Web Data Management

Compression and Search

Lecture 3: Search and basic indexing
What is Pattern Matching?

• Definition:
  – given a text string $T$ and a pattern string $P$, find the pattern inside the text
    • $T$: “the rain in spain stays mainly on the plain”
    • $P$: “n th”
The Brute Force Algorithm

- Check each position in the text T to see if the pattern P starts in that position.
Analysis

• Brute force pattern matching runs in time $O(mn)$ in the worst case.

• But most searches of ordinary text take $O(m+n)$, which is very quick.
• The brute force algorithm is fast when the alphabet of the text is large
  – e.g. A..Z, a..z, 1..9, etc.

• It is slower when the alphabet is small
  – e.g. 0, 1 (as in binary files, image files, etc.)
• Example of a worst case:
  – T: "aaaaaaaaaaaaaaaaaaaaaaaaaaah"
  – P: "aaah"

• Example of a more average case:
  – T: "a string searching example is standard"
  – P: "store"
The KMP Algorithm

• The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a left-to-right order (like the brute force algorithm).

• But it shifts the pattern more intelligently than the brute force algorithm.
Summary

• If a mismatch occurs between the text and pattern P at P[j], what is the *most* we can shift the pattern to avoid wasteful comparisons?
Summary

• If a mismatch occurs between the text and pattern P at P[j], what is the most we can shift the pattern to avoid wasteful comparisons?

• *Answer*: the largest prefix of P[0 .. j-1] that is a suffix of P[1 .. j-1]
Example

T: \[ \text{a b a c a a b a c c a b a c a b a a b b} \]

P: \[ \text{a b a c a a b} \]

Example

T:

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>23</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>F(k)</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Example

P:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
</table>

Example

T:

<table>
<thead>
<tr>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>

Example

P:

<table>
<thead>
<tr>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
</table>

Example
KMP Advantages

• KMP runs in optimal time: $O(m+n)$
  – very fast

• The algorithm never needs to move backwards in the input text, T
  – this makes the algorithm good for processing very large files that are read in from external devices or through a network stream
KMP Disadvantages

• KMP doesn’t work so well as the size of the alphabet increases
  – more chance of a mismatch (more possible mismatches)
  – mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later
The Boyer-Moore Algorithm
The Boyer-Moore Algorithm

• The Boyer-Moore pattern matching algorithm is based on two techniques.

• 1. The looking-glass technique
  – find P in T by moving backwards through P, starting at its end
• 2. The *character-jump* technique
  – when a mismatch occurs at $T[i] == x$
  – the character in pattern $P[j]$ is not the same as $T[i]$

• There are 3 possible cases.
Case 1

- If P contains x somewhere, then try to *shift P* right to align the last occurrence of x in P with T[i].

\[ T \]
\[ x \ a \]
\[ i \]

\[ P \]
\[ x \ c \ b \ a \]
\[ j \]

\[ \text{and move } i \text{ and } j \text{ right, so } j \text{ at end} \]

\[ T \]
\[ x \ a \ ? \ ? \]
\[ i_{\text{new}} \]

\[ P \]
\[ x \ c \ b \ a \]
\[ j_{\text{new}} \]
Case 2

- If $P$ contains $x$ somewhere, but a shift right to the last occurrence is *not* possible, then *shift $P$ right by 1 character to $T[i+1]$.*
Case 3

- If cases 1 and 2 do not apply, then shift $P$ to align $P[0]$ with $T[i+1]$.

\[\text{No } x \text{ in } P\]
Boyer-Moore Example (1)

T:

a p a t t e r n m a t c h i n g a l g o r i t h m

r i t h m r i t h m r i t h m r i t h m

P:

r i t h m r i t h m r i t h m r i t h m
Last Occurrence Function

• Boyer-Moore’s algorithm preprocesses the pattern P and the alphabet A to build a last occurrence function L()
  – L() maps all the letters in A to integers

• L(x) is defined as: // x is a letter in A
  – the largest index i such that P[i] == x, or
  – -1 if no such index exists
L() Example

- $A = \{a, b, c, d\}$
- $P$: "abacab"

\[
\begin{array}{c|c|c|c|c|c}
 x & a & b & c & d \\
 L(x) & 4 & 5 & 3 & -1 \\
\end{array}
\]

$L()$ stores indexes into $P[]$
Boyer-Moore Example (2)

T: $\text{a b a c a a b a d c a b a c a b a a b b}$

P: $\text{a b a c a b}$

<table>
<thead>
<tr>
<th>$x$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(x)$</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>
Analysis

- Boyer-Moore worst case running time is $O(nm + A)$

- But, Boyer-Moore is fast when the alphabet (A) is large, slow when the alphabet is small.
  - e.g. good for English text, poor for binary

- Boyer-Moore is significantly faster than brute force for searching English text.
Worst Case Example

- T: "aaaaa...a"
- P: "baaaaaa"
Regular Expressions

- Notation to specify a language
  - Declarative
  - Sort of like a programming language.
    - Fundamental in some languages like perl and applications like grep or lex
  - Capable of describing the same thing as a NFA
    - The two are actually equivalent, so RE = NFA = DFA
  - We can define an algebra for regular expressions
Definition of a Regular Expression

• R is a regular expression if it is:

1. a for some a in the alphabet \( \Sigma \), standing for the language \{a\}
2. \( \varepsilon \), standing for the language \{\varepsilon\}
3. \( \emptyset \), standing for the empty language
4. \( R_1 + R_2 \), where \( R_1 \) and \( R_2 \) are regular expressions, and + signifies union (sometimes | is used)
5. \( R_1 R_2 \), where \( R_1 \) and \( R_2 \) are regular expressions and this signifies concatenation
6. \( R^* \), where \( R \) is a regular expression and signifies closure
7. (R), where R is a regular expression, then a parenthesized R is also a regular expression

This definition may seem circular, but 1-3 form the basis
Precedence: Parentheses have the highest precedence, followed by *, concatenation, and then union.
Using Regular Expressions

• Regular expressions are a standard programmer's tool.
• Built in to Java, Perl, Unix, Python, . . . .
RE Examples

- \( L(001) = \{001\} \)
- \( L(0+10^*) = \{0, 1, 10, 100, 1000, 10000, \ldots\} \)
- \( L(0*10^*) = \{1, 01, 10, 010, 0010, \ldots\} \) i.e. \( \{w \mid w \text{ has exactly a single } 1\} \)
- \( L(\sum\sum)^* = \{w \mid w \text{ is a string of even length}\} \)
- \( L((0(0+1))^*) = \{\varepsilon, 00, 01, 0000, 0001, 0100, 0101, \ldots\} \)
- \( L((0+\varepsilon)(1+\varepsilon)) = \{\varepsilon, 0, 1, 01\} \)
- \( L(1\emptyset) = \emptyset \); concatenating the empty set to any set yields the empty set.
- \( R\varepsilon = R \)
- \( R+\emptyset = R \)

- Note that \( R+\varepsilon \) may or may not equal \( R \) (we are adding \( \varepsilon \) to the language)
- Note that \( R\emptyset \) will only equal \( R \) if \( R \) itself is the empty set.
Exercise 1

- Let $\Sigma$ be a finite set of symbols
- $\Sigma = \{10, 11\}$, $\Sigma^* = ?$
Answer: $\Sigma^* = \{\epsilon, 10, 11, 1010, 1011, 1110, 1111, \ldots \}$
Exercises 2

• \( L_1 = \{10, 1\}, \ L_2 = \{011, 11\}, \ L_1L_2 = ? \)
Answer

- \( L_1L_2 = \{10011, 1011, 111\} \)
Exercises 3

• Write RE for
  – All strings of 0’s and 1’s
  – All strings of 0’s and 1’s with at least 2 consecutive 0’s
  – All strings of 0’s and 1’s beginning with 1 and not having two consecutive 0’s
Answer

• \((0|1)^*\)  
All strings of 0’s and 1’s
• \((0|1)^*00(0|1)^*\)  
All strings of 0’s and 1’s with at least 2 consecutive 0’s
• \((1+10)^*\)  
All strings of 0’s and 1’s beginning with 1 and not having two consecutive 0’s
More Exercises

• 1) \((0|1)^*011\)
• 2) \(0^*1^*2^*\)
• 3) \(00^*11^*22^*\)
More Exercises (Answers)

1) $(0|1)^*011$
Answer: all strings of 0’s and 1’s ending in 011

2) $0^*1^*2^*$
- Answer: any number of 0’s followed by any number of 1’s followed by any number of 2’s

3) $00^*11^*22^*$
Answer: strings in $0^*1^*2$ with at least one of each symbol
NFA
Deterministic Finite Automata (DFA)

- Simple machine with $N$ states.
- Begin in start state.
- Read first input symbol.
- Move to new state, depending on current state and input symbol.
- Repeat until last input symbol read.
- Accept or reject string depending on label of last state.
DFA
Theory of DFAs and REs

- **RE.** Concise way to describe a set of strings.
- **DFA.** Machine to recognize whether a given string is in a given set.
- **Duality:** for any DFA, there exists a regular expression to describe the same set of strings; for any regular expression, there exists a DFA that recognizes the same set.
Duality Example

- DFA for multiple of 3 b’s:

- RE for multiple of 3 b’s:

\[(a*ba*ba*ba*)^* a^*\]
Fundamental Questions

• Which languages CANNOT be described by any RE?
• Set of all bit strings with equal number of 0s and 1s.
• Set of all decimal strings that represent prime numbers.
• Many more. . . .
Problem 1

• Make a DFA that accepts the strings in the language denoted by regular expression \( ab^*a \)
Solution

- $ab^*a$: 

![Diagram](attachment:image.png)
Problem 2

• Write the RE for the following automata:
Solution

- $a(a|b)^*a$
DFA to RE: State Elimination

- Eliminates states of the automaton and replaces the edges with regular expressions that includes the behavior of the eliminated states.
- Eventually we get down to the situation with just a start and final node, and this is easy to express as a RE
State Elimination

- Consider the figure below, which shows a generic state $s$ about to be eliminated.
- The labels on all edges are regular expressions.
- To remove $s$, we must make labels from each $q_i$ to $p_1$ up to $p_m$ that include the paths we could have made through $s$. 
DFA to RE via State Elimination (1)

- Starting with intermediate states and then moving to accepting states, apply the state elimination process to produce an equivalent automaton with regular expression labels on the edges.
- The result will be a one or two state automaton with a start state and accepting state.
DFA to RE State Elimination (2)

- If the two states are different, we will have an automaton that looks like the following:

- We can describe this automaton as: \((R | SU*T)^*SU^*\)
DFA to RE State Elimination (3)

- If the start state is also an accepting state, then we must also perform a state elimination from the original automaton that gets rid of every state but the start state. This leaves the following:

![Diagram of an automaton with a start state and a loop]

- We can describe this automaton as simply $R^*$
DFA to RE State Elimination (4)

- If there are $n$ accepting states, we must repeat the above steps for each accepting state to get $n$ different regular expressions, $R_1, R_2, \ldots, R_n$.
- For each repeat we turn any other accepting state to non-accepting.
- The desired regular expression for the automaton is then the union of each of the $n$ regular expressions: $R_1 \cup R_2 \ldots \cup R_n$. 
DFA->RE Example

- Convert the following to a RE:

- First convert the edges to RE’s:
DFA -> RE Example (2)

- Eliminate State 1:

- Note edge from 3->3

- Answer: \((0+10)^*11(0+1)^*\)
Second Example

- Automata that accepts even number of 1’s

- Eliminate state 2:
Second Example (2)

- Two accepting states, turn off state 3 first

- This is just $0^*$; can ignore going to state 3 since we would “die”
Second Example (3)

- Turn off state 1 second:

- This is just $0^*10^*1(0|10^*1)^*$
- Combine from previous slide to get $0^* | 0^*10^*1(0|10^*1)^*$
Text search

- Pattern matching directly
  - Brute force
  - BM
  - KMP
- Regular expressions
- Indices for pattern matching
  - Inverted files
  - Signature files
  - Suffix trees and Suffix arrays
Inverted Index

For each term $t$, we store a list of all documents that contain $t$.

- **BRUTUS** → 1 2 4 11 31 45 173 174
- **CAESAR** → 1 2 4 5 6 16 57 132 ...
- **CALPURNIA** → 2 31 54 101

...
Create postings lists, determine document frequency
Positional indexes

- Postings lists in a **nonpositional** index: each posting is just a docID
- Postings lists in a **positional** index: each posting is a docID and a list of positions
Positional indexes: Example

Query: “to\textsubscript{1} be\textsubscript{2} or\textsubscript{3} not\textsubscript{4} to\textsubscript{5} be\textsubscript{6}”

TO, 993427:
\begin{align*}
\langle 1 \rangle & : \langle 7, 18, 33, 72, 86, 231 \rangle; \\
\langle 2 \rangle & : \langle 1, 17, 74, 222, 255 \rangle; \\
\langle 4 \rangle & : \langle 8, 16, 190, 429, 433 \rangle; \\
\langle 5 \rangle & : \langle 363, 367 \rangle; \\
\langle 7 \rangle & : \langle 13, 23, 191 \rangle; \ldots
\end{align*}

BE, 178239:
\begin{align*}
\langle 1 \rangle & : \langle 17, 25 \rangle; \\
\langle 4 \rangle & : \langle 17, 191, 291, 430, 434 \rangle; \\
\langle 5 \rangle & : \langle 14, 19, 101 \rangle; \ldots
\end{align*}

Document 4 is a match!
Signature files

• Definition
  – Word-oriented index structure based on hashing.
  – Use liner search.
  – Suitable for not very large texts.

• Structure
  – Based on a Hash function that maps words to bit masks.
  – The text is divided in blocks.
    • Bit mask of block is obtained by bitwise ORing the signatures of all the words in the text block.
    • Word not found, if no match between all 1 bits in the query mask and the block mask.
Signature files

• Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many words. Words are made from letters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 000101 | 110101 | 100100 | 101101 |

Text signature

<table>
<thead>
<tr>
<th>Signature function</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(text) = 000101</td>
</tr>
<tr>
<td>h(many) = 110000</td>
</tr>
<tr>
<td>h(words) = 100100</td>
</tr>
<tr>
<td>h(made) = 001000</td>
</tr>
<tr>
<td>h(letters) = 000010</td>
</tr>
</tbody>
</table>

Signature function
Signature files

• Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text</td>
<td>A text has many words. Words are made from letters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 000101 | 110101 | 100100 | 101101 |

**Text signature**

$h(\text{text}) = 000101$
$h(\text{many}) = 110000$
$h(\text{words}) = 100100$
$h(\text{made}) = 001100$
$h(\text{letters}) = 100001$

**Signature function**
Signature files

- Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text</td>
<td>A text has many words. Words are made from letters</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 000101 | 110101 | 100100 | 101101 |

Text signature

h(text) = 000101
h(many) = 110000
h(words) = 100100
h(made) = 001100
h(letters) = 100001

Signature function
Signature files

- Example:

```
<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many words.</td>
<td>Words are made from letters</td>
<td></td>
</tr>
</tbody>
</table>

| 000101 | 110101 | 100100 | 101101 |
```

- **Text signature**

  - h(text) = 000101
  - h(many) = 110000
  - h(words) = 100100
  - h(made) = 001100
  - h(letters) = 100001

- **Signature function**
## Signature files

- **Example:**

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many</td>
<td>words. Words are made from letters</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature function</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(\text{text}) ) = 000101</td>
<td></td>
</tr>
<tr>
<td>( h(\text{many}) ) = 110000</td>
<td></td>
</tr>
<tr>
<td>( h(\text{words}) ) = 100100</td>
<td></td>
</tr>
<tr>
<td>( h(\text{made}) ) = 001100</td>
<td></td>
</tr>
<tr>
<td>( h(\text{letters}) ) = 100001</td>
<td></td>
</tr>
</tbody>
</table>
Signature files

• Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many words.</td>
<td>Words are made from letters.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h(text)</th>
<th>h(many)</th>
<th>h(words)</th>
<th>h(made)</th>
<th>h(letters)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000101</td>
<td>110101</td>
<td>100100</td>
<td>101101</td>
<td>100001</td>
</tr>
</tbody>
</table>

Text signature

Signature function
Signature files

- Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text. A text has many words. Words are made from letters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000101</td>
<td>110101</td>
<td>100100</td>
<td>101101</td>
</tr>
</tbody>
</table>

The text signature is calculated as follows:

- \( h(\text{text}) = 000101 \)
- \( h(\text{many}) = 110000 \)
- \( h(\text{words}) = 100100 \)
- \( h(\text{made}) = 001100 \)
- \( h(\text{letters}) = 100001 \)
Signature files

• Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text. A text has many words. Words are made from letters.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h(text)</td>
<td>h(many)</td>
<td>h(words)</td>
<td>h(made)</td>
</tr>
<tr>
<td>000101</td>
<td>110000</td>
<td>100100</td>
<td>001100</td>
</tr>
</tbody>
</table>

Text signature

Signature function
Signature files

• Example:

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many words.</td>
<td>Words are made from letters.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block 3</th>
<th>block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>000101</td>
<td>110101</td>
<td>100100</td>
<td>101101</td>
</tr>
</tbody>
</table>

$h(\text{text}) = 000101$
$h(\text{many}) = 110000$
$h(\text{words}) = 100100$
$h(\text{made}) = 001100$
$h(\text{letters}) = 100001$
Signature files

• False drop Problem
  – The corresponding bits are set even though the word is not there!
  – The design should insure that the probability of false drop is low.
    • Also the Signature file should be as short as possible.
  – Enhance the hashing function to minimize the error probability.
Signature files

• Searching
  1. For a single word, Hash word to a bit mask $W$.
  2. For phrases,
     1) Hash words in query to a bit mask.
     2) Bitwise OR of all the query masks to a bit mask $W$.
  3. Compare $W$ to the bit masks $B_i$ of all the text blocks.
     • If all the bits set in $W$ are also in $B_i$, then text block may
       contain the word.
  4. For all candidate text blocks, an online traversal must
     be performed to verify if the actual matches are there.

• Construction
  1. Cut the text in blocks.
  2. Generate an entry of the signature file for each block.
     • This entry is the bitwise OR of the signatures of all the words
       in the block.
Suffix trees and suffix arrays
Trie

• A tree representing a set of strings.

{ aeef
  ad
  bbfe
  bbfg
  c  }
Trie (Cont)

- Assume no string is a prefix of another

Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.
Compressed Trie

- Compress unary nodes, label edges by strings
Suffix tree

Given a string $s$ a suffix tree of $s$ is a compressed trie of all suffixes of $s$

To make these suffixes prefix-free we add a special character, say $\$$, at the end of $s$
Let $s=abab$, a suffix tree of $s$ is a compressed trie of all suffixes of $s=abab$
Trivial algorithm to build a Suffix tree

Put the largest suffix in

Put the suffix \texttt{bab$} in
Put the suffix \textit{ab$} in
Put the suffix $b\$ in
Put the suffix $ in
We will also label each leaf with the starting point of the corres. suffix.
Analysis

Takes $O(n^2)$ time to build.

We will see how to do it in $O(n)$ time
What can we do with it?

Exact string matching:

Given a Text $T$, $|T| = n$, preprocess it such that when a pattern $P$, $|P| = m$, arrives you can quickly decide when it occurs in $T$.

We may also want to find all occurrences of $P$ in $T$. 

78
Exact string matching

In preprocessing we just build a suffix tree in $O(n)$ time

Given a pattern $P = ab$ we traverse the tree according to the pattern.
Exact string matching

In preprocessing we just build a suffix tree in O(n) time

Given a pattern $P = \text{ab}$ we traverse the tree according to the pattern.
If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all $k$ occurrences in $O(n+k)$ time.
Generalized suffix tree

Given a set of strings $S$ a generalized suffix tree of $S$ is a compressed trie of all suffixes of $s \in S$

To make these suffixes prefix-free we add a special char, say $\$, at the end of $s$

To associate each suffix with a unique string in $S$ add a different special char to each $s$
Let $s_1=abab$ and $s_2=aab$ here is a generalized suffix tree for $s_1$ and $s_2$. 

```plaintext
{  
  $ \quad \#$  
  b$ \quad b#$  
  ab$ \quad ab#$  
  bab$ \quad aab#$  
  abab$  
}
```
So what can we do with it?

Matching a pattern against a database of strings
Longest common substring (of two strings)

Every node with a leaf descendant from string $S_1$ and a leaf descendant from string $S_2$ represents a maximal common substring and vice versa.

Find such node with largest “string depth”
Longest common substring (of two strings)

Every node with a leaf descendant from string $S_1$ and a leaf descendant from string $S_2$ represents a maximal common substring and vice versa.

Find such node with largest “string depth”
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it.
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it
Lowest common ancestor

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes.
Finding maximal palindromes

• A palindrome: caabaac, cbaabc
• Want to find all maximal palindromes in a string $s$

Let $s = cbaaba$

The maximal palindrome with center between $i-1$ and $i$ is the LCP of the suffix at position $i$ of $S$ and the suffix at position $m-i+1$ of $Sr$
Maximal palindromes algorithm

Prepare a generalized suffix tree for
$s = \text{cbaaba}$ and $s_r = \text{abaabc}$

For every $i$ find the LCA of suffix $i$ of $s$ and
suffix $m-i+1$ of $s_r$
Let $s = cbaaba$ then $sr = abaabc#$
Let $s = \text{cbaaba}$ then $s_r = \text{abaabc}$
Let \( s = \text{cbaaba}$ then \( s_r = \text{abaabc#} \)
Analysis

$O(n)$ time to identify all palindromes
Drawbacks

• Suffix trees consume a lot of space

• It is $O(n)$ but the constant is quite big

• Notice that if we indeed want to traverse an edge in $O(1)$ time then we need an array of ptrs. of size $|\Sigma|$ in each node
Suffix array

• We lose some of the functionality but we save space.

Let  \( s = abab \)

Sort the suffixes lexicographically: \( ab, abab, b, bab \)

The suffix array gives the indices of the suffixes in sorted order

\[
\begin{array}{c|c|c|c}
3 & 1 & 4 & 2 \\
\end{array}
\]
How do we build it?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- $O(n)$ time
How do we search for a pattern?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes $O(m \log n)$ time
Example

Let $S = \text{mississippi}$

Let $P = \text{issa}$
Supra index

- Structure
  - **Suffix arrays** are space efficient implementation of **suffix trees**.
  - Simply an array **containing all the pointers** to the text suffixes listed in lexicographical order.
  - **Supra-indices:**
    - If the suffix array is **large**, this binary search can perform **poorly** because of the number of random disk accesses.
    - Suffix arrays are designed to allow **binary searches** done by comparing the contents of each pointer.
    - To remedy this situation, the use of **supra-indices** over the suffix array has been proposed.
Supra index

- Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>6</th>
<th>9</th>
<th>11</th>
<th>17</th>
<th>19</th>
<th>24</th>
<th>28</th>
<th>33</th>
<th>40</th>
<th>46</th>
<th>50</th>
<th>55</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This is a text. A text has many words. Words are made from letters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>60</th>
<th>50</th>
<th>28</th>
<th>19</th>
<th>11</th>
<th>40</th>
<th>33</th>
<th>SuffixArray</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>lett</th>
<th>text</th>
<th>word</th>
<th>SuffixArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>50</td>
<td>28</td>
<td>19</td>
</tr>
</tbody>
</table>
This is a text. A text has many words. Words are made from letters.