Outline

- 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* $Σ_S$

- **Coded text**
  - the encoded data, string $C$ of characters from the *coded alphabet* $Σ_C$

- **Encoding**
  - an algorithm mapping source texts to coded texts

- **Decoding**
  - an algorithm mapping coded texts back to their original source text

- **Notes**
  - source “text” can be any sort of data (not always text)
  - usually the coded alphabet is just binary $Σ_C = \{0, 1\}$
  - usually $S$ and $C$ are stored as *streams*
    - read/write only one character at a time
    - convenient for handling huge texts
      - can start processing text while it is still being loaded
  - input stream supports methods: `pop()`, `top()`, `isEmpty()`
  - output stream supports methods: `append()`, `isEmpty()`
Judging Encoding Schemes

- Can measure time/space efficiency of encoding/decoding algorithms, as for any usual algorithm

- What other goals make sense?
  - reliability
    - error-correcting codes
  - security
    - encryption
  - size (our main objective in this course)

- 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
    - only applies to a certain domain (e.g. sound recordings)
  - **Physical Compression**
    - only know physical bits in data, not their meaning

- **Lossy vs. Lossless**
  - **Lossy Compression**
    - achieves better compression ratios
    - decoding is approximate
    - exact source text $S$ is not recoverable
  - **Lossless Compression**
    - always decodes $S$ exactly

Lossy, logical compression is useful
- media files: JPEG, MPEG

But we will concentrate on *physical, lossless* compression
- can be safely used for any application
Character Encodings

- **Definition:** character encoding $E$ maps each character in the source alphabet to a string in coded alphabet $E : \Sigma_S \rightarrow \Sigma_C^*$
  - for $c \in \Sigma_S$, $E(c)$ is called the codeword (or code) of $c$
- Character encoding sometimes is called character-by-character encoding
  - encode one character at a time
- Two possibilities
  - Fixed-length code: all codewords have the same length
  - Variable-length code: codewords may have different lengths
Fixed Length Codes

- Example: ASCII (American Standard Code for Information Interchange), 1963

<table>
<thead>
<tr>
<th>char in $\Sigma_S$</th>
<th>null</th>
<th>start of heading</th>
<th>start 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>...</td>
<td>48</td>
<td>49</td>
<td>...</td>
<td>65</td>
<td>66</td>
<td>...</td>
<td>126</td>
<td>127</td>
</tr>
<tr>
<td>code as binary string</td>
<td>0000000</td>
<td>0000001</td>
<td>0000010</td>
<td>0110000</td>
<td>0110001</td>
<td>0100001</td>
<td>0100010</td>
<td>1111110</td>
<td>1111111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 7 bits to encode 128 possible characters
  - control codes, spaces, letters, digits, punctuation
    - A P P L E $\rightarrow$ (65, 80, 80, 76, 69) $\rightarrow$ 01000001 1010000 1010000 1001100 1000101
- Standard in all computers and often our source alphabet
- Not well-suited for non-English text
  - ISO-8859 extends to 8 bits, handles most Western languages
- Other (earlier) fixed-length codes: Baudot code, Murray code
- To decode a fixed-length code (say codewords have $k$ bits), we look up each $k$-bit pattern in a table
Variable-Length Codes

- **Overall goal:** Find an encoding that is short
- **Observation:** Some alphabet letters occur more often than others
  - Idea: use shorter codes for more frequent characters
  - Example: frequency of letters in typical English text

<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>y</td>
<td>1.97%</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>
Variable-Length Codes

- **Example 1: Morse code**

  ![Morse Code Diagram]

  - A dash is equal to three dots.
  - The space between parts of the same letter is equal to one dot.
  - The space between two letters is equal to three dots.

- **Example 2: UTF-8 encoding of Unicode**
  - there are more than 107,000 Unicode characters
  - uses 1-4 bytes to encode any Unicode character
Encoding

- Assume we have some character encoding $E : \Sigma_S \rightarrow \Sigma_C^*$
- $E$ is a dictionary with keys in $\Sigma_S$
- Typically $E$ would be stored as array indexed by $\Sigma_S$

```text
charByChar::Encoding($E, S, C$)
$E$: encoding dictionary, $S$: input stream with characters in $\Sigma_S$
$C$: output stream

while $S$ is non-empty

\[
x \leftarrow E.\text{search}(S.\text{pop}())
\]

$C.\text{append}(x)$
```

- Example: encode text “WATT” with Morse code

```
W  A  T  T
```

![Morse Code Chart]

![Character Encoding Dictionary]

Decoding

- The **decoding algorithm** must map $\Sigma_C^*$ to $\Sigma_S$
- The code must be *uniquely decodable*
  - false for Morse code as described
    - decodes to both WATT and EAJ
  - 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 $c, c' \in \Sigma_S$
- Store codes in a **trie** with characters of $\Sigma_S$ at the leaves

- Do not need symbol $\$, codewords are prefix-free by definition
Example: Prefix-free Encoding/Decoding

- Code as table

<table>
<thead>
<tr>
<th>$c \in \Sigma_s$</th>
<th>$\cup$</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 $\text{AN\cup\top\text{ANT}} \rightarrow 01\ 001\ 000\ 0100111$
- Decode $1110000001010111 \rightarrow \text{TOL\cup EAT}$
Decoding of Prefix-Free Codes

- Any prefix-free code is uniquely decodable

PrefixFree::decoding(T, C, S)

\( T \): trie of a prefix-free code,
\( C \): input-stream with characters in \( \Sigma_C \)
\( S \): output-stream

```cpp
while \( C \) is non-empty
    \( r \leftarrow T\).root
    while \( r \) is not a leaf
        if \( C \) is empty or has no child labelled \( C\).top()
            return “invalid encoding”
        \( r \leftarrow \) child of \( r \) that is labelled with \( C\).pop()
        \( S\).append(character stored at \( r \))
```

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

- Can encode directly from the trie $T$

### $\text{PrefixFree::encoding}(T, S, C)$

$T$ : prefix-free code trie, $S$: input-stream with characters in $\Sigma_S$

- $L \leftarrow$ array of nodes in $T$ indexed by $\Sigma_S$
- for all leaves $l$ in $T$
  - $L[\text{character at } l] \leftarrow l$

  while $S$ is non-empty
    - $w \leftarrow$ empty string; $v \leftarrow L[S.\text{pop()}]$  
      while $v$ is not the root
        - $w$.prepend (character from $v$ to its parent)
        - $v \leftarrow \text{parent}(v)$
      // now $w$ is the encoding of $S$
    - $C$.append($w$)

- Run-time: $O(|T| + |C|)$
  - $O(|\Sigma_S| + |C|)$ if $T$ has no nodes with one child
Outline

- Compression
  - Encoding Basics
  - Huffman Codes
  - Run-Length Encoding
  - Lempel-Ziv-Welch
  - bzip2
  - Burrows-Wheeler Transform
Huffman’s Algorithm: Building the Best Trie

- For a given $S$ the best trie can be constructed with Huffman tree algorithm
  - trie giving the shortest coded text $C$ if alphabet is binary
    - $\Sigma_C = \{0,1\}$
  - tailored to frequencies in that particular $S$
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, \( \Sigma_S = \{G, R, E, N, Y\} \)
- Calculate character frequencies
  \[
  G: 2, \quad R: 2, \quad E: 4, \quad N: 2, \quad Y: 1
  \]
- Put each character into its own (single node) trie
  - each trie has a frequency
  - initially, frequency is equal to its character frequency

\[
\begin{array}{ccccc}
2 & & 4 & & 2 & & 1 \\
G & & R & & E & & Y \\
\end{array}
\]
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
  \[G: 2, \ R: 2, \ E: 4, \ N: 2, \ Y: 1\]
- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, \( \Sigma_S = \{G, R, E, N, Y\} \)
- Calculate character frequencies
  \[ G: 2, \; R: 2, \; E: 4, \; N: 2, \; Y: 1 \]
- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

```
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
```

```
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
Example text: GREENENERGY, \( \Sigma_S = \{G, R, E, N, Y\} \)
```
Example: Huffman Tree Construction

- Example text: *GREENENERGY*, \( \Sigma_s = \{G, R, E, N, Y\} 
- Calculate character frequencies
  \[ G: 2, \ R: 2, \ E: 4, \ N: 2, \ Y: 1 \]
- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
  
  \begin{align*}
  G &: 2, \quad R &: 2, \quad E &: 4, \quad N &: 2, \quad Y &: 1
  \end{align*}

- Join two least frequent tries into a new trie
  
  - frequency of the new trie = sum of old trie frequencies
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, \( \Sigma_S = \{G, R, E, N, Y\} \)
- Calculate character frequencies
  
  \[
  G: 2, \quad R: 2, \quad E: 4, \quad N: 2, \quad Y: 1
  \]
- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies

![Huffman Tree Diagram]
Example: Huffman Tree Construction

- Example text: \textit{GREENENERGY}, \( \Sigma_S = \{G, R, E, N, Y\} \)
- Calculate character frequencies
  \( G: 2, \ R: 2, \ E: 4, \ N: 2, \ Y: 1 \)
- Join two least frequent tries into a new trie
  \( \text{frequency of the new trie} = \text{sum of old trie frequencies} \)
Example: Huffman Tree Construction

- Example text: $GREENENERGY$, $\Sigma_S = \{G, R, E, N, Y\}$
- Calculate character frequencies
  
  \[
  G: 2, \quad R: 2, \quad E: 4, \quad N: 2, \quad Y: 1
  \]

- Join two least frequent tries into a new trie
  - frequency of the new trie = sum of old trie frequencies
Example: Huffman Tree Construction

- Example text: $\text{GREENENERGY}$, $\Sigma_s = \{G, R, E, N, Y\}$
- Calculate character frequencies
  
  $G: 2$, $R: 2$, $E: 4$, $N: 2$, $Y: 1$

- Final Huffman tree

```
G 0 1
  0 1
  |   |   E R N
G 0 1
```

- $\text{GREENENERGY} \rightarrow 000 \ 10 \ 01 \ 01 \ 11 \ 01 \ 11 \ 01 \ 10 \ 000 \ 001$

- Compression ratio

\[
\frac{25}{11 \log 5} \approx 98\%
\]

- These frequencies are not skewed enough to lead to good compression
Huffman Algorithm Summary

- For a given source $S$, to determine a trie that minimizes length of $C$
  1) determine frequency of each character $c \in \Sigma$ in $S$
  2) for each $c \in \Sigma$, create trie of height 0 holding only $c$
     - call it $c$-trie
  3) assign a weight to each trie
     - sum of frequencies of all letters in a trie
     - initially, these are just the character frequencies
  4) find the two tries with the minimum weight
  5) merge these tries with a new interior node
     - the new weight is the sum of merged tries weights
     - added one bit to the encoding of each character
  6) repeat Steps 4–5 until there is only 1 trie left
     - this is $D$, the final decoder

- Data structure for making this efficient?
  - min-ordered heap
    - step 4 is two *delete-min*
    - step 5 is *insert*
Heap Storing Tries during Huffman Tree Construction

- Efficient data structure to store tries
  - a min-ordered heap
  - \((\text{key}, \text{value}) = (\text{trie weight}, \text{link to trie})\)
  - step 4 is two \textit{delete-mins}, step 5 is \textit{insert}
Huffman’s Algorithm Pseudocode

Huffman::encoding($S, C$)

$S$: input-stream with characters in $\Sigma_S$, $S$: output-stream

1. $f \leftarrow$ array indexed by $\Sigma_S$, initialized to 0
2. \textbf{while} $S$ is non-empty \textbf{do} increase $f[S.\text{pop}()]$ by 1 // get frequencies
3. $Q \leftarrow$ min-oriented priority queue to store tries
4. \textbf{for} all $c \in \Sigma_S$ with $f[c] > 0$
   - $Q.\text{insert}$(single-node trie for $c$ with weight $f[c]$)
5. \textbf{while} $Q.\text{size}() > 1$
   - $T_1 \leftarrow Q.\text{deleteMin}()$, $f_1 \leftarrow$ weight of $T_1$
   - $T_2 \leftarrow Q.\text{deleteMin}()$, $f_2 \leftarrow$ weight of $T_2$
   - $Q.\text{insert}$(trie with $T_1$, $T_2$ as subtries and weight $f_1 + f_2$)
6. $T \leftarrow Q.\text{deleteMin}()$ // trie for decoding
7. reset input-stream $S$
8. $C \leftarrow \text{PrefixFreeEncodingFromTrie}(T, S)$ // perform actual encoding

- Total time is $O(|\Sigma_S| \log |\Sigma_S| + |C|)$
Huffman Coding Evaluation

- 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 stream twice
  - to compute frequencies and to encode
  - cannot use stream unless it can be reset
- Time to compute trie $T$ and encode $S$
  $$O(|\Sigma_s| \log |\Sigma_s| + |C|)$$
- Decoding run-time
  $$O(|C|)$$
- The constructed trie is **optimal** in the sense that
  - $C$ is shortest among all prefix-free character encodings with $\Sigma_c = \{0, 1\}$
  - proof omitted
- Many variations
  - give tie-breaking rules, estimate frequencies, adaptively change encoding, ...
Outline

- Compression
  - Encoding Basics
  - Huffman Codes
  - Run-Length Encoding
  - Lempel-Ziv-Welch
  - bzip2
  - Burrows-Wheeler Transform
Single-Character vs Multi-Character Encoding

- **Single character encoding**: each source-text character receive one codeword

  \[ S = \text{b a n a n a} \]

  \[
  \begin{array}{cccccc}
  01 & 1 & 11 & 1 & 11 & 1 \\
  \end{array}
  \]

- **Multi-character encoding**: multiple source-text characters can receive one codeword

  \[ S = \text{b a n a n a} \]

  \[
  \begin{array}{cccc}
  01 & 11 & 101 \\
  \end{array}
  \]
Run-Length Encoding

- RLE is an example of multi-character encoding
- Source alphabet and coded alphabet are both binary: \( \Sigma = \{0, 1\} \)
  - can be extended to non-binary alphabets
- Useful \( S \) has long runs of the same character: 00000 111 0000
- Dictionary is uniquely defined by *algorithm*
  - no need to store it explicitly

**Encoding idea**

- give the first bit of \( S \) (either 0 or 1)
- then give a sequence of integers indicating run lengths
- do not have to give the bit for runs since they alternate

**Example** 00000 111 0000
- becomes: 0 5 3 4

**Need to encode run length in binary, how?**

- cannot use variable length binary encoding 10111100
  - do not know how to parse in individual run lengths
- fixed length binary encoding (say 16 bits) wastes space, bad compression
  - 0000000000000101000000000000001100000000000000100
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>
</tbody>
</table>

- Easy to decode
  - (number of zeros + 1) tells you the length of $k$ (in binary representation)
  - after zeros, read binary representation of $k$ (it starts with 1)
RLE Example: Encoding

- Encoding
  \[ S = 1111111001000000000000000000011111111111 \]

- \( C = 1 \)
Encoding

\[ S = 111111100100000000000000000000000111111111111 \]

\[ k = 7 \]

\[ C = 100111 \]
RLE Example: Encoding

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\log k$</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>7</td>
<td>2</td>
<td>111</td>
<td>00111</td>
</tr>
</tbody>
</table>

- **Encoding**
  
  \[ S = 111111100100000000000000000000001111111111111 \]
  
  \[ k = 2 \]
  
  \[ C = 100111010 \]
RLE Example: Encoding

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\log k$</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>7</td>
<td>2</td>
<td>111</td>
<td>00111</td>
</tr>
</tbody>
</table>

- **Encoding**
  
  $S = 1111111001000000000000000000000111111111111$
  
  $k = 1$
  
  $C = 10011101011$
## RLE Example: Encoding

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\log k$</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>7</td>
<td>2</td>
<td>111</td>
<td>00111</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>10100</td>
<td>000010100</td>
</tr>
</tbody>
</table>

- **Encoding**

\[
S = 111111110010000000000000000000000111111111111
\]

\[
k = 20
\]

\[
c = 1001110101 \textbf{000010100}
\]
RLE Example: Encoding

- Encoding

\[ S = \overbrace{1111111100100000000000000000000000}^{11111111111111} \]

\[ k = 11 \]

\[ C = 1001110101000010100 \text{ 0001011} \]

- Compression ratio

\[ 26/41 \approx 63\% \]
RLE Encoding

\texttt{RLE::encoding}(S, C)

\(S\): input-stream of bits, \(C\): output-stream

\(b \leftarrow S.\text{top}()\)

\(C.\text{append}(b)\)

\textbf{while} \(S\) is non-empty \textbf{do}

\(k \leftarrow 1\) // initialize run length

\textbf{while} (\(S\) is non-empty and \(S.\text{top}() = b\)) //compute run length

\(k++; S.\text{pop}()\)

// compute Elias gamma code \(K\) (binary string) for \(k\)

\(K \leftarrow \text{empty string}\)

\textbf{while} (\(k > 1\))

\(C.\text{append}(0)\) // 0 appended to output \(C\) directly

\(K.\text{prepend}(k \mod 2)\) // \(K\) is built from last digit forwards

\(k \leftarrow \lfloor k/2 \rfloor\)

\(K.\text{prepend}(1)\) // the very first digit of \(K\) was not computed

\(C.\text{append}(K)\)

\(b \leftarrow 1 - b\)
RLE Example: Decoding

- Recall that (# zeros + 1) tells you the length of $k$ in binary representation

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\log k$</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>13</td>
<td>3</td>
<td>1101</td>
<td>0001101</td>
</tr>
</tbody>
</table>

- Decoding

$C = \textcolor{red}{00\textcolor{green}{0011011001001010}}$

$\begin{align*}
    b & = 0 \\
    l & = 4 \\
    k & = 13 \\
    S & = \textcolor{red}{0000000000000000}
\end{align*}$
RLE Example: Decoding

Decoding

\[ C = 00001101001001010 \]
\[ b = 1 \]
\[ l = 3 \]
\[ k = 4 \]

\[ S = 000000000000001111 \]
RLE Example: Decoding

- Decoding
  
  \[ C = 00001101001001010 \]
  
  \[ b = 0 \]
  
  \[ l = 1 \]
  
  \[ k = 1 \]
  
  \[ S = 00000000000011110 \]
RLE Example: Decoding

- Decoding

\[ C = 00001101001001010 \]

\[ b = 1 \]

\[ l = 2 \]

\[ k = 2 \]

\[ S = 0000000000000001111011 \]
RLE Decoding

\[ \text{RLE-Decoding}(C) \]

\( C \): input stream of bits, \( S \): output-stream

\( b \leftarrow C.pop() \) // bit-value for the first run

\textbf{while} \( C \) is non-empty

\( l \leftarrow 0 \) // length of base-2 number - 1

\textbf{while} \( C.pop() = 0 \)

\( l++ \)

\( k \leftarrow 1 \) // base-2 number converted

\textbf{for} \( (j = 1 \text{ to } l) \) // translate \( k \) from binary string to integer

\( k \leftarrow k \times 2 + C.pop() \)

// if \( C \) runs out of bits then encoding was invalid

\textbf{for} \( (j = 1 \text{ to } k) \)

\( S.append(b) \)

\( b \leftarrow 1 - b \) // alternate bit-value
RLE Properties

- Variable length encoding
- Dictionary is uniquely defined by an algorithm
  - no need to explicitly store or send dictionary
- Best compression is for $S = 000 \ldots 000$ of length $n$
  - compressed to $2[\log n] + 2 \in o(n)$ bits
    - 1 for the initial bit
    - $[\log n]$ zeros to encode the length of binary representation of integer $n$
    - $[\log n] + 1$ digits that represent $n$ itself in binary
- Usually 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
  - but then the encoding for each run must also store the character
- Method can be adapted to encode only runs of 0
  - we will need this soon
- Used in some image formats (e.g. TIFF)
Outline

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

- Huffman and RLE 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
  - Encode characters and *frequent substrings*
    - no need to know which substrings are frequent
      - will discover frequent substring as we process text
      - will encode them as we read the text
    - dictionary constructed during encoding/decoding, no need to send it with encoding
      - how?
Adaptive Dictionaries

- ASCII, UTF-8, and RLE use *fixed* dictionaries
  - same dictionary for any text encoded
  - no need to pass dictionary to the decoder
- In Huffman, the dictionary is not fixed, but it is *static*
  - each text has its own dictionary
  - the dictionary *does not change* for the entire encoding/decoding
  - need to pass dictionary to the decoder
- **Ingredient 2** for LZW: *adaptive dictionary*
  - start with some initial dictionary $D_0$
    - usually ASCII
  - at iteration $i \geq 0$, $D_i$ is used to determine the $i$th output
  - after iteration $i$, update $D_i$ to $D_{i+1}$
    - a new character combination is inserted
  - encoder and decoder must both know how dictionary changes
    - compute dictionary during encoding/decoding
LZW Overview

- Start with dictionary $D_0$ for $\Sigma_S$
  - usually $\Sigma_S = ASCII$
  - codes from 0 to 127
  - every step adds to dictionary multi-character string, using codenumbers 128, 129, ...
- Iteration $i$ of encoding, current dictionary $D_i$
  
  $S = abbcbbad$
  
  $D_i = \{a:65, \text{ab:140, bb:145, bbc:146}\}$

  find longest substring that starts at current pointer and already in dictionary
  encode ‘bb’
  
  $D_{i+1} = D_i . \text{insert('bba', next_available_code)}$

  (logic: ‘bba’ would have been useful at iteration $i$, so it may be useful in the future)

- Store current dictionary $D_i$ in a trie
- Output is a list of numbers (codewords)
  - each number is usually converted to bit-string with fixed-width encoding using 12 bits
    - this limits code numbers to 4096
Tries for LZW

- Key-value pairs are (string, code)
- Trie stores KVP at all nodes (external and *internal*) except the root
  - works because a string is inserted only after all its prefixes are inserted
- We show code (value) at each node, because the key can be read off from the edges
LZW Example

- **Start dictionary** $D$
  - ASCII characters
  - codes from 0 to 127
  - next inserted code will be 128
  - variable idx keeps track of next available code
  - initialize $idx = 128$

- **Text**
  - A N A N A N A N A N N N A
LZW Example

- Dictionary $D$
  - $idx = 129$

- Text
  - A N A N A N A N A N A

- Encoding
  - 65

- Add to dictionary “string just encoded” + “first character of next string to be encoded”
- Inserting new item is $O(1)$ since we stopped at the right node in the trie when we searched for ‘A’
LZW Example

- **Dictionary** $D$
  - $idx = 130$

- **Text**
  - A N A N A N A N A N A A

- **Encoding**
  - 65 78

*add to dictionary*
LZW Example

- Dictionary $D$
  - $idx = 131$

- Text
  - ANANAN

- Encoding
  - 65 78 128

- Dictionary
  - $D_{\text{id}} = 130$

- Add to dictionary
  - 83
LZW Example

- Dictionary D
  - $idx = 132$

- Text
  - A A N A N

- Encoding
  - 65 78 128 130

- Dictionary D
  - $idx = 132$

- Text
  - A A N A N

- Encoding
  - 65 78 128 130
LZW Example

- Dictionary D
  - \( \text{idx} = 133 \)

- Text
  - A N A N A N A N A

- Encoding
  - 65 78 128 130 78

- Add to dictionary: N N A
LZW Example

- **Dictionary D**
  - \( idx = 133 \)

- **Text**
  - A N A N A N A A N N N A

- **Encoding**
  - 65 78 128 130 78 129
LZW Example

- Dictionary D
  - $idx = 133$

- Text
  - A N A N A N

- Encoding
  - 65 78 128 130 78 129

- Final output
  - 0000100001 0000100110 00001000000 0000100010 0000100110 00001000001

- Use fixed length (12 bits) per code
  - 12 bit binary string representation for each code
  - total of $2^{12} = 4096$ codes available during encoding
LZW encoding pseudocode

$LZW\text{-encode}(S)$

$S$ : input stream of characters, $C$ : output-stream
initialize dictionary $D$ with ASCII in a trie

$idx \leftarrow 128$

while there is input in $S$ do
    $v \leftarrow$ root of trie $D$
    while $S$ is non-empty and $v$ has a child $c$ labelled $S$. $\text{top}()$
        $v \leftarrow c$
        $S.\text{pop}()$
        $C.\text{append}(\text{codenumber stored at } v)$
    if $S$ is non-empty
        create child of $v$ labelled $S.\text{top}()$ with code $idx$
        $idx \leftarrow$ $idx + 1$
**LZW encoding pseudocode**

**LZW-encode**($S$)

$S$ : input stream of characters, $C$ : output-stream

initialize dictionary $D$ with ASCII in a trie

$idx \leftarrow 128$

while there is input in $S$ do

$\forall \leftarrow$ root of trie $D$

while $S$ is non-empty and $\forall$ has a child $c$ labelled $S$. $\text{top}()$

$\forall \leftarrow c$

$S$. $\text{pop}()$

$C$. $\text{append}$(codenumber stored at $\forall$)

if $S$ is non-empty

create child of $\forall$ labelled $S$. $\text{top}()$ with code $idx$

$idx \leftarrow idx + 1$

- Running time is $O(|S|)$
LZW Encoder vs Decoder

- For decoding, need a dictionary
- Construct a dictionary during decoding, but one step behind
  - at iteration $i$ of decoding we can reconstruct the substring which encoder inserted into dictionary at iteration $i - 1$
    - delay is due to not having access to the original text
LZW Decoding Example

- Given encoding to decode back to the source text

<table>
<thead>
<tr>
<th>65</th>
<th>78</th>
<th>128</th>
<th>130</th>
<th>78</th>
<th>129</th>
</tr>
</thead>
</table>

- Build dictionary adaptively, while decoding
- Decoding starts with the same initial dictionary as encoding
  - use array instead of trie, need $D$ that allows efficient search by code
- We will show the original text during decoding in this example, but just for reference
  - do not need original text to decode

Initial $D$:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
</tbody>
</table>

$idx = 128$
LZW Decoding Example

- **Text**
  - A N A N A N A N N A

- **Encoding**
  - 65 78 128 130 78 129

- **Decoding**
  - Iter $i = 0$

  - $D =
    | 65 | A |
    | 78 | N |
    | 83 | S |

  - $idx = 128$

- First step: $s = D(65) = ‘A’$
- During encoding, added new string ‘AN’ to the dictionary at iteration $i = 0$
  - looked ahead at the text and saw ‘N’
- During decoding, when read 65, cannot look ahead in the text
  - no new word added at iteration $i = 0$
  - but keep track of $s_{prev} = \text{string decoded at previous iteration}$
  - it is also the string encoder encoded at previous iteration
LZW Decoding Example

- **Text**
  - A
  - N

- **Encoding**
  - 65
  - 78

- **Decoding**
  - Iteration $i = 1$

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>128</td>
<td>AN</td>
</tr>
</tbody>
</table>

- $D = \begin{array}{c|c}
  65 & A \\
  78 & N \\
  83 & S \\
  128 & AN \\
\end{array}$

- $idx = 129$

- $s_{prev} = \text{‘A’}$
- First step: $s = D(78) = \text{‘N’}$
- Now know that at iteration $i = 0$ of encoding, next character we peaked at was ‘N’
- So can add string ‘A’ + ‘N’ = ‘AN’ to the dictionary

\[ s_{prev} \quad s[0] \]

- $s$ is string decoded at current iteration

- Starting at iteration $i = 1$ of decoding
  - Add $s_{prev} + s[0]$ to dictionary
LZW Decoding Example Continued

- **Text**: 
  A N A N A N N A A N

- **Encoding**: 
  65 78 128 130 78 129

- **Decoding**
  iter $i = 2$

\[
\begin{array}{c|c}
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA \\
\end{array}
\]

\[
D = \begin{array}{c|c}
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA \\
\end{array}
\]

- $s_{prev} = 'N'$
- First step: $s = D(128) = 'AN'$
- Next step: add to dictionary $s_{prev} + s[0]$
  
  'N' + 'A' = 'NA'

\[idx = 130\]
LZW Decoding Example

- **Text**
  - A N A N A N A N A

- **Encoding**
  - 65 78 128 130 78 129

- **Decoding**
  - A N AN
  - \( i = 3 \)

\[
D = \begin{bmatrix}
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA
\end{bmatrix}
\]

- \( s_{prev} = \text{‘AN’} \)
- First step: \( s = D(130) = ??? \)
  - problem: code 130 is not in \( D \)
- Dictionary is exactly one step behind at decoding
- Current decoder iteration is \( i = 3 \)
- Encoder added \((s, 130)\) to \( D \) at iteration \( i = 2 \)
- Encoder adds \( s_{prev} + s[0] \)
  - \( s_{prev} = \text{‘AN’} \)
  - \( s[0] = s_{prev}[0] = \text{‘A’} \)
  - \( s = \text{‘ANA’} \)
## LZW Decoding Example

- **Text**: `ANAANANAANNNNA`
- **Encoding**: `65 78 128 130 78 129`
- **Decoding**: `AN ANA`

### Iteration i = 3

<table>
<thead>
<tr>
<th>Code</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>128</td>
<td>AN</td>
</tr>
<tr>
<td>129</td>
<td>NA</td>
</tr>
<tr>
<td>130</td>
<td>ANA</td>
</tr>
</tbody>
</table>

### Dictionary (`D`)

<table>
<thead>
<tr>
<th>Code</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>128</td>
<td>AN</td>
</tr>
<tr>
<td>129</td>
<td>NA</td>
</tr>
<tr>
<td>130</td>
<td>ANA</td>
</tr>
</tbody>
</table>

- **General rule**: if code `C` is not in `D`
  - `s = s_{prev} + s_{prev}[0]`
- **In our example**
  - `AN + A = ANA`
- **Continue the example**
- **Add to dictionary** `s_{prev} + s[0]`

**idx = 131**
LZW Decoding Example

- **Text**: A N A N A N A N N A
- **Encoding**: 65 78 128 130 78 129
- **Decoding**: A N AN AN

**iter \( i = 4 \)**

\[
D = \begin{array}{c|c}
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA \\
130 & ANA \\
131 & ANAN \\
\end{array}
\]

- \( s_{\text{prev}} = 'ANA' \)
- If code \( C \) is not in \( D \) 
  \[
  s = s_{\text{prev}} + s_{\text{prev}}[0]
  \]
- Add to dictionary \( s_{\text{prev}} + s[0] \)

\( idx = 132 \)
LZW Decoding Example

- **Text**: A N A N A N A N N A
- **Encoding**: 65 78 128 130 78 129
- **Decoding**: iter \(i = 5\)

\[
D = \begin{array}{c|c}
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA \\
130 & ANA \\
131 & ANAN \\
\end{array}
\]

- \(s_{prev} = 'N'\)
- If code \(C\) is not in \(D\)
  \[
s = s_{prev} + s_{prev}[0]
\]
- Add to dictionary \(s_{prev} + s[0]\)

\(idx = 132\)
LZW Decoding Pseudocode

\[ LZW::decoding(C, S) \]

\( C \) : input-stream of integers, \( S \) : output-stream

\[ D \leftarrow \text{dictionary that maps } \{0, \ldots, 127\} \text{ to ASCII} \]

\[ idx \leftarrow 128 \quad \text{// next available code} \]

\[ code \leftarrow C.pop() \]

\[ s \leftarrow D(code) \]

\[ S.append(s) \]

\textbf{while} there are more codes in \( C \) \textbf{do}

\[ s_{prev} \leftarrow s \]

\[ code \leftarrow C.pop() \]

\textbf{if} code < idx

\[ s \leftarrow D(code) \quad \text{//code in } D, \text{ look up string } s \]

\textbf{if} code = idx \quad \text{// code not in } D \text{ yet, reconstruct string}

\[ s \leftarrow s_{prev} + s_{prev}[0] \]

\textbf{else} Fail

\[ S.append(s) \]

\[ D.insert(idx, s_{prev} + s[0]) \]

\[ idx ++ \]

- Running time is \( O(|S|) \)
LZW decoding

- To save space, store new codes using its prefix code + one character
  - for each codeword, can find corresponding string $s$ in $O(|s|)$ time

| $D = \begin{array}{|c|c|} \hline
65 & A \\
78 & N \\
83 & S \\
128 & AN \\
129 & NA \\
130 & ANA \\
131 & ANAN \\
\hline \end{array}$ |
| $\begin{array}{|c|c|} \hline
65 & A \\
78 & N \\
83 & S \\
128 & 65, N \\
129 & 78, A \\
130 & 128, A \\
131 & 130, N \\
\hline \end{array}$ |

- means ‘AN’
- means ‘NA’
- means ‘ANA’
- means ‘ANAA’

wasteful storage
Lempel-Ziv Family

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

- Encoding is $O(|S|)$ time, uses a trie of encoded substrings to store the dictionary.
- Decoding is $O(|S|)$ time, uses an array indexed by code numbers to store the dictionary.
- Encoding and decoding need to go through the string only one time and do not need to see the whole string.
  - can do compression while streaming the text.
- Works badly if no repeated substrings.
  - dictionary gets bigger, but no new useful substrings inserted.
- In practice, compression rate is around 45% on English text.
Outline

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

- **Text transform** changes input text into a *different text*
  - not necessarily shorter
  - but has properties likely to lead to better compression at a later stage
- To achieve better compression, bzip2 uses the following text transforms

  \[ T_0 = \text{alf e at s a l f a} \]

  **Burrows-Wheeler transform**

  if \( T_0 \) has repeated longer substrings, then \( T_1 \) has long runs of characters

  \[ T_1 = \text{aff s$ e f l l l a a a a t a} \]

  **Move-to-front transform**

  if \( T_1 \) has long runs of characters, then \( T_2 \) has long runs of zeros and skewed frequencies

  **Modified RLE**

  if \( T_2 \) has long runs of zeroes, then \( T_3 \) is shorter and skewed frequencies remain

  **Huffman encoding**

  compresses well since frequencies are skewed

  text \( T_2 = 13053435006006 \)

  text \( T_3 \)

  text \( T_4 \)
Move-to-Front transform

- Recall the MTF heuristic
  - after an element is accessed, move it to array front

```
A   B   C   D   E
```

search D

```
D   A   B   C   E
```

- Use this idea for MTF (move to front) text transformation
MTF Encoding Example

- Source alphabet $\Sigma_s$ with size $|\Sigma_s| = m$
- Store alphabet in an array

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

- This gives us encoding dictionary $D$
  - single character encoding $E$
  - Code of any character = index of array where character stored in dictionary $D$
    - $E('B') = 1$
    - $E('H') = 7$
- Coded alphabet is $\Sigma_c = \{0, 1 \ldots, m - 1\}$
- Change dictionary $D$ dynamically (like LZW)
  - unlike LZW
    - no new items added to dictionary
    - codeword for one or more letters can change at each iteration
MTF Encoding Example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

\[ S = \text{MISSISSIPPI} \]

\[ C = \]
MTF Encoding Example

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[
S = \text{MISSISSIPPI}
\]

\[
C = 12
\]
MTF Encoding Example

\[ S = \text{MISSISSIPPI} \]

\[ C = 12 \ 9 \]
MTF Encoding Example

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[S = \text{MISSISSIPPI}\]

\[C = 12 \ 9 \ 18\]
MTF Encoding Example

$S = \text{MISSISSIPPI}$

$C = 12 \ 9 \ 18 \ 0$
MTF Encoding Example

S = MISSISSIPPI

C = 12 9 18 0 1
MTF Encoding Example

<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>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>S</td>
<td>M</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

S = **MISSISSIPPI**

C = 12 9 18 0 1 1
MTF Encoding Example

\[
\begin{array}{cccccccccccccccccc}
\end{array}
\]

\[S = \text{MISSISSIPPI}\]

\[C = 12 \ 9 \ 18 \ 0 \ 1 \ 1 \ 0\]
MTF Encoding Example

\[ S = \text{MISSISSIPPI} \]
\[ C = 12 \ 9 \ 18 \ 0 \ 1 \ 1 \ 0 \ 1 \ 16 \ 0 \ 1 \]

- What does a run in \( C \) mean about the source \( S \)?
  - zeros tell us about consecutive character runs
MTF Decoding Example

\[
S = \begin{array}{cccccccccccccc}
C &=& 12 & 9 & 18 & 0 & 1 & 1 & 0 & 1 & 16 & 0 & 1
\end{array}
\]

- Decoding is similar
- Start with the same dictionary \( D \) as encoding
- Apply the same MTF transformation at each iteration
  - dictionary \( D \) undergoes exactly the transformations when decoding
  - no delays, identical dictionary at encoding and decoding iteration \( i \)
  - can always decode original letter
MTF Decoding Example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

\[ S = \text{M} \]

\[ C = 12 \ 9 \ 18 \ 0 \ 1 \ 1 \ 0 \ 1 \ 16 \ 0 \ 1 \]
### MTF Decoding Example

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| M | A | B | C | D | E | F | G | H | I | J | K | L | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

\[
S = M | \\
C = 12\ 9\ 18\ 0\ 1\ 1\ 0\ 1\ 16\ 0\ 1
\]
MTF Decoding Example

S = M I S
C = 12 9 18 0 1 1 0 1 16 0 1
Move-to-Front Transform: Properties

- If a character in \( S \) repeats \( k \) times, then \( C \) has a run of \( k - 1 \) zeros
- \( C \) contains a lot of small numbers and a few big ones
- \( C \) has the same length as \( S \), but better properties for encoding
Move-to-Front Encoding/Decoding Pseudocode

**MTF::encoding**($S, C$)

$L \leftarrow$ array with $\sum S$ in some pre-agreed, fixed order (i.e. ASCII)

while $S$ is non-empty do

$c \leftarrow S.pop()$

$i \leftarrow$ index such that $L[i] = c$

for $j = i - 1$ down to 0

swap $L[j]$ and $L[j + 1]$

**MTF::decoding**($C, S$)

$L \leftarrow$ array with $\sum S$ in some pre-agreed, fixed order (i.e. ASCII)

while $C$ is non-empty do

$i \leftarrow$ next integer of $C$

$S.append(L[i])$

for $j = i - 1$ down to 0

swap $L[j]$ and $L[j + 1]$
Move-to-Front Transform Summary

- **MTF text transform**
  - source alphabet is $\Sigma_S$ with size $|\Sigma_S| = m$
  - store alphabet in an array
    - code of any character = index of array where character stored
    - coded alphabet is $\Sigma_C = \{0, 1, \ldots, m - 1\}$
  - Dictionary is adaptive
    - nothing new is added, but meaning of codewords are changed
  - MTF is an *adaptive* text-transform algorithm
    - it does not compress input
    - the output has the same length as input
    - but output has better properties for compression
Outline

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

- Transformation (not compression) algorithm
  - transforms source text to a coded text with the same letters but in different order
    - source and coded alphabets are the same
    - if original text had frequently occurring substrings, then transformed text should have many runs of the same character
  - more suitable for MTF transformation

\[ S = \text{alf eats alf alfa} \rightarrow \text{Burrows-Wheeler Transform} \rightarrow C = \text{aff s$efllllaaata} \]

- Required: the source text \( S \) ends with \textit{end-of-word character $}\)
  - $ occurs nowhere else in \( S \)
- Based on cyclic shifts for a string
  - a cyclic shift of string \( X \) of length \( n \) is the concatenation of
    \[ X[i + 1…n - 1] \text{ and } X[0…i], \text{ for } 0 \leq i < n \]
  - example: string abcde a cyclic shift cdeab
BWT Algorithm and Example

\[ S = \text{alfalfa} \]

- Write all consecutive cyclic shifts
  - forms *an array of shifts*
  - last letter in any row is the first letter of the previous row

\[ \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \quad \text{alfalfa} \]
BWT Algorithm and Example

\[ S = \text{alfeatsalfalfa}$ \]

- Array of cyclic shifts
  - the first column is the original \( S \)
BWT Algorithm and Example

\[ S = \text{alf}feats\text{al}f\text{alf}\text{a} \]

- Array of cyclic shifts
- \( S \) has ‘alf’ repeated 3 times
  - 3 different shifts start with ‘lf’ and end with ‘a’
BWT Algorithm and Example

\[ S = alf\text{a}sale\text{f}\text{a}salf\text{a}\$ \]

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
  - strict sorting order due to ‘$’
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
  - sort groups ‘If’ lines together, and they all end with ‘a’
BWT Algorithm and Example

\[ S = \text{alfalfa alfalfa alfalfa} \]

- Array of cyclic shifts
- Sort (lexographically) cyclic shifts
  - strict sorting order due to ‘$’
- First column (of course) has many consecutive character runs
- But also the last column has many consecutive character runs
  - sort groups ‘lf’ lines together, and they all end with ‘a’
  - could happen that another pattern will interfere
    - ‘hlfd’ broken into ‘h’ and ‘lfd’
  - the longer is repeated pattern, the less chance of interference

sorted shifts array

\[
\begin{align*}
\text{alfeatsalfalfa} \newline
\text{alfeatsalfalfa} a \newline
\text{alfeatsalfalfa} \newline
\text{alfeatsalfalfa$alf} \newline
\text{alfeatsalfalfa$af} \newline
\text{alfeatsalfalf} \newline
\text{alfeatsalfala} \newline
\text{alfeatsalfal} \newline
\text{alfeatsalf} \newline
\text{alfeatsal} \newline
\text{alfeats} \newline
\text{alfeats} \newline
\text{alfeats} \newline
\text{alfeats} \newline
\end{align*}
\]
BWT Algorithm and Example

$S = \text{alfeatsalfalfa}$

- Sorted array of cyclic shifts
- First column is useless for encoding
  - cannot decode it
- Last column can be decoded
- BWT Encoding
  - last characters from sorted shifts
    - i.e. the last column

$C = \text{affs$efllllaaata}$

sorted shifts array

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{atsalfalfa}$

$\text{eatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfalfalfa}$

$\text{alfalfalfa}$

$\text{alfalfalfa}$

$\text{alfalfalfa}$

$\text{atsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$

$\text{alfeatsalfalfa}$
### BWT Fast Encoding: Efficient Sorting

$$S = \text{alfeatsalfalfa}$$

<table>
<thead>
<tr>
<th>$i$</th>
<th>cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\text{alfeatsalfalfa}$</td>
</tr>
<tr>
<td>1</td>
<td>$\text{lfeatsalfalfa}$a</td>
</tr>
<tr>
<td>2</td>
<td>$\text{featsalfalfa}$al</td>
</tr>
<tr>
<td>3</td>
<td>$\text{eatsalfalfa}$alf</td>
</tr>
<tr>
<td>4</td>
<td>$\text{atsalfalfa}$alfe</td>
</tr>
<tr>
<td>5</td>
<td>$\text{tsalfalfa}$alfe$a$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{salfalfa}$alfeat</td>
</tr>
<tr>
<td>7</td>
<td>$\text{alfalfa}$alfeats</td>
</tr>
<tr>
<td>8</td>
<td>$\text{lfalfa}$alfeatsa</td>
</tr>
<tr>
<td>9</td>
<td>$\text{falfa}$alfeatsal</td>
</tr>
<tr>
<td>10</td>
<td>$\text{alfa}$alfeatsalf</td>
</tr>
<tr>
<td>11</td>
<td>$\text{lfa}$alfeatsalfa</td>
</tr>
<tr>
<td>12</td>
<td>$\text{fa}$alfeatsalfal</td>
</tr>
<tr>
<td>13</td>
<td>$\text{a}$alfeatsalfalf</td>
</tr>
<tr>
<td>14</td>
<td>$\text{}$alfeatsalfalfa</td>
</tr>
</tbody>
</table>

- Can refer to a cyclic shift by the start index in the text, no need to write it out explicitly.
- For sorting, letters after ‘$’ do not matter.

$$\text{alfalfa}$alfeats  
$$\text{lfa}$alfeatsalfa$$
BWT Fast Encoding: Efficient Sorting

\[ S = \text{alfeatsalfalfa}\$ \]

<table>
<thead>
<tr>
<th>i</th>
<th>cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alfeatsalfalfa$</td>
</tr>
<tr>
<td>1</td>
<td>lfeatsalfalfa$a</td>
</tr>
<tr>
<td>2</td>
<td>featsalfalfa$al</td>
</tr>
<tr>
<td>3</td>
<td>eatsalfalfa$alf</td>
</tr>
<tr>
<td>4</td>
<td>atsalfalfa$alfe</td>
</tr>
<tr>
<td>5</td>
<td>tsalfalfa$alfeal</td>
</tr>
<tr>
<td>6</td>
<td>salfalfa$alfeat</td>
</tr>
<tr>
<td>7</td>
<td>alfalfa$alfeats</td>
</tr>
<tr>
<td>8</td>
<td>lalfa$alfeatsa</td>
</tr>
<tr>
<td>9</td>
<td>falfa$alfeatsal</td>
</tr>
<tr>
<td>10</td>
<td>alfa$alfeatsalf</td>
</tr>
<tr>
<td>11</td>
<td>lfa$alfeatsalfa</td>
</tr>
<tr>
<td>12</td>
<td>fa$alfeatsalfal</td>
</tr>
<tr>
<td>13</td>
<td>a$alfeatsalfalf</td>
</tr>
<tr>
<td>14</td>
<td>$alfeatsalfalfa</td>
</tr>
</tbody>
</table>

- Can refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after ‘$’ do not matter

\[ lfa$alfeatsalfa \\
\text{salfalfa}$alfeat \]
BWT Fast Encoding: Efficient Sorting

\[ S = \text{alfeatsalfalfa}$ \]

- Can refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after ‘$’ do not matter

<table>
<thead>
<tr>
<th>i</th>
<th>cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alfeatsalfalfa$</td>
</tr>
<tr>
<td>1</td>
<td>lfeatsalfalfa$a</td>
</tr>
<tr>
<td>2</td>
<td>featsalfalfa$al</td>
</tr>
<tr>
<td>3</td>
<td>eatsalfalfa$alf</td>
</tr>
<tr>
<td>4</td>
<td>atsalfalfa$alfe</td>
</tr>
<tr>
<td>5</td>
<td>tselfalfa$alfeaa</td>
</tr>
<tr>
<td>6</td>
<td>safalfa$alfeat</td>
</tr>
<tr>
<td>7</td>
<td>alfalfa$alfeats</td>
</tr>
<tr>
<td>8</td>
<td>lfa$alfeatsa</td>
</tr>
<tr>
<td>9</td>
<td>falfa$alfeatsal</td>
</tr>
<tr>
<td>10</td>
<td>alfa$alfeatsalf</td>
</tr>
<tr>
<td>11</td>
<td>lfa$alfeatsalfa</td>
</tr>
<tr>
<td>12</td>
<td>fa$alfeatsalfal</td>
</tr>
<tr>
<td>13</td>
<td>a$alfeatsalfal</td>
</tr>
<tr>
<td>14</td>
<td>$alfeatsalfalfa</td>
</tr>
</tbody>
</table>

\[ \text{lfa$alfeatsalfa} \]
\[ \text{lfa$alfeatsalfa} \]
\[ \text{lfa$alfeatsal}a \]
**BWT Fast Encoding: Efficient Sorting**

\[ S = \text{alfeatsalfalfa}\$ \]

<table>
<thead>
<tr>
<th>i</th>
<th>cyclic shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alfeatsalfalfa$</td>
</tr>
<tr>
<td>1</td>
<td>lfeatsalfalfa$a</td>
</tr>
<tr>
<td>2</td>
<td>featsalfalfa$al</td>
</tr>
<tr>
<td>3</td>
<td>eatsalfalfa$alf</td>
</tr>
<tr>
<td>4</td>
<td>atsalfalfa$alfe</td>
</tr>
<tr>
<td>5</td>
<td>tsalfalfa$alfea</td>
</tr>
<tr>
<td>6</td>
<td>salfalfa$alfeat</td>
</tr>
<tr>
<td>7</td>
<td>alfalfa$alfeats</td>
</tr>
<tr>
<td>8</td>
<td>lalfa$alfeatsa</td>
</tr>
<tr>
<td>9</td>
<td>falfa$alfeatsal</td>
</tr>
<tr>
<td>10</td>
<td>alfa$alfeatsalf</td>
</tr>
<tr>
<td>11</td>
<td>lfa$alfeatsalf</td>
</tr>
<tr>
<td>12</td>
<td>fa$alfeatsalfal</td>
</tr>
<tr>
<td>13</td>
<td>a$alfeatsalfalf</td>
</tr>
<tr>
<td>14</td>
<td>$\text{alfeatsalfalfa}$</td>
</tr>
</tbody>
</table>

- Can refer to a cyclic shift by the start index in the text, no need to write it out explicitly
- For sorting, letters after ‘\$’ do not matter
- This is the same as sorting suffixes of \( S \)
- We already know how to do it
  - exactly as for suffix arrays, with MSD-Radix-Sort
  - \( O(n \log n) \) running time
## BWT Fast Encoding: Efficient Sorting

\[ S = a l f e a t s a l f a l f a $ \]

<table>
<thead>
<tr>
<th>i</th>
<th>cyclic shift</th>
<th>( A^s[j] )</th>
<th>sorted cyclic shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>alf	extit{featsa}lf	extit{alfafa}$</td>
<td>14</td>
<td>$ a l f e a t s a l f a l f a $</td>
</tr>
<tr>
<td>1</td>
<td>lfe	extit{atsa}lf	extit{alfafa}$a</td>
<td>13</td>
<td>a$ a l f e a t s a l f a l f a $</td>
</tr>
<tr>
<td>2</td>
<td>fe	extit{atsa}lf	extit{alfafa}$a l</td>
<td>10</td>
<td>a l f a$ a l f e a t s a l f a $</td>
</tr>
<tr>
<td>3</td>
<td>e	extit{atsa}lf	extit{alfafa}$a l f</td>
<td>7</td>
<td>a l f a l f a$ a l f e a t s a l f a $</td>
</tr>
<tr>
<td>4</td>
<td>a	extit{tsa}lf	extit{alfafa}$a l f e</td>
<td>0</td>
<td>a l f e a t s a l f a l f a$</td>
</tr>
<tr>
<td>5</td>
<td>t	extit{sa}lf	extit{alfafa}$a l f e a</td>
<td>4</td>
<td>a t s a l f a l f a$ a l f e a t s a l f a $</td>
</tr>
<tr>
<td>6</td>
<td>s	extit{alfafa}$a l f e a t</td>
<td>3</td>
<td>e a t s a l f a l f a$ a l f e a t s a l f a</td>
</tr>
<tr>
<td>7</td>
<td>a	extit{lfafa}$a l f e a t s</td>
<td>12</td>
<td>f a$ a l f e a t s a l f a l f a$</td>
</tr>
<tr>
<td>8</td>
<td>l	extit{afa}$a l f e a t s a</td>
<td>9</td>
<td>f a$ a l f e a t s a l f a l f a$</td>
</tr>
<tr>
<td>9</td>
<td>f	extit{afa}$a l f e a t s a l</td>
<td>2</td>
<td>f e a t s a l f a l f a$ a l f e a t s a l f a</td>
</tr>
<tr>
<td>10</td>
<td>a	extit{fafa}$a l f e a t s a l f</td>
<td>11</td>
<td>l f a$ a l f e a t s a l f a l f a$</td>
</tr>
<tr>
<td>11</td>
<td>l	extit{fa}$a l f e a t s a l f a</td>
<td>8</td>
<td>l f a$ a l f e a t s a l f a l f a$</td>
</tr>
<tr>
<td>12</td>
<td>f	extit{a}$a l f e a t s a l f a l</td>
<td>1</td>
<td>l f e a t s a l f a l f a$ a l f e a t s a l f a</td>
</tr>
<tr>
<td>13</td>
<td>a	extit{$a}$a l f e a t s a l f a l f</td>
<td>6</td>
<td>s a l f a l f a$ a l f e a t s a l f a $</td>
</tr>
<tr>
<td>14</td>
<td>$ a l f e a t s a l f a $</td>
<td>5</td>
<td>t s a l f a l f a$ a l f e a t s a l f a $</td>
</tr>
</tbody>
</table>
BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

\[ S = \begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{a} & \text{l} & \text{f} & \text{e} & \text{a} & \text{t} & \text{s} & \text{a} & \text{l} & \text{f} & \text{a} & \text{l} & \text{f} & \text{a} & \$
\end{array} \]

\[ A^s = \begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
14 & 13 & 10 & 7 & 0 & 4 & 3 & 12 & 9 & 2 & 11 & 8 & 1 & 6 & 5
\end{array} \]

cyclic shift starts at $S[14]$
we need the last letter of that cyclic shift, it is at $S[13]$

\[ \text{a} \]
BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

<table>
<thead>
<tr>
<th>$S$</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a l f e a t s a l f a l f a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$A^s$</th>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14 13 10 7 0 4 3 12 9 2 11 8 1 6 5</td>
</tr>
</tbody>
</table>

Cyclic shift starts at $S[13]$

We need the last letter of that cyclic shift, it is at $S[12]$

a f
BWT Fast Encoding: Efficient Sorting

- Can read BWT encoding from suffix array in $O(n)$ time

$$S = \begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\text{a} & \text{l} & \text{f} & \text{e} & \text{a} & \text{t} & \text{s} & \text{a} & \text{l} & \text{f} & \text{a} & \text{l} & \text{f} & \text{a} & \$ \\
\end{array}$$

$$A^s = \begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
14 & 13 & 10 & 7 & 0 & 4 & 3 & 12 & 9 & 2 & 11 & 8 & 1 & 6 & 5 \\
\end{array}$$

cyclic shift starts at $S[5]$
we need the last letter of that cyclic shift, it is at $S[4]$

$$\text{a} \text{f} \text{f} \text{s} \$ \text{e} \text{f} \text{l} \text{l} \text{l} \text{l} \text{a} \text{a} \text{a} \text{a} \text{t} \text{a}$$
# BWT Fast Encoding: Efficient Sorting

<table>
<thead>
<tr>
<th>$j$</th>
<th>$A^s[j]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$$alfeatsalfalkf\alpha$</td>
</tr>
<tr>
<td>1</td>
<td>a$$alfeatsalfalkf$</td>
</tr>
<tr>
<td>2</td>
<td>alfa$$alfeatsalfalkf$</td>
</tr>
<tr>
<td>3</td>
<td>alfalfa$$alfeatsalkf$</td>
</tr>
<tr>
<td>4</td>
<td>alfeatsalfalkf$\alpha$</td>
</tr>
<tr>
<td>5</td>
<td>atsalfalkf$\alpha$le</td>
</tr>
<tr>
<td>6</td>
<td>eatsalfalkf$\alpha$lf</td>
</tr>
<tr>
<td>7</td>
<td>fa$$alfeatsalphal$</td>
</tr>
<tr>
<td>8</td>
<td>falaf$$alfeatsal$</td>
</tr>
<tr>
<td>9</td>
<td>featsal$\alpha$alf</td>
</tr>
<tr>
<td>10</td>
<td>lfa$$alfeatsal$</td>
</tr>
<tr>
<td>11</td>
<td>lfeatsal$\alpha$</td>
</tr>
<tr>
<td>12</td>
<td>lfeatsalfalkf$\alpha$</td>
</tr>
<tr>
<td>13</td>
<td>salfalkf$\alpha$leats</td>
</tr>
<tr>
<td>14</td>
<td>tsalfalkf$\alpha$le</td>
</tr>
</tbody>
</table>

The table above illustrates the process of Burrows-Wheeler Transform (BWT) for a given input sequence. Each row represents a rotation of the input sequence, and the corresponding $A^s[j]$ shows the resulting transformation.
BWT Decoding

\[ C = \text{affs}$\text{eflllalaata} \]

- Unsorted array of cyclic shifts
  - the first column is the original \( S \)

\[ \text{unsorted shifts array} \]

```
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
alfalfa$alfalfalfeats
```

BWT Decoding

\[ C = \text{affs}\$\text{efllllaata} \]

- Given \( C \), last column of sorted shifts array
- Can reconstruct the first column of sorted shifts array by sorting
  - first column has exactly the same characters as the last column
  - \textit{and} they must be sorted
BWT Decoding

\[ C = \text{affs$eflllaaata} \]

- Given \( C \), last column of sorted shifts array
- Can reconstruct the first column of sorted shifts array by sorting
  - first column has exactly the same characters as the last column
  - \textit{and} they must be sorted
- Also need row number for decoding

```
.....a,0
.....f,1
.....f,2
.....s,3
.....$ ,4
.....e,5
.....f,6
.....l,7
.....l,8
.....l,9
.....a,10
.....a,11
.....a,12
.....t,13
.....a,14
```
BWT Decoding

\[ C = \text{affs$efllllaata} \]

- Now have the first and the last columns of sorted shifts array
  - use stable sort
  - equal letters stay in the same order

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>,</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>a</td>
<td>,</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>f</td>
<td>,</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>a</td>
<td>,</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>f</td>
<td>,</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>a</td>
<td>,</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>s</td>
<td>,</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>a</td>
<td>,</td>
<td>1</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>$</td>
<td>,</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>a</td>
<td>,</td>
<td>1</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>e</td>
<td>,</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>e</td>
<td>,</td>
<td>5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>f</td>
<td>,</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>f</td>
<td>,</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>l</td>
<td>,</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>f</td>
<td>,</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>l</td>
<td>,</td>
<td>8</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>f</td>
<td>,</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>l</td>
<td>,</td>
<td>9</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>l</td>
<td>,</td>
<td>7</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>a</td>
<td>,</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>l</td>
<td>,</td>
<td>8</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>a</td>
<td>,</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>l</td>
<td>,</td>
<td>9</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>a</td>
<td>,</td>
<td>1</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>s</td>
<td>,</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>t</td>
<td>,</td>
<td>1</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>t</td>
<td>,</td>
<td>1</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>a</td>
<td>,</td>
<td>1</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
BWT Decoding

\[ C = \text{affs}eflllaaata \]

- Now have the first and the last columns of sorted shifts array
- Key for decoding is figuring out where in the sorted shifts array are the unsorted rows 0, 1, ...
- Where is row 0 of *unsorted* shifts array?
  - must end with ‘$’

\[
\begin{array}{c}
\$ , 4 \ldots \ldots \ldots a , 0 \\
a , 0 \ldots \ldots \ldots f , 1 \\
a , 10 \ldots \ldots \ldots f , 2 \\
a , 11 \ldots \ldots \ldots s , 3 \\
a , 12 \ldots \ldots \ldots \$ , 4 \\
a , 14 \ldots \ldots \ldots e , 5 \\
e , 5 \ldots \ldots \ldots f , 6 \\
f , 1 \ldots \ldots \ldots l , 7 \\
f , 2 \ldots \ldots \ldots l , 8 \\
f , 6 \ldots \ldots \ldots l , 9 \\
l , 7 \ldots \ldots \ldots a , 10 \\
l , 8 \ldots \ldots \ldots a , 11 \\
l , 9 \ldots \ldots \ldots a , 12 \\
s , 3 \ldots \ldots \ldots t , 13 \\
t , 13 \ldots \ldots \ldots a , 14 \\
\end{array}
\]
**BWT Decoding**

\[ C = \text{affs}$eflllaaata \]
\[ S = \text{a} \]

- Row = 0 of unsorted shifts starts with \text{a}
- Therefore
  - string \( S \) starts with \text{a}
- Where is row = 1 of the unsorted shifts array?

**sorted shifts array**

\[
\begin{align*}
&\$ , 4 . \ldots . \ldots . a , 0 \\
a , 0 . \ldots . \ldots . f , 1 \\
a , 10 . \ldots . \ldots . f , 2 \\
a , 11 . \ldots . \ldots . s , 3 \\
&\text{a , 1} \text{2} . \ldots . \ldots . \$ , 4 \\
a , 14 . \ldots . \ldots . e , 5 \\
e , 5 . \ldots . \ldots . f , 6 \\
f , 1 . \ldots . \ldots . l , 7 \\
f , 2 . \ldots . \ldots . l , 8 \\
f , 6 . \ldots . \ldots . l , 9 \\
l , 7 . \ldots . \ldots . a , 10 \\
l , 8 . \ldots . \ldots . a , 11 \\
l , 9 . \ldots . \ldots . a , 12 \\
s , 3 . \ldots . \ldots . t , 13 \\
t , 13 . \ldots . \ldots . a , 14
\end{align*}
\]
BWT Decoding

\[ C = \text{affs$eflllaaata} \]

- In the unsorted shifts array, any row ends with the first letter of previous row
  - unsorted row 1 ends with the same letter that unsorted row 0 begins with
BWT Decoding

$C = \text{affs$efllllaata}$

$S = a$

- Row $= 0$ of unsorted shifts starts with $a$
- Therefore
  - string $S$ starts with $a$
- Where is row $= 1$ of the unsorted shifts array?
  - row $= 1$ of unsorted shifts array ends with $a$

sorted shifts array

- $\$, 4, 0, a, 1, 12, $, 4, 0$
- $a, 0, f, 1$
- $a, 10, f, 2$
- $a, 11, s, 3$
- $a, 12, $, 4$
- $a, 14, e, 5$
- $e, 5, f, 6$
- $f, 1, l, 7$
- $f, 2, l, 8$
- $f, 6, l, 9$
- $l, 7, a, 10$
- $l, 8, a, 11$
- $l, 9, a, 12$
- $s, 3, t, 13$
- $t, 13, a, 14$
BWT Decoding

\[ C = \text{affs}\$eflllaaata \]

\[ S = a \]

- Row = 0 of unsorted shifts starts with \( a \)
- Therefore
  - string \( S \) starts with \( a \)
- Where is row = 1 of the unsorted shifts array?
  - row = 1 of unsorted shifts array ends with \( a \)
- Multiple rows end with \( a \), which one is row 1 of unsorted shifts?

<table>
<thead>
<tr>
<th>Sorted shifts array</th>
<th>Unsorted shifts array</th>
</tr>
</thead>
<tbody>
<tr>
<td>$, 4, \ldots, a, 0, \ldots$</td>
<td>( a, 0, \ldots, f, 1 )</td>
</tr>
<tr>
<td>( a, 10, \ldots, f, 2 )</td>
<td>( a, 11, \ldots, s, 3 )</td>
</tr>
<tr>
<td>( a, 12, \ldots, $, 4 )</td>
<td>( a, 14, \ldots, e, 5 )</td>
</tr>
<tr>
<td>( e, 5, \ldots, f, 6 )</td>
<td>( f, 1, \ldots, l, 7 )</td>
</tr>
<tr>
<td>( f, 2, \ldots, l, 8 )</td>
<td>( f, 6, \ldots, l, 9 )</td>
</tr>
<tr>
<td>( l, 7, \ldots, a, 10 )</td>
<td>( l, 8, \ldots, a, 11 )</td>
</tr>
<tr>
<td>( l, 9, \ldots, a, 12 )</td>
<td>( s, 3, \ldots, t, 13 )</td>
</tr>
<tr>
<td>( t, 13, \ldots, a, 14 )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>
BWT Algorithm and Example

$S = \text{alfeatsalfalfa}$

- Consider all patterns in sorted array that start with ‘a’
BWT Algorithm and Example

\( S = \text{alfeatsalfalfa}$

- Consider all patterns in sorted array that start with ‘a’
  - a$alfeatsalfalf
  - alfa$alfeatsalf
  - alfalfa$alfeats
  - alfeatsalfalfa$
  - a$tsalfalfa$alfe

- Take their cyclic shifts (by one letter)
  - $alfeatsalfalfa
  - lfa$alfeatsalf
  - lalfa$alfeatsa
  - lfeatsalfalfa$a
  - tsalfalfa$alfe

- Find them in sorted array of cyclic shifts
- They have ‘a’ at the end, and are the only rows that have ‘a’ at the end
- They appear in the same relative order as before cyclic shift
  - for patterns with same first letter, cyclic shift by one letter does not change relative sorting order
BWT Algorithm and Example

$S = \text{alfeatsalfalfa}\$

- Consider all patterns in sorted array that start with ‘a’
  - $a\$alfeatsalfalf$
  - alfa$\text{alfeatsalf}$
  - alfalfa$\text{alfeats}$
  - alfeatsalfalfa$
  - atsalfalfa$\text{alfe}$

- Take their cyclic shifts (by one letter)
  - $\text{alfeatsalfalfa} a$
  - $\text{alfeatsalfalfa} a$
  - $\text{alfeatsalfalfa} a$
  - $\text{alfeatsalfalfa} a$
  - $\text{alfeatsalfalfa} a$

- Unsorted row 1 is a cyclic shift by 1 letter of unsorted row 0
  - unsorted row 0 is #4 among all rows starting with ‘a’
  - unsorted row 1 is #4 among all rows ending with ‘a’
BWT Algorithm and Example

\[ S = \text{alfeatsalfalfa}$ \]

- Consider all patterns in sorted array that start with ‘a’
  \[ \text{a}$\text{alfeatsalfalf} \]
  \[ \text{alpha}$\text{alfeatsalf} \]
  \[ \text{alfalfa}$\text{alfeats} \]
  \[ \text{alfeatsalfalfa}$ \]
  \[ \text{atsalfalfa}$\text{alfe} \]

- Take their cyclic shifts (by one letter)
  \[ \text{alfeatsalfalfa} \]
  \[ \text{lfa}$\text{alfeatsalf} \]
  \[ \text{lfa}$\text{alfeatsalsa} \]
  \[ \text{alfeatsalfalfa} \]
  \[ \text{tsalfalfa}$\text{alfe} \]

- Unsorted row 1 is a cyclic shift by 1 letter of unsorted row 0
  - unsorted row 0 is #4 among all rows starting with ‘a’
  - unsorted row 1 is #4 among all rows ending with ‘a’
BWT Decoding

\[ C = \text{affs}\$e\ell\ell\ell\lalaata \]
\[ S = a^l \]

- Multiple rows end with a, which one is row 1 of unsorted shifts?
- Unsorted row = 1 is located in row 12 of the sorted shifts
  - \( S[1] = l \)
BWT Decoding

\[ C = \text{affs}$$efllllaaata \]
\[ S = \text{alf} \]

- Unsorted row \( = 2 \) is located in row 9 of the sorted shifts
  - \( S[2] = f \)

<table>
<thead>
<tr>
<th>sorted shifts array</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
</tr>
<tr>
<td>$, 4 \ldots \ldots \ldots a, 0</td>
</tr>
<tr>
<td>a, 0 \ldots \ldots \ldots f, 1</td>
</tr>
<tr>
<td>a, 10 \ldots \ldots \ldots f, 2</td>
</tr>
<tr>
<td>a, 11 \ldots \ldots \ldots s, 3</td>
</tr>
<tr>
<td>a, 12 \ldots \ldots \ldots $, 4</td>
</tr>
<tr>
<td>a, 14 \ldots \ldots \ldots e, 5</td>
</tr>
<tr>
<td>e, 5 \ldots \ldots \ldots f, 6</td>
</tr>
<tr>
<td>f, 1 \ldots \ldots \ldots l, 7</td>
</tr>
<tr>
<td>f, 2 \ldots \ldots \ldots l, 8</td>
</tr>
<tr>
<td>f, 6 \ldots \ldots \ldots l, 9</td>
</tr>
<tr>
<td>l, 7 \ldots \ldots \ldots a, 10</td>
</tr>
<tr>
<td>l, 8 \ldots \ldots \ldots a, 11</td>
</tr>
<tr>
<td>l, 9 \ldots \ldots \ldots a, 12</td>
</tr>
<tr>
<td>s, 3 \ldots \ldots \ldots t, 13</td>
</tr>
<tr>
<td>t, 13 \ldots \ldots \ldots a, 14</td>
</tr>
<tr>
<td>row 1</td>
</tr>
<tr>
<td>\text{highlighted cell}</td>
</tr>
<tr>
<td>row 2</td>
</tr>
<tr>
<td>\text{highlighted cell}</td>
</tr>
</tbody>
</table>
BWT Decoding

\[ C = \text{affs}\$eflllaata \]
\[ S = \text{alf}e \]

- Unsorted row = 3 is located in row 6 of the sorted shifts
  - \( S[3] = e \)

<table>
<thead>
<tr>
<th>row 0</th>
<th>row 1</th>
<th>row 2</th>
<th>row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$, 4 . . . . . . a, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, 0 . . . . . . f, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, 1 0 . . . . . . f, 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, 1 1 . . . . . . s, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, 1 2 . . . . . . $, 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a, 1 4 . . . . . . e, 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e, 5 . . . . . . f, 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f, 1 . . . . . . l, 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f, 2 . . . . . . l, 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{red} f, 6 . . . . . . l, 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l, 7 . . . . . . a, 1 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l, 8 . . . . . . a, 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l, 9 . . . . . . a, 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s, 3 . . . . . . t, 1 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t, 1 3 . . . . . . a, 1 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BWT Decoding

\[ C = \text{affs}\$efllllaaata \]
\[ S = \text{alfe}a \]

- Unsorted row = 4 is located in row 5 of the sorted shifts
  - \( S[4] = a \)
**BWT Decoding**

\[ S = \text{alfeatsalfalfa} $\]

<table>
<thead>
<tr>
<th>Sorted Shifts Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>$, 4 ........ a, 0</td>
</tr>
<tr>
<td>a, 0 ........ f, 1</td>
</tr>
<tr>
<td>a, 10 ........ f, 2</td>
</tr>
<tr>
<td>a, 11 ........ s, 3</td>
</tr>
<tr>
<td>a, 12 ........ $, 4</td>
</tr>
<tr>
<td>a, 14 ........ e, 5</td>
</tr>
<tr>
<td>e, 5 ........ f, 6</td>
</tr>
<tr>
<td>f, 1 ........ l, 7</td>
</tr>
<tr>
<td>f, 2 ........ l, 8</td>
</tr>
<tr>
<td>f, 6 ........ l, 9</td>
</tr>
<tr>
<td>l, 7 ........ a, 10</td>
</tr>
<tr>
<td>l, 8 ........ a, 11</td>
</tr>
<tr>
<td>l, 9 ........ a, 12</td>
</tr>
<tr>
<td>s, 3 ........ t, 13</td>
</tr>
<tr>
<td>t, 13 .......... a, 14</td>
</tr>
</tbody>
</table>

*row 0*  
*row 1*  
*row 2*  
*row 3*  
*row 4*  
*row 5*  
*row 6*  
*row 7*  
*row 8*  
*row 9*  
*row 10*  
*row 11*  
*row 12*  
*row 13*  
*row 14*
**BWT Decoding**

\[ \text{BWT::decoding}(C[0 \ldots n - 1], S) \]

\( C \): string of characters over alphabet \( \Sigma_c \), \( S \): output stream

\[ A \leftarrow \text{array of size } n \quad \// \text{leftmost column} \]

\[ \text{for } i = 0 \text{ to } n - 1 \]

\[ A[i] \leftarrow (C[i], i) \quad \// \text{store character and index} \]

stably sort \( A \) by character

\[ \text{for } j = 0 \text{ to } n \quad \// \text{find $} \]

\[ \text{if } C[j] = $ \text{ break} \]

repeat

\[ S.\text{append}(\text{character stored in } A[j]) \]

\[ j \leftarrow \text{index stored in } A[j] \]

until we have appended $
BWT Overview

- **Encoding cost**
  - \( O(n \log n) \) with special sorting algorithm
  - in practice MSD sort is good enough but worst case is \( \Theta(n^2) \)
  - read encoding from the suffix array

- **Decoding cost**
  - faster than encoding
  - \( O(n + |\Sigma_s|) \)

Encoding and decoding both use \( O(n) \) space

- They need all of the text (no streaming possible)
  - can use on blocks of text (block compression method)

- BWT tends to be slower than other methods

- But combined with MTF, RLE and Huffman leads to better compression
## Compression Summary

<table>
<thead>
<tr>
<th>Huffman</th>
<th>Run-length encoding</th>
<th>Lempel-Ziv-Welch</th>
<th>Bzip2 (uses Burrows-Wheeler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable-length</td>
<td>variable-length</td>
<td>fixed-length</td>
<td>multi-step</td>
</tr>
<tr>
<td>single-character</td>
<td>multi-character</td>
<td>multi-character</td>
<td>multi-step</td>
</tr>
<tr>
<td>2-pass</td>
<td>1-pass</td>
<td>1-pass</td>
<td>not streamable</td>
</tr>
<tr>
<td>60% compression on English text</td>
<td>bad on text</td>
<td>45% compression on English text</td>
<td>70% compression 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>
<td>better on English text</td>
</tr>
<tr>
<td>requires uneven frequencies</td>
<td>requires runs</td>
<td>requires repeated substrings</td>
<td>requires repeated substrings</td>
</tr>
<tr>
<td>rarely used directly</td>
<td>rarely used directly</td>
<td>frequently used</td>
<td>used but slow</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>
<td>bzip2 and variants</td>
</tr>
</tbody>
</table>