Module 10: Compression

T. Biedl  M. Petrick  O. Veksler
Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2019

References: Goodrich & Tamassia 10.3
Outline

1. Compression
   - Encoding Basics
   - Huffman Codes
   - Run-Length Encoding
   - Lempel-Ziv-Welch
   - bzip2
   - Burrows-Wheeler Transform
Outline

Compression
  - Encoding Basics
    - Huffman Codes
    - Run-Length Encoding
    - Lempel-Ziv-Welch
    - bzip2
    - Burrows-Wheeler Transform
Data Storage and Transmission

**The problem**: How to store and transmit data?

Source text  The original data, string \( S \) of characters from the *source alphabet* \( \Sigma_S \)

Coded text  The encoded data, string \( C \) of characters from the *coded alphabet* \( \Sigma_C \)

Encoding  An algorithm mapping source texts to coded texts

Decoding  An algorithm mapping coded texts back to their original source text

**Note**: Source “text” can be any sort of data (not always text!)

Usually the coded alphabet \( \Sigma_C \) is just binary: \( \{0,1\} \).
Judging Encoding Schemes

We can always measure efficiency of encoding/decoding algorithms.

What other goals might there be?

- Processing speed
- Reliability (e.g. error-correcting codes)
- Security (e.g. encryption)
- **Size**

Encoding schemes that try to minimize the size of the coded text perform *data compression*. We will measure the *compression ratio*:

\[
\frac{|C| \cdot \log |\Sigma_C|}{|S| \cdot \log |\Sigma_S|}
\]
Types of Data Compression

Logical vs. Physical

- **Logical Compression** uses the meaning of the data and only applies to a certain domain (e.g. sound recordings)
- **Physical Compression** only knows the physical bits in the data, not the meaning behind them

Lossy vs. Lossless

- **Lossy Compression** achieves better compression ratios, but the decoding is approximate; the exact source text $S$ is not recoverable
- **Lossless Compression** always decodes $S$ exactly

For media files, lossy, logical compression is useful (e.g. JPEG, MPEG)

We will concentrate on *physical, lossless* compression algorithms. These techniques can safely be used for any application.
Character Encodings

**Definition:** Map each character in $\Sigma_S$ to a string in coded alphabet.

$$E : \Sigma_S \rightarrow \Sigma_C^*$$

- For $c \in \Sigma_S$, we call $E(c)$ the *codeword* of $c$
- **Fixed-length code:** All codewords have the same length.
- **ASCII** (American Standard Code for Information Interchange), 1963:

<table>
<thead>
<tr>
<th>char</th>
<th>null</th>
<th>start of heading</th>
<th>start of text</th>
<th>end of text</th>
<th>...</th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>A</th>
<th>B</th>
<th>...</th>
<th>∼</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>48</td>
<td>49</td>
<td>...</td>
<td>65</td>
<td>66</td>
<td>...</td>
<td>126</td>
<td>127</td>
</tr>
</tbody>
</table>

- 7 bits to encode 128 possible characters:
  - “control codes”, spaces, letters, digits, punctuation
- Not well-suited for non-English text:
  - ISO-8859 extends to 8 bits, handles most Western languages
- To decode ASCII, we look up each 7-bit pattern in a table.

- **Other (earlier) fixed-length codes:** Baudot code, Murray code
Variable-Length Codes

**Definition:** Different codewords have different lengths

**Example 1:** Morse code.

![Morse Code Diagram](http://apfelmus.nfshost.com/articles/fun-with-morse-code.html)


**Example 2:** UTF-8 encoding of Unicode:

- Encodes any Unicode character (more than 107,000 characters) using 1-4 bytes
Encoding

Assume we have some character encoding \( E : \Sigma_S \rightarrow \Sigma_C^* \).

- Note that \( E \) is a dictionary with keys in \( \Sigma_S \).
- Typically \( E \) would be stored as array indexed by \( \Sigma_S \).

\[
\text{Encoding}(E, S[0..n-1])
\]

\( E \): the encoding dictionary, \( S \): text with characters in \( \Sigma_S \)

1. initialize empty string \( C \)
2. for \( i = 0 \ldots n - 1 \)
3. \( x \leftarrow E.\text{search}(S[i]) \)
4. \( C.\text{append}(x) \)
5. return \( C \)

Example: encode text “WATT” with Morse code:

\[
\begin{align*}
\cdot & \cdot \cdot \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
W & A & T & T
\end{align*}
\]
Decoding

The **decoding algorithm** must map $\Sigma_C^*$ to $\Sigma_S^*$.

- The code must be *uniquely decodable*.
  - This is false for Morse code as described!
    - $\bullet \ldots \bullet \ldots \bullet \ldots \bullet$ decodes to WATT and ANO and WJ.
      (Morse code uses ‘end of character’ pause to avoid ambiguity.)

- From now on only consider *prefix-free* codes $E$: $E(c)$ is not a prefix of $E(c')$ for any $c, c' \in \Sigma_S$.

- This corresponds to a **trie** with characters of $\Sigma_S$ only at the leaves.

- The codewords need no end-of-string symbol $\$\$ if $E$ is prefix-free.
Decoding of Prefix-Free Codes

Any prefix-free code is uniquely decodable (why?)

\[
PrefixFreeDecoding(T, C[0..n-1])
\]

\(T\): the trie of a prefix-free code, \(C\): text with characters in \(\Sigma_C\)

1. initialize empty string \(S\)
2. \(i \leftarrow 0\)
3. \textbf{while} \(i < n\)
4. \(r \leftarrow T.root\)
5. \textbf{while} \(r\) is not a leaf
6. \(\text{if } i = n\) return “invalid encoding”
7. \(c \leftarrow \text{child of } r\) that is labelled with \(C[i]\)
8. \(i \leftarrow i + 1\)
9. \(r \leftarrow c\)
10. \(S.append(\text{character stored at } r)\)
11. return \(S\)

Run-time: \(O(|C|)\).
**Encoding from the Trie**

We can also encode directly from the trie.

```
PrefixFreeEncodingFromTrie(T, S[0..n − 1])
T : the trie of a prefix-free code, S: text with characters in Σ_S
1.   L ← array of nodes in T indexed by Σ_S
2.   for all leaves ℓ in T
3.     L[character at ℓ] ← ℓ
4.   initialize empty string C
5.   for i = 0 to n − 1
6.     w ← empty string; v ← L[S[i]]
7.     while v is not the root
8.        w.prepend(character from v to its parent)
9.     // Now w is the encoding of S[i].
10.    C.append(w)
11.   return C
```

Run-time: \(O(\mid T\mid + \mid C\mid) = O(\mid \Sigma_S\mid + \mid C\mid).\)
Example: Prefix-free Encoding/Decoding

Code as table:

<table>
<thead>
<tr>
<th>( c \in \Sigma_S )</th>
<th>( \sqcup )</th>
<th>A</th>
<th>E</th>
<th>N</th>
<th>O</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E(c) )</td>
<td>000</td>
<td>01</td>
<td>101</td>
<td>001</td>
<td>100</td>
<td>11</td>
</tr>
</tbody>
</table>

Code as trie:

- Encode \( AN\sqcup ANT \rightarrow 010010000100111 \)
- Decode \( 111000001010111 \rightarrow TO\sqcup EAT \)
Outline

1. Compression
   - Encoding Basics
   - Huffman Codes
   - Run-Length Encoding
   - Lempel-Ziv-Welch
   - bzip2
   - Burrows-Wheeler Transform
Character Frequency

**Overall goal:** Find an encoding that is short.

**Observation:** Some letters in $\Sigma$ occur more often than others. So let’s use shorter codes for more frequent characters.

For example, the frequency of letters in typical English text is:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>12.70%</td>
</tr>
<tr>
<td>t</td>
<td>9.06%</td>
</tr>
<tr>
<td>a</td>
<td>8.17%</td>
</tr>
<tr>
<td>o</td>
<td>7.51%</td>
</tr>
<tr>
<td>i</td>
<td>6.97%</td>
</tr>
<tr>
<td>n</td>
<td>6.75%</td>
</tr>
<tr>
<td>s</td>
<td>6.33%</td>
</tr>
<tr>
<td>h</td>
<td>6.09%</td>
</tr>
<tr>
<td>r</td>
<td>5.99%</td>
</tr>
<tr>
<td>d</td>
<td>4.25%</td>
</tr>
<tr>
<td>l</td>
<td>4.03%</td>
</tr>
<tr>
<td>c</td>
<td>2.78%</td>
</tr>
<tr>
<td>u</td>
<td>2.76%</td>
</tr>
<tr>
<td>m</td>
<td>2.41%</td>
</tr>
<tr>
<td>w</td>
<td>2.36%</td>
</tr>
<tr>
<td>f</td>
<td>2.23%</td>
</tr>
<tr>
<td>g</td>
<td>2.02%</td>
</tr>
<tr>
<td>p</td>
<td>1.93%</td>
</tr>
<tr>
<td>b</td>
<td>1.49%</td>
</tr>
<tr>
<td>v</td>
<td>0.98%</td>
</tr>
<tr>
<td>k</td>
<td>0.77%</td>
</tr>
<tr>
<td>j</td>
<td>0.15%</td>
</tr>
<tr>
<td>x</td>
<td>0.15%</td>
</tr>
<tr>
<td>q</td>
<td>0.10%</td>
</tr>
<tr>
<td>z</td>
<td>0.07%</td>
</tr>
</tbody>
</table>
Huffman’s Algorithm: Building the best trie

For a given source text $S$, how to determine the “best” trie that minimizes the length of $C$?

1. Determine frequency of each character $c \in \Sigma$ in $S$
2. For each $c \in \Sigma$, create “$\overline{c}$” (height-0 trie holding $c$).
3. Assign a “weight” to each trie: sum of frequencies of all letters in trie. Initially, these are just the character frequencies.
4. Find the two tries with the minimum weight.
5. Merge these tries with new interior node; new weight is the sum. (Corresponds to adding one bit to the encoding of each character.)
6. Repeat Steps 4–5 until there is only 1 trie left; this is $T$.

What data structure should we store the tries in to make this efficient?
Example: Huffman tree construction

Example text: LOSSLESS, \( \Sigma S = \{L, O, S, E\} \)

Character frequencies: \( E:1, \quad L:2, \quad O:1, \quad S:4 \)

\[
\begin{align*}
\text{LOSSLESS} & \rightarrow 01001110100011 \\
\text{Compression ratio:} & \quad \frac{14}{8 \cdot \log 4} = \frac{14}{16} \approx 88\%
\end{align*}
\]
Huffman’s Algorithm: Pseudocode

\begin{algorithm}
\caption{Huffman-Encoding($S[0..n-1]$)}
\textbf{S}: text over some alphabet $\Sigma_S$
\begin{algorithmic}[1]
\State $f \leftarrow$ array indexed by $\Sigma_S$, initially all-0 \hspace{1cm} // frequencies
\For{$i = 0$ to $n - 1$}
\State increase $f[S[i]]$ by 1
\EndFor
\State $Q \leftarrow$ min-oriented priority queue that stores tries \hspace{1cm} // initialize PQ
\For{all $c \in \Sigma_S$ with $f[c] > 0$}
\State $Q$.\textit{insert} \hspace{1cm} (single-node trie for $c$ with weight $f[c]$)
\EndFor
\While{$Q$.\textit{size()} $> 1$} \hspace{1cm} // build decoding trie
\State $T_1 \leftarrow Q$.\textit{deleteMin}(), $f_1 \leftarrow$ weight of $T_1$
\State $T_2 \leftarrow Q$.\textit{deleteMin}(), $f_2 \leftarrow$ weight of $T_2$
\State $Q$.\textit{insert} \hspace{1cm} (trie with $T_1$, $T_2$ as subtries and weight $f_1 + f_2$)
\EndWhile
\State $T \leftarrow Q$.\textit{deleteMin}$\$
\State $C \leftarrow \text{PrefixFreeEncodingFromTrie}(T, S)$
\State \textbf{return} $C$ and $T$
\end{algorithmic}
\end{algorithm}
Huffman Coding Evaluation

- Note: constructed trie is **not unique** (why?)
  So decoding trie must be transmitted along with the coded text \( C \).
- This may make encoding bigger than source text!
- Encoding must pass through text twice (to compute frequencies and to encode)

- Encoding run-time: \( O(|\Sigma_S| \log |\Sigma_S| + |C|) \)
- Decoding run-time: \( O(|C|) \) (this is an *asymmetric* scheme).

- The constructed trie is *optimal* in the sense that \( C \) is shortest
  (among all prefix-free character-encodings with \( \Sigma_C = \{0, 1\} \)).
  We will not go through the proof.
- Many variations (give tie-breaking rules, estimate frequencies, adaptively change encoding, ....)
Outline

1. Compression
   - Encoding Basics
   - Huffman Codes
   - Run-Length Encoding
   - Lempel-Ziv-Welch
   - bzip2
   - Burrows-Wheeler Transform
Run-Length Encoding

- Variable-length code
- Example of **multi-character encoding**: multiple source-text characters receive one code-word.
- The source alphabet and coded alphabet are both binary: \{0, 1\}.
- Decoding dictionary is uniquely defined and not explicitly stored.

**Useful if**: \(S\) has long runs: 00000 111 0000

**Encoding idea**:
- Give the first bit of \(S\) (either 0 or 1)
- Then give a sequence of integers indicating run lengths.
- We don’t have to give the bit for runs since they alternate.

Example becomes: 0, 5, 3, 4

**Question**: How to encode a run length \(k\) in binary?
Prefix-free Encoding for Positive Integers

Use *Elias gamma code* to encode $k$:

- $\lfloor \log k \rfloor$ copies of 0, followed by
- binary representation of $k$ (always starts with 1)

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\lfloor \log k \rfloor$</th>
<th>$k$ in binary</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>100</td>
<td>00100</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>101</td>
<td>00101</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>110</td>
<td>00110</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
RLE Encoding

**RLE-Encoding**($S[0...n-1]$)

$S$: bitstring

1. initialize output string $C \leftarrow S[0]$
2. $i \leftarrow 0$  // index of parsing $S$
3. while $i < n$ do
4.   $k \leftarrow 1$  // length of run
5.   while ($i + k < n$ and $S[i + k] = S[i]$) do $k++$
6.   $i \leftarrow i + k$
7. // compute and append Elias gamma code
8. $K \leftarrow$ empty string
9. while $k > 1$
10.   $C.append(0)$
11.   $K.prepend(k \ mod \ 2)$
12.   $k \leftarrow \lfloor k/2 \rfloor$
13.   $K.prepend(1)$  // $K$ is binary encoding of $k$
14.   $C.append(K)$
15. return $C$

Biedl, Petrick, Veksler (SCS, UW)
RLE Decoding

RLE-Decoding(C)
C: stream of bits
1. initialize output string S
2. $b \leftarrow C.pop()$ // bit-value for the current run
3. while C has bits left
4. $\ell \leftarrow 0$ // length of base-2 number $-1$
5. while $C.pop() = 0$ do $\ell++$
6. $k \leftarrow 1$ // base-2 number converted
7. for $(j \leftarrow 1$ to $\ell)$ do $k \leftarrow k \times 2 + C.pop()$
8. // if C runs out of bits then encoding was invalid
9. for $(j \leftarrow 1$ to $k)$ do $S.append(b)$
10. $b \leftarrow 1 - b$
11. return $S$
RLE Example

Encoding:
\[ S = 11111110010000000000000000000011111111111 \]

Decoding:
\[ C = 00001101001001010 \]
\[ S = 000000000000001111011 \]
RLE Properties

- An all-0 string of length $n$ would be compressed to $2\lfloor \log n \rfloor + 2 \in o(n)$ bits.
- Usually, we are not that lucky:
  - No compression until run-length $k \geq 6$
  - Expansion when run-length $k = 2$ or $4$
- Method can be adapted to larger alphabet sizes
- Used in some image formats (e.g. TIFF)
Outline

1 Compression
   • Encoding Basics
   • Huffman Codes
   • Run-Length Encoding
   • Lempel-Ziv-Welch
   • bzip2
   • Burrows-Wheeler Transform
Longer Patterns in Input

Huffman and RLE mostly take advantage of frequent or repeated *single characters*.

**Observation**: Certain *substrings* are much more frequent than others.

Examples:

- **English text:**
  - Most frequent digraphs: TH, ER, ON, AN, RE, HE, IN, ED, ND, HA
  - Most frequent trigraphs: THE, AND, THA, ENT, ION, TIO, FOR, NDE
- **HTML**: “<a href”, “<img src”, “<br>”
- **Video**: repeated background between frames, shifted sub-image

**Ingredient 1** for Lempel-Ziv-Welch compression: take advantage of such substrings *without* needing to know beforehand what they are.
Adaptive Dictionaries

ASCII, UTF-8, and RLE use fixed dictionaries.

In Huffman, the dictionary is not fixed, but it is static: the dictionary is the same for the entire encoding/decoding.

**Ingredient 2** for LZW: *adaptive encoding*:

- There is a fixed initial dictionary $D_0$. (Usually ASCII.)
- For $i \geq 0$, $D_i$ is used to determine the $i$th output character
- After writing the $i$th character to output, both encoder and decoder update $D_i$ to $D_{i+1}$

Encoder and decoder must both know how the dictionary changes.
Lempel-Ziv

Lempel-Ziv is a family of *adaptive* compression algorithms.

**Main Idea:** Each character in the coded text $C$ either refers to a single character in $\Sigma_S$, or a *substring* of $S$ that both encoder and decoder have already seen.

**Variants:**

- **LZ77** Original version ("sliding window")
  - Derivatives: LZSS, LZFG, LZRW, LZP, DEFLATE, 
  - DEFLATE used in (pk)zip, gzip, PNG

- **LZ78** Second (slightly improved) version
  - Derivatives: LZW, LZMW, LZAP, LZY, 
  - LZW used in compress, GIF

**Patent issues!**
LZW Overview

- Start with dictionary \( D_0 \) for \( |\Sigma_S| \).
  Usually \( \Sigma_S = \text{ASCII} \), then this uses codenumbers 0, \ldots, 127.
- Every step adds to dictionary a multi-character string, using codenumbers 128, 129, \ldots.
- Encoding:
  - Store current dictionary \( D_i \) as a trie.
  - Parse trie to find longest prefix \( x \) already in \( D_i \).
    So all of \( x \) can be encoded with one number.
  - Add to dictionary the prefix that would have been useful:
    add \( xK \) where \( K \) is the character that follows \( x \) in \( S \).
    This creates one child in trie at the leaf where we stopped.
- Output is a list of numbers. This is usually converted to bit-string with fixed-width encoding using 12 bits.
  - This limits the codenumbers to 4096.
LZW Example

Text: ANANA SANN A

Dictionary:

Final output: 00001000001 00001001110 00001000000 00001000001 00001010011 00001000000 00001000001

Biedl, Petrick, Veksler (SCS, UW)
LZW encoding pseudocode

\[
\text{LZW-encode}(S)
\]
\[
S : \text{stream of characters}
\]
1. Initialize dictionary \( D \) with ASCII in a trie
2. \( \text{idx} \leftarrow 128 \)
3. \( \textbf{while} \) there is input in \( S \) \( \textbf{do} \)
4. \( v \leftarrow \text{root of trie } D \)
5. \( K \leftarrow S.\text{peek()} \)
6. \( \textbf{while} \ (v \text{ has a child } c \text{ labelled } K) \)
7. \( v \leftarrow c; S.\text{pop()} \)
8. \( \textbf{if} \) there is no more input in \( S \) \( \textbf{break} \) \ (goto 10)
9. \( K \leftarrow S.\text{peek()} \)
10. \( \textbf{output} \ \text{codenumber stored at } v \)
11. \( \textbf{if} \) there is more input in \( S \)
12. \( \text{create child of } v \text{ labelled } K \text{ with codenumber } \text{idx} \)
13. \( \text{idx}++ \)
LZW decoding

- Same idea: build dictionary while reading string.
- Dictionary maps numbers to strings.
  To save space, store string as code of prefix + one character.
- Example: 67 65 78 32 66 129 133

<table>
<thead>
<tr>
<th>Code #</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32</td>
<td>( \square )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
</tbody>
</table>

\[ D = \]

<table>
<thead>
<tr>
<th>input</th>
<th>decodes to</th>
<th>Code #</th>
<th>String (human)</th>
<th>String (computer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>c</td>
<td>128</td>
<td>CA</td>
<td>67, A</td>
</tr>
<tr>
<td>65</td>
<td>a</td>
<td>129</td>
<td>AN</td>
<td>65, N</td>
</tr>
<tr>
<td>78</td>
<td>n</td>
<td>130</td>
<td>N( \square )</td>
<td>78, ( \square )</td>
</tr>
<tr>
<td>32</td>
<td>( \square )</td>
<td>131</td>
<td>( \square )B</td>
<td>32, B</td>
</tr>
<tr>
<td>129</td>
<td>AN</td>
<td>132</td>
<td>BA</td>
<td>66, A</td>
</tr>
<tr>
<td>133</td>
<td>??</td>
<td>133</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LZW decoding: the catch

- In this example: Want to decode 133, but not yet in dictionary!
- What happened during the corresponding encoding?

Text: CA N □ B A N x₁ x₂ ... 

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>65</td>
<td>78</td>
<td>33</td>
<td>66</td>
<td>129</td>
<td>A</td>
<td>133</td>
</tr>
</tbody>
</table>

Dictionary

(parts omitted):

65 — N — 129 — x₁ — 133

66 — A — 132

- We know: 133 encodes ANx₁ (for unknown x₁)
- We know: Next step uses 133 = ANx₁
- So x₁ = A and 133 encodes ANA

Generally: If code number is not yet in D, then it encodes

“previous string + first character of previous string”
LZW decoding pseudocode

\[
\text{LZW-decode}(C)
\]

C: stream of integers

1. \( D \leftarrow \text{dictionary that maps } \{0, \ldots, 127\} \text{ to ASCII} \)
2. \( idx \leftarrow 128 \)
3. \( S \leftarrow \text{empty string} \)
4. \( code \leftarrow \text{first code from } C \)
5. \( s \leftarrow D(code); S.append(s) \)
6. \( \text{while there are more codes in } C \text{ do} \)
7. \( s_{\text{prev}} \leftarrow s \)
8. \( code \leftarrow \text{next code of } C \)
9. \( \text{if } code \neq idx \)
10. \( s \leftarrow D(code) \)
11. \( \text{else} \quad // \text{special situation!} \)
12. \( s \leftarrow s_{\text{prev}} + s_{\text{prev}}[0] \)
13. \( S.append(s) \)
14. \( D.insert(idx, s_{\text{prev}} + s[0]) \)
15. \( idx++ \)
16. \( \text{return } S \)
LZW decoding example revisited

- Example: 67 65 78 32 66 129 133 83

\[
D =
\]

<table>
<thead>
<tr>
<th>Code #</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>32</td>
<td>□</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>decodes to</th>
<th>Code #</th>
<th>String (human)</th>
<th>String (computer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>67</td>
<td>C</td>
<td>128</td>
<td>CA</td>
<td>67, A</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
<td>129</td>
<td>AN</td>
<td>65, N</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
<td>130</td>
<td>N□</td>
<td>78, □</td>
</tr>
<tr>
<td>32</td>
<td>□</td>
<td>131</td>
<td>□B</td>
<td>32, B</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>132</td>
<td>BA</td>
<td>66, A</td>
</tr>
<tr>
<td>129</td>
<td>AN</td>
<td>133</td>
<td>ANA</td>
<td>129, A</td>
</tr>
<tr>
<td>133</td>
<td>ANA</td>
<td>134</td>
<td>ANAS</td>
<td>133, S</td>
</tr>
</tbody>
</table>
## Compression summary

<table>
<thead>
<tr>
<th>Huffman</th>
<th>Run-length encoding</th>
<th>Lempel-Ziv-Welch</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-length</td>
<td>variable-length</td>
<td>fixed-length</td>
</tr>
<tr>
<td>single-character</td>
<td>multi-character</td>
<td>multi-character</td>
</tr>
<tr>
<td>2-pass</td>
<td>1-pass</td>
<td>1-pass</td>
</tr>
<tr>
<td>60% compression</td>
<td>bad on text</td>
<td>45% compression</td>
</tr>
<tr>
<td>on English text</td>
<td></td>
<td>on English text</td>
</tr>
<tr>
<td>optimal 01-prefix-code</td>
<td>good on long runs  (e.g., pictures)</td>
<td>good on English text</td>
</tr>
<tr>
<td>must send dictionary</td>
<td>can be worse than ASCII</td>
<td>can be worse than ASCII</td>
</tr>
<tr>
<td>rarely used directly</td>
<td>rarely used directly</td>
<td>frequently used</td>
</tr>
<tr>
<td>part of pkzip, JPEG, MP3</td>
<td>fax machines, old picture-formats</td>
<td>GIF, some variants of PDF, Unix compress</td>
</tr>
</tbody>
</table>
Outline

1 Compression
   • Encoding Basics
   • Huffman Codes
   • Run-Length Encoding
   • Lempel-Ziv-Welch
   • bzip2
   • Burrows-Wheeler Transform
bzip2 overview

To achieve even better compression, bzip2 uses *text transform*: Change input into a different text that is not necessarily shorter, but that has other desirable qualities.

- **text** $T_0$
  - Burrows-Wheeler transform
    - If $T_0$ has repeated longer substrings, then $T_1$ has long runs of characters.
- **text** $T_1$
  - Move-to-front transform
    - If $T_1$ has long runs of characters, then $T_2$ has long runs of zeros.
- **text** $T_2$
  - Modified RLE
    - If $T_2$ has long runs of zeroes, then $T_3$ has chars 0' and 1' very frequently
- **text** $T_3$
  - Huffman encoding
    - Compresses well since input-chars unevenly distributed
Move-to-Front transform

Recall the MTF heuristic for self-organizing search:

- Dictionary is stored as an unsorted array or linked list
- After an element is accessed, move it to the front of the dictionary

How can we use this idea for text transformations?

Take advantage of *locality* in the data.
If we see a character now, we’ll probably see it again soon.

**Specifics:** MTF is an *adaptive* text-transform algorithm.
If the source alphabet is $\Sigma_S$ with size $|\Sigma_S| = m$,
then the coded alphabet will be $\Sigma_C = \{0, 1, \ldots, m - 1\}$. 
Move-to-Front Encoding/Decoding

\[ MTF\text{-}encode(S) \]
1. \( L \leftarrow \text{array with } \Sigma_S \text{ in some pre-agreed, fixed order} \)
2. while \( S \) has more characters do
3. \( c \leftarrow \text{next character of } S \)
4. output index \( i \) such that \( L[i] = c \)
5. for \( j = i - 1 \) down to 0
6. swap \( L[j] \) and \( L[j + 1] \)

Decoding works in \textit{exactly} the same way:

\[ MTF\text{-}decode(C) \]
1. \( L \leftarrow \text{array with } \Sigma_S \text{ in some pre-agreed, fixed order} \)
2. while \( C \) has more characters do
3. \( i \leftarrow \text{next integer from } C \)
4. output \( L[i] \)
5. for \( j = i - 1 \) down to 0
6. swap \( L[j] \) and \( L[j + 1] \)
MTF Example

\[
S = \text{MISSISSIPPI}
\]

\[
C = \quad
\]

- What does a run in \(S\) encode to in \(C\)?
- What does a run in \(C\) mean about the source \(S\)?
Outline

Compression
- Encoding Basics
- Huffman Codes
- Run-Length Encoding
- Lempel-Ziv-Welch
- bzip2
- Burrows-Wheeler Transform
Burrows-Wheeler Transform

- Transforms source text to a coded text with the same letters, just in a different order.
- The coded text will be more easily compressible with MTF.
- Required: the source text $S$ ends with end-of-word character $\$\$ that occurs nowhere else in $S$.
- Encoding algorithm needs all of $S$ (no streaming possible). BWT is a block compression method.
- (As we will see) decoding is more efficient than encoding, so BWT is an asymmetric scheme.

Crucial ingredient: A cyclic shift of a string $X$ of length $n$ is the concatenation of $X[i+1..n-1]$ and $X[0..i]$, for $0 \leq i < n$. 
BWT Algorithm and Example

\[ S = \text{alf}_1\text{eats}_1\text{alfalfa}_1 \]

1. Write all cyclic shifts
2. Sort cyclic shifts
3. Extract last characters from sorted shifts

\[ C = \]

\[ $\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
\[ \text{alf}_1\text{eats}_1\text{alfalfa}_1\text{alf}_1\text{eats}_1\text{alfalfa}_1 \]
BWT Decoding

**Idea:** Given \( C \), we can reconstruct the *first* and *last column* of the array of cyclic shifts by sorting.

\[
C = \text{ard}\$rcaaaabb
\]

1. **Last column:** \( C \)
   
   \[
   \begin{array}{c}
   \$,3...........a,0 \\
   a,0...........r,1 \\
   a,6...........d,2 \\
   a,7...........$,3 \\
   a,8...........r,4 \\
   a,9...........c,5 \\
   b,10............a,6 \\
   b,11............a,7 \\
   c,5............a,8 \\
   d,2............a,9 \\
   r,1...........b,10 \\
   r,4...........b,11 \\
   \end{array}
   \]

2. **First column:** \( C \) sorted
   
   \[
   \begin{array}{c}
   \$,3...........a,0 \\
   a,0...........r,1 \\
   a,6...........d,2 \\
   a,7...........$,3 \\
   a,8...........r,4 \\
   a,9...........c,5 \\
   b,10............a,6 \\
   b,11............a,7 \\
   c,5............a,8 \\
   d,2............a,9 \\
   r,1...........b,10 \\
   r,4...........b,11 \\
   \end{array}
   \]

3. **Disambiguate by row-index**

4. **Starting from \$,** recover \( S \)

\[
S = \text{,3}$a,0 $r,1 d,2 $,3 $r,4 c,5 $a,6 a,7 a,8 a,9 a,10 a,11 r,1 b,10 r,4 b,11$
\]
BWT Decoding

\[ \text{BWT-decoding}(C[0..n-1]) \]

\( C : \text{string of characters over alphabet } \Sigma \)

1. \( A \leftarrow \text{array of size } n \)
2. \( \text{for } i = 0 \text{ to } n-1 \)
3. \( A[i] \leftarrow (C[i], i) \)
4. \( \text{Stably sort } A \text{ by first entry} \)
5. \( \text{for } j = 0 \text{ to } n \) // where is the $-char?
6. \( \text{if } C[j] = $ \break \)
7. \( S \leftarrow \text{empty string} \)
8. \( \text{repeat} \)
9. \( j \leftarrow \text{second entry of } A[j] \)
10. \( S.append(C[j]) \)
11. \( \text{until } C[j] = $ \)
12. \( \text{return } S \)
Encoding cost: $O(n^2)$ (using MSD radix sort) and often better

Encoding is theoretically possible in $O(n)$ time:
- Sorting cyclic shifts of $S$ is equivalent to sorting the suffixes of $S \cdot S$ that have length $> n$
- This can be done by traversing the suffix tree of $S \cdot S$

Decoding cost: $O(n)$ (faster than encoding)

Encoding and decoding both use $O(n)$ space.

Tends to be slower than other methods, but (combined with MTF, RLE and Huffman) gives better compression.