Last Time

• Strings
  – They’re important
  – We need to sort them (and search them)

• MSD String Sort

• Tries
Trie – Visual

- Characters are implicitly defined by link index
- Each node has an array of links and a value
Trie – Analysis

• **Theorem:** The number of links in a trie is between $RN$ and $RNw$, where $w$ is the average key length.

• **Proof:** Immediate from the code.
  – Every key in the trie has a node containing $R$ links, so the number of links is at least $RN$.
  – If the first characters of all of the keys are different, then there is a node with $R$ links for every key character, so the number of links is $R$ times the total number of characters, or $RNw$.

• In other words, lots of wasted space.
de la Briandais Trie

• Old version: Node = value + $R$ links
• New version: Node = value + 2 links
  – Link 1: reference to child node (move to next level)
  – Link 2: reference to sibling node (stay on same level)

• Notice that now we need to store the key as a character, rather than use them as indices.
de la Briandais Trie

bye, by, get, got, gets
de la Briandais Trie - Analysis

• In the worst case, with an $R$-character alphabet and a string of length $w$, our search time is $Rw$, because we search through all $R$ characters on level $i$ before moving to level $i + 1$.

• In the average case, our search time is $\sim \log_R(w)$, since we won’t search through all of the characters before moving levels, and based on the level-sparsity rationale from the regular trie analysis.
  – For small $N$, search time is just $\sim w$. 
Substring Search

• Problem overview:
  – Given a pattern and a text sequence, find the pattern in the text.
  – Return the location where the pattern starts, or length of the text if not found
Brute Force Approach

```java
int search(String pat, String txt) {
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N-M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (txt.charAt(i+j) != pat.charAt(j)) {  // if
                break;
            }  // if
        }  // for
        if (j == M) {
            return i;  // found
        }  // if
    }  // for
    return N;  // not found
}  // search
```
Brute Force – Visual

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<th>txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>R</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
<td>B</td>
</tr>
</tbody>
</table>
```

entries in red are mismatches

```
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<th>txt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>A</td>
</tr>
</tbody>
</table>
```

entries in red are matches

return i when j is
Brute Force – Analysis

• **Theorem:** Brute force substring search requires $\sim NM$ character compares to search for a pattern of length $M$ in a text of length $N$, in the worst case.

• **Proof:** The worst case occurs when the pattern is checked for all characters and fails at the final character for all starting positions except the last one.

  – There are $N - M + 1$ possible match positions.
  – Each match position checks $M$ characters.
  – Running time is $(M) \times (N - M + 1)$, which reduces to $\sim NM$ if $M$ is very small compared to $N$. 

03/10/2015
Knuth-Morris-Pratt Substring Search

• Idea: Whenever we detect a mismatch, we already know some of the characters in the text. We can take advantage of this information to avoid backing up the text pointer over all of those known characters.
Knuth-Morris-Pratt Substring Search

• Rules:
  – Never back up the text pointer \( i \)
  – Use array \( dfa[][] \) to record how far to back up the pattern pointer \( j \) when a mismatch is detected
    • For every character \( c \), \( dfa[c][j] \) is the pattern position to compare against the next text position after comparing \( c \) with \( \text{pat.charAt}(j) \).
    • During the search, \( dfa[\text{txt.charAt}(i)][j] \) is the pattern position to compare with \( \text{txt.charAt}(i+1) \) after we compare \( \text{txt.charAt}(i) \) with \( \text{pat.charAt}(j) \).
Knuth-Morris-Pratt Substring Search

```java
int search(String txt) {
    int i, j;
    int N = txt.length();
    int M = pat.length();
    for (i=0, j=0; i<N && j<M; i++) {
        j = dfa[txt.charAt(i)][j];
    } //for
    if (j == M) {
        return i-M;
    } else {
        return N;
    } //if-else
} //search
```
KMP Substring Search – Visual

match:
set j to dfa[txt.charAt(i)][j]
= dfa[pat.charAt(j)][j]
= j+1

mismatch:
set j to dfa[txt.charAt(i)][j]
implies pattern shift to align
pat.charAt(j) with
txt.charAt(i+1)
KMP Substring Search – DFA

internal representation

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat.charAt(j)</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>dfa[][][j]</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

mismatch transition (back up)
mismatch transition (increment)
halt state
When we have a mismatch at `pat.charAt(j)`, we want to know what state the DFA would be if we were to back up the text index and rescan.

- We don’t actually do the backup, we just want to restart the DFA as if we had done the backup.
- The characters that would need to be rescanned are `pat.charAt(1)` through `pat.charAt(j-1)`.
- Since we already know these characters, we can figure out the state ahead of time!
KMP Substring Search – DFA Construction

dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++) {
    for (int c = 0; c < R; c++) {
        dfa[c][j] = dfa[c][X];
    } //for
    dfa[pat.charAt(j)][j] = j+1;
    X = dfa[pat.charAt(j)][X];
} //for
KMP – DFA Construction – Visual

\[ j \quad 0 \quad 1 \quad 2 \]
\[ \text{pat.\ characterAt}(j) \]
\[ A \quad B \quad A \]
\[ \text{dfa}[][j] \]
\[ A \quad 1 \quad 1 \quad 3 \]
\[ B \quad 0 \quad 2 \quad 0 \]
\[ C \quad 0 \quad 0 \quad 0 \]

\[ \text{copy dfa}[][X] \text{ to dfa}[][j] \]
\[ \text{dfa}[	ext{pat.\ characterAt}(j)][j] = j+1; \]
\[ X = \text{dfa}[	ext{pat.\ characterAt}(j)][X]; \]
KMP – DFA Construction – Visual
KMP – Analysis

• **Theorem:** Knuth-Morris-Pratt substring search accesses no more than $N + M$ characters for search for a pattern of length $M$ in a text of length $N$.

• **Proof:** Immediate from the code
  - We access each pattern character once when computing $\text{dfa}[][]$.
  - We access each text character once (in the worst case) in `search()`.
Any Questions?