: linear time suffix array construction algorithm
Goal: linear time suffix array construction algorithm

- Allows for lack of bottleneck with regards to linear time algorithmic solutions for string matching, etc.
- Should also be space efficient
Goal: linear time suffix array construction algorithm
- Allows for lack of bottleneck with regards to linear time algorithmic solutions for string matching, etc.
- Should also be space efficient

A few choices for the alphabet
Goal: linear time suffix array construction algorithm
- Allows for lack of bottleneck with regards to linear time algorithmic solutions for string matching, etc.
- Should also be space efficient

A few choices for the alphabet
- Need not be limited to only 26 or 52 letters from English alphabet
  - Example of a constant alphabet
Goal: linear time suffix array construction algorithm
- Allows for lack of bottleneck with regards to linear time algorithmic solutions for string matching, etc.
- Should also be space efficient

A few choices for the alphabet
- Need not be limited to only 26 or 52 letters from English alphabet
  - Example of a constant alphabet
- **Integer alphabet**: characters are integers from a linear-sized range
Goal: linear time suffix array construction algorithm
- Allows for lack of bottleneck with regards to linear time algorithmic solutions for string matching, etc.
- Should also be space efficient

A few choices for the alphabet
- Need not be limited to only 26 or 52 letters from English alphabet
  - Example of a constant alphabet
- Integer alphabet: characters are integers from a linear-sized range
  - Prior algorithm already exists, but is complicated and somewhat suboptimal
Definitions/Setup

Restrict the alphabet to $[1, n]$

- Not as limiting as it seems: can run coordinate compression over the letters to reduce an arbitrarily complex string into a linear alphabet representation
  - Ranking each letter relatively
Definitions/Setup

Let the input be a string $T$ of size $n$
Definitions/Setup

➔ Let the input be a string $T$ of size $n$
➔ Denote $[i, j]$ and $(i, j)$ as ranges of integers (including and excluding
  $j$, respectively)
Let the input be a string $T$ of size $n$

Denote $[i, j]$ and $(i, j)$ as ranges of integers (including and excluding $j$, respectively)

- Extend to substrings as: $T[0, n) = t_0t_1...t_{n-1}$
- Assume $t_j = 0$ for $j \geq n$
Definitions/Setup

→ Let the input be a string $T$ of size $n$
→ Denote $[i, j]$ and $[i, j)$ as ranges of integers (including and excluding $j$, respectively)
  ● Extend to substrings as: $T[0, n) = t_0t_1...t_{n-1}$
  ● Assume $t_j = 0$ for $j \geq n$
  ● Denote $S_i$ as the suffix $T[i, n)$
Let the input be a string $T$ of size $n$

Denote $[i, j]$ and $[i, j)$ as ranges of integers (including and excluding $j$, respectively)

- Extend to substrings as: $T[0, n) = t_0t_1...t_{n-1}$
- Assume $t_j = 0$ for $j \geq n$
- Denote $S_i$ as the suffix $T[i, n)$
- Also extend to sets: for a set $C$, $S_C$ is set of all $S_i$ for $i$ in $C$
Definitions/Setup

Let the input be a string $T$ of size $n$

Denote $[i, j]$ and $[i, j)$ as ranges of integers (including and excluding $j$, respectively)

- Extend to substrings as: $T[0, n) = t_0t_1...t_{n-1}$
- Assume $t_j = 0$ for $j \geq n$
- Denote $S_i$ as the suffix $T[i, n)$
- Also extend to sets: for a set $C$, $S_C$ is set of all $S_i$ for $i$ in $C$
- Want to find the suffix array $SA[0, n]$ of $T$
Analysis (Motivation)

Prior algorithm by Farach has a half-recursive divide-and-conquer approach
Analysis (Motivation)

➔ Prior algorithm by Farach has a half-recursive divide-and-conquer approach
  ● 1. Construct suffix tree of suffixes starting at odd positions via reduction
Analysis (Motivation)

➔ Prior algorithm by Farach has a half-recursive divide-and-conquer approach
  ● 1. Construct suffix tree of suffixes starting at odd positions via reduction
  ● 2. Construct suffix tree of remaining suffixes using result of first step
Analysis (Motivation)

Prior algorithm by Farach has a half-recursive divide-and-conquer approach

- 1. Construct suffix tree of suffixes starting at odd positions via reduction
- 2. Construct suffix tree of remaining suffixes using result of first step
- 3. Merge two suffix trees into one (pretty costly, intricate, and complex)
Prior algorithm by Farach has a half-recursive divide-and-conquer approach

1. Construct suffix tree of suffixes starting at odd positions via reduction
2. Construct suffix tree of remaining suffixes using result of first step
3. Merge two suffix trees into one (pretty costly, intricate, and complex)
   May compare two suffixes in constant time using what you already know
Analysis (Motivation)

➔ Prior algorithm by Farach has a half-recursive divide-and-conquer approach

- 1. Construct suffix tree of suffixes starting at odd positions via reduction
- 2. Construct suffix tree of remaining suffixes using result of first step
- 3. Merge two suffix trees into one (pretty costly, intricate, and complex)
- May compare two suffixes in constant time using what you already know
  - For instance, if you know $S_3 > S_5$, then comparing $S_2$ and $S_4$ is very quick
  - When would $S_2$ and $S_4$ take a long time to compare?
Prior algorithm by Farach has a half-recursive divide-and-conquer approach

1. Construct suffix tree of suffixes starting at odd positions via reduction
2. Construct suffix tree of remaining suffixes using result of first step
3. Merge two suffix trees into one (pretty costly, intricate, and complex)
May compare two suffixes in constant time using what you already know
   - For instance, if you know $S_3 > S_5$, then comparing $S_2$ and $S_4$ is very quick
   - When would $S_2$ and $S_4$ take a long time to compare?
     - If many characters are the same between them
Analysis (Motivation)

Prior algorithm by Farach has a half-recursive divide-and-conquer approach

1. Construct suffix tree of suffixes starting at odd positions via reduction
2. Construct suffix tree of remaining suffixes using result of first step
3. Merge two suffix trees into one (pretty costly, intricate, and complex)

May compare two suffixes in constant time using what you already know

- For instance, if you know $S_3 > S_5$, then comparing $S_2$ and $S_4$ is very quick
- When would $S_2$ and $S_4$ take a long time to compare?
  - If many characters are the same between them
  - After comparing $t_2$ and $t_4$ and seeing they're equal, we can simply use what we know about the remaining characters in $S_3$ and $S_5$ to deduce that $S_2 > S_4$
Consider using $\frac{2}{3}$-recursion instead of half-recursion.
Analysis (Motivation)

→ Consider using $\frac{2}{3}$-recursion instead of half-recursion
  ● 1. Construct suffix array of suffixes at indices i not divisible by 3
Analysis (Motivation)

➔ Consider using \( \frac{2}{3} \)-recursion instead of half-recursion
  
  1. Construct suffix array of suffixes at indices i not divisible by 3
  2. Construct suffix array of remaining suffixes using result of first step
Consider using $\frac{2}{3}$-recursion instead of half-recursion

1. Construct suffix array of suffixes at indices i not divisible by 3
2. Construct suffix array of remaining suffixes using result of first step
3. Merge two suffix arrays into one
Analysis (Motivation)

➔ Consider using $\frac{2}{3}$-recursion instead of half-recursion
  
  - 1. Construct suffix array of suffixes at indices i not divisible by 3
  - 2. Construct suffix array of remaining suffixes using result of first step
  - 3. Merge two suffix arrays into one

➔ This actually makes the last step almost trivial
Analysis (Motivation)

➔ Consider using \( \frac{2}{3} \)-recursion instead of half-recursion
  ● 1. Construct suffix array of suffixes at indices i not divisible by 3
  ● 2. Construct suffix array of remaining suffixes using result of first step
  ● 3. Merge two suffix arrays into one

➔ This actually makes the last step almost trivial
  ● Comparison-based merging is always sufficient in this case
    ● Given \( S_i \) and \( S_j \), just need to compare \( t_i \) and \( t_j \), then compare later suffixes whose relative order we already know
Analysis (DC3)

→ Simple linear-time algorithm (DC3) along with example
  ● Again, take $T = \text{aladdin}$, $n = 7$
Analysis (DC3)

→ Simple linear-time algorithm (DC3) along with example
  ● Again, take $T = \text{aladdin}$, $n = 7$
→ For $k = 0, 1, 2$, define $B_k = \{i \in [0, n] \mid i \mod 3 = k\}$
Analysis (DC3)

- Simple linear-time algorithm (DC3) along with example
  - Again, take $T = \text{aladdin}$, $n = 7$
  - For $k = 0, 1, 2$, define $B_k = \{i \in [0, n] \mid i \mod 3 = k\}$
    - Let $C = B_1 \cup B_2$ be the set of sample positions and $S_C$ be the set of sample suffixes
Simple linear-time algorithm (DC3) along with example

Again, take $T = \text{aladdin}$, $n = 7$

For $k = 0, 1, 2$, define $B_k = \{i \text{ in } [0, n] \mid i \text{ mod } 3 = k\}$

Let $C = B_1 \cup B_2$ be the set of sample positions and $S_C$ be the set of sample suffixes

- $B_1 = \{1, 4, 7\}$, $B_2 = \{2, 5\}$, $B_0 = \{0, 3, 6\}$, $C = \{1, 4, 7, 2, 5\}$, $S_C = \{\text{laddin, din, ...}\}$
Analysis (DC3)

➔ Step 1: Sort Sample Suffixes
Analysis (DC3)

➔ Step 1: Sort Sample Suffixes
  • We can take advantage of our modulo 3 construction by constructing character triplets
Analysis (DC3)

➔ Step 1: Sort Sample Suffixes
  ● We can take advantage of our modulo 3 construction by constructing character triplets
  ● For $k = 1, 2$, construct $R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] ...$
  ● Let $R$ be concatenation of $R_1$ and $R_2$
Analysis (DC3)

→ Step 1: Sort Sample Suffixes

- We can take advantage of our modulo 3 construction by constructing character triplets
- For \( k = 1, 2 \), construct \( R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots \)
- Let \( R \) be concatenation of \( R_1 \) and \( R_2 \)
  - \( R = [l][a][d][d][i][n][0][0][0][a][d][d][i][n][0] \)
Analysis (DC3)

→ Step 1: Sort Sample Suffixes

- We can take advantage of our modulo 3 construction by constructing character triplets
- For \( k = 1, 2 \), construct \( R_k = [t_{k}\ t_{k+1}\ t_{k+2}][t_{k+3}\ t_{k+4}\ t_{k+5}]... \)
- Let \( R \) be concatenation of \( R_1 \) and \( R_2 \)
  - \( R = [lad][din][000][add][in0] \)

  ○ By sorting the suffixes of this, we get the order of the sample suffixes \( S_C \)
Analysis (DC3)

→ Step 1: Sort Sample Suffixes

- We can take advantage of our modulo 3 construction by constructing character triplets.
- For \( k = 1, 2 \), construct \( R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots \)
- Let \( R \) be concatenation of \( R_1 \) and \( R_2 \)
  - \( R = [lad][din][000][add][in0] \)
    - By sorting the suffixes of this, we get the order of the sample suffixes \( S_C \)
- Radix sort "characters" of \( R \) and coordinate compress to get \( R' \)
Analysis (DC3)

→ Step 1: Sort Sample Suffixes

- We can take advantage of our modulo 3 construction by constructing character triplets
- For $k = 1, 2$, construct $R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] ...$
- Let $R$ be concatenation of $R_1$ and $R_2$
  - $R = [lad][din][000][add][in0]$
    - By sorting the suffixes of this, we get the order of the sample suffixes $S_c$
- Radix sort "characters" of $R$ and coordinate compress to get $R'$
  - If all numbers here are different, then we have the order of suffixes
  - Otherwise, recursively sort suffixes of $R'$ with DC3
Analysis (DC3)

→ Step 1: Sort Sample Suffixes

- We can take advantage of our modulo 3 construction by constructing character triplets
- For $k = 1, 2$, construct $R_k = \left[ t_k t_{k+1} t_{k+2} \right] \left[ t_{k+3} t_{k+4} t_{k+5} \right] ...$
- Let $R$ be concatenation of $R_1$ and $R_2$
  - $R = \left[ lad \right] \left[ din \right] \left[ 000 \right] \left[ add \right] \left[ in0 \right]$
    - By sorting the suffixes of this, we get the order of the sample suffixes $S_C$
- Radix sort "characters" of $R$ and coordinate compress to get $R'$
  - If all numbers here are different, then we have the order of suffixes
  - Otherwise, recursively sort suffixes of $R'$ with DC3
- In this case, $R' = (5, 3, 1, 2, 4)$
Analysis (DC3)

→ Step 1: Sort Sample Suffixes
  ● We have $R' = (5, 3, 1, 2, 4)$
Analysis (DC3)

→ Step 1: Sort Sample Suffixes
  ● We have $R' = (5, 3, 1, 2, 4)$
  ● Now, assign ranks to each suffix that we know of
  ● Let • denote value we do not know
Step 1: Sort Sample Suffixes

- We have \( R' = (5, 3, 1, 2, 4) \)
- Now, assign ranks to each suffix that we know of
- Let \( \bullet \) denote value we do not know
- \( \text{rank}(S_i) = \bullet 5 2 \bullet 3 4 \bullet 1 \)
Analysis (DC3)

→ Step 1: Sort Sample Suffixes
  ● We have R' = (5, 3, 1, 2, 4)
  ● Now, assign ranks to each suffix that we know of
  ● Let • denote value we do not know
  ● rank(S_i) = • 5 2 • 3 4 • 1
    ● Remember that R' is a concatenation of R_1 and R_2, not an interleaving (so it's somewhat out of order)
Analysis (DC3)

→ Step 2: Sort Nonsample Suffixes
Analysis (DC3)

➔ Step 2: Sort Nonsample Suffixes
  ● Can represent each nonsample suffix $S_i$ as the pair $(t_i, \text{rank}(S_{i+1}))$
    ● Takes advantage of the reuse of information discussed earlier
Step 2: Sort Nonsample Suffixes

- Can represent each nonsample suffix $S_i$ as the pair $(t_i, \text{rank}(S_{i+1}))$
  - Takes advantage of the reuse of information discussed earlier
- Each suffix can then be radix sorted with at most two comparisons
Analysis (DC3)

→ **Step 2: Sort Nonsample Suffixes**
  - Can represent each nonsample suffix $S_i$ as the pair $(t_i, \text{rank}(S_{i+1}))$
    - Takes advantage of the reuse of information discussed earlier
  - Each suffix can then be radix sorted with at most two comparisons

→ In this case, our nonsample suffixes are "aladdin", "ddin", and "n"
Analysis (DC3)

➔ **Step 2: Sort Nonsample Suffixes**
  - Can represent each nonsample suffix $S_i$ as the pair $(t_i, \text{rank}(S_{i+1}))$
    - Takes advantage of the reuse of information discussed earlier
  - Each suffix can then be radix sorted with at most two comparisons

➔ **In this case, our nonsample suffixes are "aladdin", "ddin", and "n"**
  - We know $\text{rank}(S_i) = 5 \ 2 \ 3 \ 4 \ 1$
    - Thus, our pairs to sort are (a, 5), (d, 3), and (n, 1)
Analysis (DC3)

➔ Step 2: Sort Nonsample Suffixes
  ● Can represent each nonsample suffix \( S_i \) as the pair \( (t_i, \text{rank}(S_{i+1})) \)
    ● Takes advantage of the reuse of information discussed earlier
  ● Each suffix can then be radix sorted with at most two comparisons

➔ In this case, our nonsample suffixes are "aladdin", "ddin", and "n"
  ● We know rank\( (S_i) = 5 \ 2 \ 3 \ 4 \ 1 \)
    ● Thus, our pairs to sort are (a, 5), (d, 3), and (n, 1)
    ● (a, 5) < (d, 3) < (n, 1), so \( S_0 < S_3 < S_6 \)
Analysis (DC3)

➤ Step 3: Merge
Analysis (DC3)

→ Step 3: Merge
  ● Two sorted sets are merged using standard comparison merging (e.g. in mergesort)
Analysis (DC3)

Step 3: Merge

- Two sorted sets are merged using standard comparison merging (e.g. in mergesort)
- To compare $S_i$ and $S_j$, there are two simple cases
  - $i$ is 1 mod 3: use the same pairing $(t_i, \text{rank}(S_{i+1}))$ formulation to compare
  - $i$ is 2 mod 3: use a triplet $(t_i, t_{i+1}, \text{rank}(S_{i+2}))$ formulation to compare
Analysis (DC3)

⇒ Step 3: Merge

- Two sorted sets are merged using standard comparison merging (e.g. in mergesort)
- To compare $S_i$ and $S_j$, there are two simple cases
  - $i$ is 1 mod 3: use the same pairing $(t_i, \text{rank}(S_{i+1}))$ formulation to compare
  - $i$ is 2 mod 3: use a triplet $(t_i, t_{i+1}, \text{rank}(S_{i+2}))$ formulation to compare
- In either case, comparison can be done in $O(1)$, since the ranks will be well-defined in all cases
Analysis (DC3)

⇒ Step 3: Merge

- Two sorted sets are merged using standard comparison merging (e.g. in mergesort)
- To compare $S_i$ and $S_j$, there are two simple cases
  - $i$ is 1 mod 3: use the same pairing $(t_i, \text{rank}(S_{i+1}))$ formulation to compare
  - $i$ is 2 mod 3: use a triplet $(t_i, t_{i+1}, \text{rank}(S_{i+2}))$ formulation to compare
- In either case, comparison can be done in $O(1)$, since the ranks will be well-defined in all cases
- In our example, a simple merge results in: (7, 2, 0, 3, 4, 5, 1, 6)
  - As we saw earlier, this is the suffix array!
Analysis (DC3)

→ We can apply the Master Theorem to analyze the complexity of DC3
Analysis (DC3)

We can apply the Master Theorem to analyze the complexity of DC3.

- At each step, everything can be done in linear time thanks to constant comparison time between suffixes.
We can apply the Master Theorem to analyze the complexity of DC3

- At each step, everything can be done in linear time thanks to constant comparison time between suffixes
- Our recursion is bottlenecked by a call of $\frac{2}{3}$ size at each level
Analysis (DC3)

We can apply the Master Theorem to analyze the complexity of DC3:

- At each step, everything can be done in linear time thanks to constant comparison time between suffixes.
- Our recursion is bottlenecked by a call of $\frac{2}{3}$ size at each level.
- $T(n) = T(2n/3) + O(n)$
  - Solving yields $T(n) = O(n)$ overall.
Analysis (Generalization)

➔ Sample suffixes $S_C$ we used in DC3 is a special case of a difference cover sample
Analysis (Generalization)

➔ Sample suffixes $S_C$ we used in DC3 is a special case of a difference cover sample

- Defined by two *sample conditions*
  - 1. Sample itself can be sorted efficiently
  - 2. The sorted sample helps in sorting the total suffix set
Analysis (Generalization)

→ Sample suffixes $S_C$ we used in DC3 is a special case of a difference cover sample
  ● Defined by two *sample conditions*
    ● 1. Sample itself can be sorted efficiently
    ● 2. The sorted sample helps in sorting the total suffix set

→ DC3 uses a difference cover sample modulo 3
Analysis (Generalization)

➔ Sample suffixes $S_C$ we used in DC3 is a special case of a difference cover sample
  ● Defined by two sample conditions
    ● 1. Sample itself can be sorted efficiently
    ● 2. The sorted sample helps in sorting the total suffix set

➔ DC3 uses a difference cover sample modulo 3

➔ A generalized DC algorithm can use any difference cover modulo a given $v$
  ● Can show that the time complexity of this is $O(vn)$
Why do we care about a generalization when the time complexity appears to get worse?

- The more v increases, the longer the $O(vn)$ takes
Analysis (Generalization)

→ Why do we care about a generalization when the time complexity appears to get worse?

- The more $v$ increases, the longer the $O(vn)$ takes
- However, it also takes less space
  - DC can be implemented in $O(n/\sqrt{v})$ space by reusing the output array as temporary storage
Analysis (Generalization)

Why do we care about a generalization when the time complexity appears to get worse?
- The more $v$ increases, the longer the $O(vn)$ takes
- However, it also takes less space
  - DC can be implemented in $O(n/\sqrt{v})$ space by reusing the output array as temporary storage

Another key improvement given by DC: it is **space-efficient**
- Can also tune the parameter $v$ to control the space- and time-efficiency tradeoff
DC3 can be adapted for different models of computation as well

- Efficient in external memory usage
- Cache obliviousness
- EREW/CRCW PRAM
- etc.
Reflection (Strengths)

➔ Really well written
  ● Interleaving of a general description of DC3 and examples
    ● Allows the reader to fully digest each step of the algorithm
  ● Follows DC3 up with a generalization to DC that highlights its strengths and flexibility
  ● Extends further to different computational models
➔ Includes source code in the appendix
➔ Explains all the terms it uses and refrains from using excessive amounts of jargon
Reflection (Weaknesses)

➔ Source code is somewhat hard to sift through since all the variable names are short
  • Could also have included snippets throughout the paper to further elucidate certain confusing steps
➔ Tables comparing with prior work are somewhat lengthy and hard to digest
Reflection (Future Work)

➔ Paper mentions that suffix array is commonly augmented with the lcp array (longest common prefix)
  ● Stores longest common prefix between adjacent suffixes SA_i and SA_{i+1}
    ● Note: these are not adjacent suffixes in the original string, but in the suffix array
    ● Doesn't fully explain a way to retrieve this as well, could be looked into further in a future paper

➔ Further optimizations regarding memory/time could be possible
Discussion Questions

➔ How would a suffix array be used to solve string matching problems? E.g. finding all occurrences of a string in another string.
➔ In what ways would a lcp array be a helpful augment to the suffix array?
➔ What specific kinds of problems/applications can you think of that suffix array would be helpful for?