Text indexing

Preprocess a text string $T$ (length $n$) to answer pattern matching queries...

$T = \text{a b c a b a c b a b}$

After preprocessing, a query is a pattern $P$ (length $m$), the output is a list of all matches in $T$.

- Last lecture we saw the suffix tree which uses $O(n)$ space
- Queries take $O(m + \text{occ})$ time when the alphabet size is constant
  - $\text{occ}$ is the number of occurrences (matches)
- We saw an $O(n^2)$ time construction algorithm (an $O(n)$ time algorithm also exists)

The suffix array

Sort the suffixes lexicographically

- The symbols must have an order throughout we will use alphabetical

In lexicographical ordering we sort strings based on the first symbol that differs:

$\text{a a} < \text{b a} < \text{b c} < \text{b c a}$

(In a 'tie', the shorter string is smaller)

If the symbols don’t have a natural order, we use their binary representation in memory

Constructing the Suffix Array from the Suffix Tree

Recall that we added a unique symbol $\$\$ to make sure the tree exists

- the $\$\$ is the smallest symbol in the alphabet

To get the Suffix array perform a depth-first search (in lexicographical order)

this takes $O(n)$ time

Searching in the Suffix Array

$\text{find } P = \text{a b c a b a c}$

Key Idea: Find an occurrence of $P$ using binary search

How long does this take?

$O(m)$ time per step

so $O(m \log n)$ time in total

This method generalises to $O(m \log n + \text{occ})$ time
to find all $\text{occ}$ occurrences. (we skip the details)
LCP - the Longest Common Prefix

For any pair of locations \((i, j)\), we define \(\text{LCP}(i, j)\) to be the largest \(d\) such that
\[T[i + d - 1] = T[j + d - 1]\]
it's the furthest you can go before hitting a mismatch
or equivalently it's the length of the longest common prefix of suffix \(i\) and suffix \(j\)

Notation
\[T[x . . . y]\] is the substring of \(T\) starting at the \(x\)-th location and ending at the \(y\)-th location (inclusive)
\[T[x]\text{ is the } x\text{-th character of } T\]

Faster searching in Suffix Arrays

- While binary searching the suffix array for \(P\), keep track of:
  \(\ell\) - the longest common prefix of \(P\) and \(T[L . . . n - 1]\)
  \(r\) - the longest common prefix of \(P\) and \(T[R . . . n - 1]\)

\[P \quad 4 \quad 9 \quad 2 \quad 5 \quad 7 \quad 2 \quad 8 \quad \ldots\]
\[L \quad 6 \quad 48 \quad 35 \quad 71 \quad 86 \quad 14 \quad 22 \quad 12\]
\[\ell = 4\]

Because \(\ell = r\), every suffix between \(L\) and \(R\) matches \(P\) on the first \(\ell\) characters.

So we can start comparing suffix \(M\) from character \(\ell + 1\)

Finally we update \(L, M, R\) and \(\ell, r\)

what about when \(\ell \neq r\)?
Faster searching in Suffix Arrays

- While binary searching keep track of:
  - \( \ell \) - the longest common prefix of \( P \) and \( T[L \ldots n-1] \)
  - \( r \) - the longest common prefix of \( P \) and \( T[R \ldots n-1] \)

Assume that \( \ell > r \) (or \( \ell \) is symmetric)

Subcase 2: \( \text{LCP}(L, M) < \ell \)

<table>
<thead>
<tr>
<th>LCP(L, M) = 5</th>
<th>L</th>
<th>M</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>25, 6</td>
<td>48</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>( \ell = 5 )</td>
<td>( \text{LCP}(L, M) = 4 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Every suffix starting with the first \( \ell \) characters of \( P \)...

Update \( L, M \), and \( R \) when you hit a mismatch.
- If \( r > \ell \), \( r \) becomes the previous \( \text{LCP}(L, M) \) \( R \) moves to \( M \) (and \( M \) moves to the new middle)

Subcase 3: \( \text{LCP}(L, M) = \ell \)

<table>
<thead>
<tr>
<th>LCP(L, M) = 4</th>
<th>L</th>
<th>M</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>25, 6</td>
<td>48</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>( \ell = 4 )</td>
<td>( \text{LCP}(L, M) = 4 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compare \( P \) to \( M \) from character \( \ell + 1 \)...

When you hit a mismatch, rule out either the range \( L \) to \( M \) or, the range \( M \) to \( R \).

Subcase 3: \( \text{LCP}(L, M) = \ell \)

<table>
<thead>
<tr>
<th>LCP(L, M) = 4</th>
<th>L</th>
<th>M</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>25, 6</td>
<td>48</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>( \ell = 4 )</td>
<td>( \text{LCP}(L, M) = 4 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compare \( P \) to \( M \) from character \( \ell + 1 \)...

When you hit a mismatch, rule out either the range \( L \) to \( M \) or, the range \( M \) to \( R \).

How long does it the faster search take?

- We perform a binary search on the suffix array (which has length \( n \))
- Therefore there are \( O(\log n) \) steps
  (a step is when we move \( M \) - the ‘middle’ suffix)
- In each step we compute \( \text{LCP}(L, M) \) and \( \text{LCP}(M, R) \) (in \( O(1) \) time)
- We then decide which subcase we are in

The cases all take \( O(1) \) time except...

When we compare many characters of \( M \) to \( P \)

Crucially, inspection of the algorithm shows that we never look at the same character of \( P \) more than twice (equivalently \( \ell \) and \( r \) never decrease)

This implies that the time complexity is \( O(n \log n) \)

- \( \ell \) - the longest common prefix of \( P \) and \( T[L \ldots n-1] \)
- \( r \) - the longest common prefix of \( P \) and \( T[R \ldots n-1] \)

This method generalises to \( O(n \log n + \text{occ}) \) time to find all \( \text{occ} \) occurrences, (we skip the details)

\( m \) is the length of \( T \)

The suffix array

<table>
<thead>
<tr>
<th>( T )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Sort the suffixes lexicographically

Finding an occurrence of a pattern (length \( m \)) takes \( O(m + \log n) \) time

Finding all occurrences takes \( O(m + \log n + \text{occ}) \) time

where \( \text{occ} \) is the number of occurrences

Do we really need to build the suffix tree to construct the suffix array?
### The DC3 method

#### B1 contains indices with \( i \mod 3 = 1 \)

#### B2 contains indices with \( i \mod 3 = 2 \)

<table>
<thead>
<tr>
<th>T</th>
<th>R₁</th>
<th>R₂</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>yabbdbddo</td>
<td>abbdabddo</td>
<td>( y )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>yabbdbddo</td>
<td>( y ) + 1</td>
<td>( y ) + 1</td>
<td>( y ) + 1</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
</tr>
<tr>
<td>5</td>
<td>dabbddo</td>
<td>( b ) + 4</td>
<td>( b ) + 4</td>
<td>( b ) + 4</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
</tr>
<tr>
<td>0</td>
<td>yabbdbddo</td>
<td>( a ) + 7</td>
<td>( a ) + 7</td>
<td>( a ) + 7</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>3</td>
<td>dabbddo</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>6</td>
<td>dabbddo</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>9</td>
<td>dodd</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
</tbody>
</table>

Each suffix \( i \in B_0 \) is represented by \( (T[i], r) \) where \( r \) is the rank of suffix \( (i + 1) \)

(The ranks are given by the array below)

#### Rank:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

Suffix array for just \( B_1 \cup B_2 \):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

How do we find the ordering of the suffixes from \( B_0 \)? (where \( i \mod 3 = 0 \))

### The DC3 method

#### B1 contains indices with \( i \mod 3 = 1 \)

#### B2 contains indices with \( i \mod 3 = 2 \)

<table>
<thead>
<tr>
<th>T</th>
<th>R₁</th>
<th>R₂</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>yabbdbddo</td>
<td>abbdabddo</td>
<td>( y )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>yabbdbddo</td>
<td>( y ) + 1</td>
<td>( y ) + 1</td>
<td>( y ) + 1</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
<td>( y )</td>
</tr>
<tr>
<td>5</td>
<td>dabbddo</td>
<td>( b ) + 4</td>
<td>( b ) + 4</td>
<td>( b ) + 4</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
<td>( b )</td>
</tr>
<tr>
<td>0</td>
<td>yabbdbddo</td>
<td>( a ) + 7</td>
<td>( a ) + 7</td>
<td>( a ) + 7</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>3</td>
<td>dabbddo</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>6</td>
<td>dabbddo</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
<tr>
<td>9</td>
<td>dodd</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a ) + 9</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
<td>( a )</td>
</tr>
</tbody>
</table>

Each suffix \( i \in B_0 \) is represented by \( (T[i], r) \) where \( r \) is the rank of suffix \( (i + 1) \)

(The ranks are given by the array below)

#### Rank:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

Suffix array for just \( B_1 \cup B_2 \):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

How do we merge these?
The DC3 method

$T = yabbadabaddo$

$B_1$ contains indices with $i \mod 3 = 1$

$B_2$ contains indices with $i \mod 3 = 2$

Merge them like in mergesort...

1

which is smaller, suffix 1 or 6?

$e = a + 7$ (a, 4)

$1 = a + 1$ (a, 3)

It takes $O(1)$ time to decide that 1 is smaller

Suffix array for just $B_1 \cup B_2$

1 2 3 4 5 6 7 8 9 10 11

how do we merge these?

Theorem

The DC3 algorithm constructs a suffix array in $O(n)$ time.

Proof

Suppose $T(n)$ is the running time. We have

$T(n) = T(2n/3) + O(n)$

Solving this recurrence gives $T(n) \in O(n)$.

radix sorting and merging