Lecture 20: Text Indexing II

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: December 10, 2024

Announcements

1. **Today is the final lecture!!!**
2. Final exam revision materials soon:
    - Practice exam questions
    - Solutions
3. Attendance Code:

873741

Meeting Goals

1. Finish discussion of text indexing
    - Recap of suffix trees
    - Introduce suffix arrays
    - Introduce LCP arrays
    - Discuss efficient computation of suffix and LCP arrays
2. Final exam overview

Text Indexing

From Last Time

Text Indexing Problem. Given a text $T[0..n)$, *preprocess* T so that queries to T can be performed *efficiently*

- Pattern matching for any $P[0..m)$
- Approximate matching
- Matching with wildcards
- Find longest repeated substring
- …

From Last Time

Text Indexing Problem. Given a text $T[0..n)$, *preprocess* T so that queries to T can be performed *efficiently*

- Pattern matching for any $P[0..m)$
- Approximate matching
- Matching with wildcards
- Find longest repeated substring
- …

Remarkably Useful Tool. *Suffix Trees!*

- Form **compact trie** of all suffixes of T: $T[0..n]$, $T[1..n]$, $T[2..n], \ldots, T[n..n]$
- Given the suffix tree $\mathcal{T}$, all of the examples above can be computed efficiently!

From Last Time

Text Indexing Problem. Given a text $T[0..n)$, *preprocess* T so that queries to T can be performed *efficiently*

- Pattern matching for any $P[0..m)$
- Approximate matching
- Matching with wildcards
- Find longest repeated substring
- …

Remarkably Useful Tool. *Suffix Trees!*

- Form **compact trie** of all suffixes of T: $T[0..n]$, $T[1..n]$, $T[2..n], \ldots, T[n..n]$
- Given the suffix tree $\mathcal{T}$, all of the examples above can be computed efficiently!

Question. Can we compute $\mathcal{T}$ from T efficiently?

From Last Time

Text Indexing Problem. Given a text $T[0..n)$, *preprocess* T so that queries to T can be performed *efficiently*

- Pattern matching for any $P[0..m)$
- Approximate matching
- Matching with wildcards
- Find longest repeated substring
- …

Remarkably Useful Tool. *Suffix Trees!*

- Form **compact trie** of all suffixes of T: $T[0..n]$, $T[1..n]$, $T[2..n], \ldots, T[n..n]$
- Given the suffix tree $\mathscr{T}$, all of the examples above can be computed efficiently!

Question. Can we compute $\mathscr{T}$ from T efficiently? **Today: Yes!**

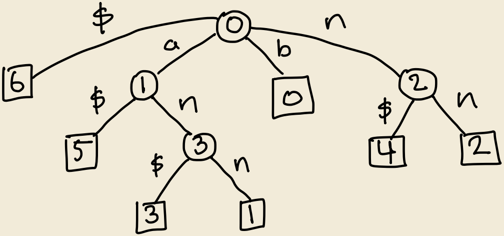
Banana Example

Example. $T = \texttt{banana\$}$.

Suffix Trees and Sorting Suffixes

Question. Consider the *pre-order traversal* of the leaves in the suffix tree. In what order are the corresponding suffixes?

0	1	2	3	4	5	6
b	a	n	a	n	a	$
	a	n	a	n	a	$
		n	a	n	a	$
			a	n	a	$
				n	a	$
					a	$
						$



Suffix Trees and Sorting Suffixes

Question. Consider the *pre-order traversal* of the leaves in the suffix tree. In what order are the corresponding suffixes?

Observation. The suffixes are sorted in *lexicographical order*.

Suffix Trees and Sorting Suffixes

Question. Consider the *pre-order traversal* of the leaves in the suffix tree. In what order are the corresponding suffixes?

Observation. The suffixes are sorted in *lexicographical order*.

- This is already sufficient to perform string matching with pattern $P[0..m)$ reasonably efficiently
  - $O(m \log n)$ time
  - Not much worse than $O(m)$ for string matching with suffix array
  - Still want to do better

Suffix Trees and Sorting Suffixes

Question. Consider the *pre-order traversal* of the leaves in the suffix tree. In what order are the corresponding suffixes?

Observation. The suffixes are sorted in *lexicographical order*.

- This is already sufficient to perform string matching with pattern $P[0..m)$ reasonably efficiently
  - $O(m \log n)$ time
  - Not much worse than $O(m)$ for string matching with suffix array
  - Still want to do better

Question. Can we perform suffix tree-type computations without computing the full suffix array?

Suffix Trees and Sorting Suffixes

Question. Consider the *pre-order traversal* of the leaves in the suffix tree. In what order are the corresponding suffixes?

Observation. The suffixes are sorted in *lexicographical order*.

- This is already sufficient to perform string matching with pattern $P[0..m)$ reasonably efficiently
  - $O(m \log n)$ time
  - Not much worse than $O(m)$ for string matching with suffix array
  - Still want to do better

Question. Can we perform suffix tree-type computations without computing the full suffix array?

Definition. The **suffix array**, $L[0..n]$ of $T[0..n]$ is the array of indices of the suffixes of T when the suffixes are sorted in lexicographic order.

- This is the same as pre-order traversal of the leaves of $\mathcal{T}$.

Suffix Array Example

Example. Compute the suffix array L for $T = \texttt{abbabbaa\$}$.

Suffix Array Example

Example. Compute the suffix array L for $T =$ `abbabbaa$`.

`abbabbaa$`	`$`	8
`bbabbaa$`	`a$`	7
`babbaa$`	`aa$`	6
`abbaa$`	`abbaa$`	3
`bbaa$`	`abbabbaa$`	0
`baa$`	`baa$`	5
`aa$`	`babbaa$`	2
`a$`	`bbaa$`	4
`$`	`bbabbaa$`	1

So. $L = [8, 7, 6, 3, 0, 5, 2, 4, 1]$

Is Too Much Lost?

Question. Is the suffix array L (together with T) sufficient to perform queries efficiently?

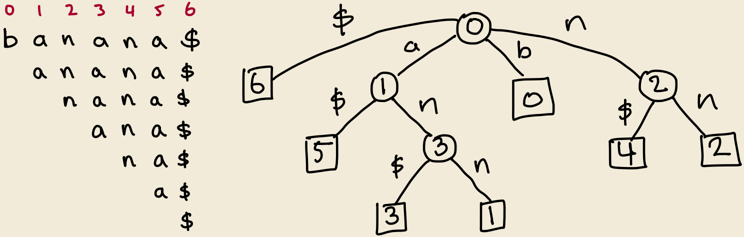
- Somewhat for string matching!
- Maybe not for longest repeated substring
  - required knowledge of *internal structure* of the suffix tree $\mathcal{T}$

What additional structure of $\mathcal{T}$ might we need to store?

Sufficient Tree Structure. Consider the suffix tree $\mathcal{T}$ for a text $T[0..n]$. The **longest common prefix array** $LCP[1..n]$ stores at index i the length of the longest common prefix of $T[L[i]..n]$ and $T[L[i-1]..n]$

LCP Array Example

Example. Compute the LCP array for the text $T = \texttt{banana\$}$.



Sufficient Information

Fact. Given $T[0..n]$, L, and LCP, it is possible to compute $\mathscr{T}$ in time $\Theta(n)$.

Sufficient Information

Fact. Given $T[0..n]$, L, and LCP, it is possible to compute $\mathscr{T}$ in time $\Theta(n)$.

Illustration. Construct $\mathscr{T}$ for T = `banana$` from L and LCP:

- T = `banana$`
- $L = [6, 5, 3, 1, 0, 4, 2]$
- $LCP = [0, 1, 3, 0, 0, 2]$

Sufficient Information

Fact. Given $T[0..n]$, L, and LCP, it is possible to compute $\mathscr{T}$ in time $\Theta(n)$.

Illustration. Construct $\mathscr{T}$ for $T =$ `banana$` from L and LCP:

- $T =$ `banana$`
- $L = [6, 5, 3, 1, 0, 4, 2]$
- $LCP = [0, 1, 3, 0, 0, 2]$

Consequence. In order to compute $\mathscr{T}$ in $O(n)$ time, it suffices to compute L and LCP in $O(n)$ time.

One More Definition

Definition. Given a suffix array L, the **inverse suffix array** or **rank array** R is defined by $L[r] = i \iff R[j] = r$.

- R is the inverse permutation of L
- $R[i]$ gives the (sorted) *rank* of the suffix $T[i..n]$
- R and L can be computed from one another in linear time
  - Example: $L = [6, 5, 3, 1, 0, 4, 2]$

One More Definition

Definition. Given a suffix array L, the **inverse suffix array** or **rank array** R is defined by $L[r] = i \iff R[j] = r$.

- R is the inverse permutation of L
- $R[i]$ gives the (sorted) *rank* of the suffix $T[i..n]$
- R and L can be computed from one another in linear time
  - Example: $L = [6, 5, 3, 1, 0, 4, 2]$
- To compute L, it suffices to compute R (efficiently)

So. To compute $\mathscr{T}$, it suffices to compute R and LCP.

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

A Non-obvious Approach.

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

A Non-obvious Approach.

1. Compute a rank array $R_{1,2}$ for $T_i = T[i..n]$ with i not divisible by 3 *recursively*.
   - Challenge: make recursive calls smaller instances of original problem

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

A Non-obvious Approach.

1. Compute a rank array $R_{1,2}$ for $T_i = T[i..n]$ with i not divisible by 3 *recursively*.
2. Use $R_{1,2}$ to find the rank array R_3 for suffix T_i with i divisible by 3
   - Trick: to compare T_0 and T_3, compare first characters. If they're the same use $R_{1,2}$ to compare T_1 and T_4

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

A Non-obvious Approach.

1. Compute a rank array $R_{1,2}$ for $T_i = T[i..n]$ with i not divisible by 3 *recursively*.
2. Use $R_{1,2}$ to find the rank array R_3 for suffix T_i with i divisible by 3
3. Merge $R_{1,2}$ and R_0
   - Similar to MERGESORT merge, but use Trick above to perform comparisons in $O(1)$ time

Computing R, An Overview

Goal. Given $T[0..n]$, compute $R[0..n]$ where $R[i]$ is the sorted rank of $T[i..n]$ among all prefixes of T.

A Non-obvious Approach.

1. Compute a rank array $R_{1,2}$ for $T_i = T[i..n]$ with i not divisible by 3 *recursively*.
2. Use $R_{1,2}$ to find the rank array R_3 for suffix T_i with i divisible by 3
3. Merge $R_{1,2}$ and R_0

Analysis

- Can perform steps 2 and 3 in linear time
- Overall running time is

$$n + \frac{2}{3}n + \left(\frac{2}{3}\right)^2 n + \cdots + 1 \leq n \sum_{i \geq 0} \left(\frac{2}{3}\right)^i = 3n = \Theta(n).$$

Computing LCP, An Overview

Goal. Compute $LCP[1..n]$ where $LCP[i]$ is the length of the longest common prefix of $T_{L[i]}$ and $T_{L[i-1]}$.

Computing LCP, An Overview

Goal. Compute $LCP[1..n]$ where $LCP[i]$ is the length of the longest common prefix of $T_{L[i]}$ and $T_{L[i-1]}$.

Observation. If $LCP[i] = \ell$, then there are two other prefixes of length $\ell - 1$

- namely, if $r = R[i]$ then T_{r+1} maches some string to at least $\ell - 1$ characters

T = bananaban$

sorted suffixes:

⋮

ban$

bananaban$

⋮

Computing LCP, An Overview

Goal. Compute $LCP[1..n]$ where $LCP[i]$ is the length of the longest common prefix of $T_{L[i]}$ and $T_{L[i-1]}$.

Observation. If $LCP[i] = \ell$, then there are two other prefixes of length $\ell - 1$

- namely, if $r = R[i]$ then T_{r+1} maches some string to at least $\ell - 1$ characters

Efficient Procedure.

- Compute L and R (in $O(n)$) time

Computing LCP, An Overview

Goal. Compute $LCP[1..n]$ where $LCP[i]$ is the length of the longest common prefix of $T_{L[i]}$ and $T_{L[i-1]}$.

Observation. If $LCP[i] = \ell$, then there are two other prefixes of length $\ell - 1$

- namely, if $r = R[i]$ then T_{r+1} maches some string to at least $\ell - 1$ characters

Efficient Procedure.

- Compute L and R (in $O(n)$) time
- Process prefixes in descending length order $i = 0, 1, 2, \ldots, n-1$
  - Find the rank r of T_i
  - Find LCP of T_i and T_j with $j = L[r-1]$
    - must be at least LCP corresponding T_{i-1} minus 1

Computing LCP, An Overview

Goal. Compute $LCP[1..n]$ where $LCP[i]$ is the length of the longest common prefix of $T_{L[i]}$ and $T_{L[i-1]}$.

Observation. If $LCP[i] = \ell$, then there are two other prefixes of length $\ell - 1$

- namely, if $r = R[i]$ then T_{r+1} maches some string to at least $\ell - 1$ characters

Efficient Procedure.

- Compute L and R (in $O(n)$) time
- Process prefixes in descending length order $i = 0, 1, 2, \ldots, n-1$
  - Find the rank r of T_i
  - Find LCP of T_i and T_j with $j = L[r-1]$
    - must be at least LCP corresponding T_{i-1} minus 1

Conclusion. This can be performed in $O(n)$ time!

Concluding Thoughts

We have shown:

- Suffix trees can be used to preform many queries to T efficiently
- We can compute the following in linear time:
  - the suffix array L
  - the inverse suffix array (rank array) R
  - the LCP array LCP
- From these, we can compute $\mathscr{T}$ in time $O(n)$
- These are surprising (and relatively recent) developments!

Final Exam

From Day 1: Goals & Content

Module Goals:

- build / enhance your toolbox of algorithmic methods and techniques
  $\Longrightarrow$ focus on practical methods
- enable you to reason about and communicate algorithmic solutions
  $\Longrightarrow$ level of abstraction, proofs, mathematical analysis, vocabulary
- enable you to apply, combine and extend methods

Units:

1. Module Overview & Proof Techniques
2. Machines & Models
3. Fundamental Data Structures
4. Efficient Sorting
5. String Matching
6. Compression
7. Error-Correcting Codes
8. Parallel Algorithms
9. Text indexing
10. ~~Streaming Algorithms~~

From Day 1: Goals & Content

Module Goals:

- build / enhance your toolbox of algorithmic methods and techniques
  $\Longrightarrow$ focus on practical methods
- enable you to reason about and communicate algorithmic solutions
  $\Longrightarrow$ level of abstraction, proofs, mathematical analysis, vocabulary
- enable you to apply, combine and extend methods

Units:

1. Module Overview & Proof Techniques
2. Machines & Models
3. Fundamental Data Structures
4. Efficient Sorting
5. String Matching
6. Compression
7. Error-Correcting Codes
8. Parallel Algorithms
9. Text indexing
10. ~~Streaming Algorithms~~

Exam Purpose. Determine the extent to which you achieved these goals.

Exam Format

The Basics.

- Written Exam, Closed Book
  - 2 1/2 hours to complete (invigilated)
  - no outside resources: just you, pencil, and paper
- 100 marks total
- 5 multi-part questions, each worth 25 marks
- Total mark is sum of 4 highest marks
  - only need to answer 4 of 5 questions
- Content from all module units
- Focus on conceptual and computational aspects of module content

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
   - Example: Define the *compression ratio* of an encoding scheme and describe a scenario in which one of the compression algorithms from lecture gives a small compression ratio.

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
2. **Factual:** recall a pertinent fact about a particular concept or algorithm from lecture.
   - Example: What is the worst-case running time of MergeSort applied to an array of length n?

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
2. **Factual:** recall a pertinent fact about a particular concept or algorithm from lecture.
3. **Computational:** apply a known algorithm to a new input
   - Example: apply the Burrows-Wheeler transformation to the text $T = \texttt{mississippi\$}$.

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
2. **Factual:** recall a pertinent fact about a particular concept or algorithm from lecture.
3. **Computational:** apply a known algorithm to a new input
4. **Critical Analysis:** explain/analyze a concept and how it relates to another concept
   - Example: Consider the task of sorting an array of size n containing numbers from the range 1 to c for some constant c. Explain why the $O(n)$ running time of CountingSort does not contradict the $\Omega(n \log n)$ lower bound we proved for comparison based sorting algorithms.

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
2. **Factual:** recall a pertinent fact about a particular concept or algorithm from lecture.
3. **Computational:** apply a known algorithm to a new input
4. **Critical Analysis:** explain/analyze a concept and how it relates to another concept
5. **Transfer Task:** apply concepts or techniques from lecture to solve a novel problem.
   - Example: Two strings $S_1[0..n)$ and $S_2[0..n)$ are **anagrams** if they are rearrangements of precisely the same letters (with multiplicity). Describe a procedure that determines if two strings are anagrams in time $O(n \log n)$.

Question Types

1. **Definitional:** concisely define a concept from class together with examples or applications of the concept
2. **Factual:** recall a pertinent fact about a particular concept or algorithm from lecture.
3. **Computational:** apply a known algorithm to a new input
4. **Critical Analysis:** explain/analyze a concept and how it relates to another concept
5. **Transfer Task:** apply concepts or techniques from lecture to solve a novel problem.

Assessment.

- **Pass** (50–60). Answer types 1–3 with only minor errors.
- **Merit** (60–70). Answer 1–3, and show some insight on 4–5.
- **Distinction** (70+). Answer 1–3 with significant progress on 4–5.

Forthcoming

Lecture Review Materials

- Exhaustive list of topics
- Example questions
- Model solutions

Forthcoming

Lecture Review Materials

- Exhaustive list of topics
- Example questions
- Model solutions

PollEverywhere

In what format do you find example solutions most helpful?

- thorough written (typeset) solution
- a video walking through solutions (handwritten)
- either one is fine

pollev.com/comp526

Forthcoming

Lecture Review Materials

- Exhaustive list of topics
- Example questions
- Model solutions

Marking

- Programming Assignment 1
- Programming Assignment 2

Thank You!!!

Scratch Notes