Last Time

• Substring search
  – Given a pattern and a text sequence, find the pattern in the text
• Brute Force substring search
• KMP (Knuth-Morris-Pratt) substring search
• Boyer-Moore substring search
• Rabin-Karp substring search
Data Compression Problem Overview

• Compression:
  – **Input 1:** A sequence of bits
  – **Output 1:** A smaller sequence of bits
  – Transforms a bitstream $B$ into a compressed version $C(B)$

• Expansion
  – **Input 2:** A sequence of bits
  – **Output 2:** An expanded sequence of bits
  – Transforms the compressed $C(B)$ back into the original $B$
Data Compression Challenges

• Compression must be reversible (lossless compression), or at least a very close approximation (lossy compression)
• Compression must be computationally efficient (time is obvious, what about space?)
• Compression must work for all inputs, not just a subset (working at the bit level)
Data Compression Challenges

• Compression must work for all inputs, not just a subset. (really?)

• **Theorem:** No algorithm can compress every bitstream.

• **Proof:** Suppose that you have an algorithm that can compress every bitstream. Then, you could use the algorithm to compress its output to get a still shorter bitstream, and continue until you have a bitstream of length 0. This conclusion is absurd, therefore no algorithm can compress every bitstream. □

• Compression [algorithms] must [run on] all inputs, not just a subset.
Huffman Compression

- **Idea:** Use a small number of bits for frequent characters, and a large number of bits for rare characters.

- **ABRACADABRA!**
  - In ASCII (7 bits per char):
    1000001100001010100101000011100001110000011000011100000110000101001010000101000011000100100000110000101010010100001010001 (84 bits)
Huffman Compression

• ABRACADABRA!
  – *Variable-Length* \( (A=0, \ B=1, \ R=00, \ C=01, \ D=10, \ !=11) \):
    01000010100100011 (17 bits)
    • Problem – Not reversible, could also be CRRDDCRCB.
  – *Prefix-Free Variable-Length* – No character code is the prefix of another.
    – \( (A=0, \ B=1111, \ C=110, \ D=100, \ R=1110, \ !=101) \):
      011111110011001000111111100101 (30 bits)

• Now we just need to figure out how to generate prefix-free variable-length character codes...
Huffman Compression – Tries

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1111</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>1110</td>
</tr>
</tbody>
</table>

**Codeword Table**

**Trie Representation**

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>101</td>
</tr>
<tr>
<td>A</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>100</td>
</tr>
<tr>
<td>R</td>
<td>011</td>
</tr>
</tbody>
</table>

**Compressed Bitstring**

- **Left side**: `0111111100110010001111111100101` → 30 bits
- **Right side**: `1100011111010111100111100111101` → 29 bits
Huffman Compression – Steps

• Compression:
  1. Build an encoding trie (hardest step)
  2. Write the trie for use in expansion
  3. Use the trie to encode the bitstream

• Expansion:
  1. Read the trie
  2. Use the trie to decode the bitstream
Huffman Compression – Construct Trie

class Node implements Comparable<Node> {
    char ch;
    int freq;
    final Node left, right;

    Node(char c, int f, Node l, Node r) {
        ch = c;  freq = f;
        left = l;  right = r;
    } //Node (constructor)

    boolean isLeaf() {
        return (left == null) && (right == null);
    } //isLeaf

    int compareTo(Node that) {
        return this.freq - that.freq;
    } //compareTo
}
}  //Node (class)
Huffman Compression – Construct Trie

Node buildTrie(int[] freq) {
    MinPQ<Node> pq = min MinPQ<Node>();
    for (char c = 0; c < R; c++) {
        if (freq[c] > 0) {
            pq.insert(new Node(c, freq[c], null, null));
        } //if
    } //for
    while (pq.size() > 1) {
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node('\0', x.freq + y.freq, x, y);
        pq.insert(parent);
    } //while
    return pq.delMin();
} //buildTrie
Huffman Compression – Construct Trie

two tries with smallest weights

new parent for those two tries
Huffman Compression – Construct Trie

trie representation

Huffman code for the character stream “it was the best of times it was the worst of times LF”

labels on path from root are 11010 so 11010 is code for m

3 occurrences of w in input

codeword table

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF</td>
<td>101010</td>
</tr>
<tr>
<td>SP</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>11011</td>
</tr>
<tr>
<td>b</td>
<td>101011</td>
</tr>
<tr>
<td>e</td>
<td>000</td>
</tr>
<tr>
<td>f</td>
<td>11000</td>
</tr>
<tr>
<td>h</td>
<td>11001</td>
</tr>
<tr>
<td>i</td>
<td>1011</td>
</tr>
<tr>
<td>m</td>
<td>11010</td>
</tr>
<tr>
<td>o</td>
<td>0011</td>
</tr>
<tr>
<td>r</td>
<td>10100</td>
</tr>
<tr>
<td>s</td>
<td>100</td>
</tr>
<tr>
<td>t</td>
<td>111</td>
</tr>
<tr>
<td>w</td>
<td>0010</td>
</tr>
</tbody>
</table>
Huffman Compression – Writing Trie

```java
void writeTrie(Node x) {
    if (x.isLeaf()) {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```
Huffman Compression – Writing Trie
Huffman Compression – Compression

```java
String[] buildCode(Node root) {
    String[] st = new String[R];
    buildCode(st, root, ""');
    return st;
} //buildCode

void buildCode(String[] st, Node x, String s) {
    if (x.isLeaf()) {
        st[x.ch] = s;
        return;
    } //if
    buildCode(st, x.left, s+'0');
    buildCode(st, x.right, s+'1');
} //buildCode (recursive)
```
Huffman Compression – Compression

```java
void compress(char[] input) {
    for (int i = 0; i < input.length; i++) {
        String code = st[input[i]];
        for (int j = 0; j < code.length; j++) {
            if (code.charAt(j) == '1') {
                BinaryStdOut.write(true);
            } else {
                BinaryStdOut.write(false);
            }
        }
    }
}
```
Huffman Compression – Analysis

• **Theorem:** For any prefix-free code, the length of the encoded bitstring is equal to the weighted external path length of the corresponding trie.

• **Proof:** From the construction of the trie:
  – The depth of each leaf is the number of bits used to encode the character in the leaf.
  – Thus, the weighted external path length is the length of the encoded bitstring – the sum over all letters of the number of occurrences times the number of bits per occurrence.
Any Questions?