BINARY CHARACTER CODES

for

LOSSLESS COMPRESSION

- Introduction to data compression - Khalid Sayood

& CLRS
BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string
BINARY CHARACTER CODES

How to encode \text{...ABACBBFAEFDACB...} in binary, knowing full string

\text{alphabet}

A B C D E F

\begin{align*}
000 & \quad 001 & \quad 010 & \quad 011 & \quad 100 & \quad 101 & \quad \rightarrow \\
\end{align*}

3 bits/char.

impossible w/ 2 bits
**BINARY CHARACTER CODES**

How to encode ...ABACBBFAEDACB... in binary, knowing full string

**alphabet**

A  B  C  D  E  F

000  001  010  011  100  101 → 3 bits/char.

Why not 0 1 01 011 100 101? impossible w/ 2 bits
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>ambiguous</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>ambiguous</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
<td>uniquely decodable</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>------</td>
<td>----------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>ambiguous</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
<td>uniquely decodable</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>?</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>Notes</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>ambiguous</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
<td>uniquely decodable</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>uniquely decodable, but not instantaneous</td>
</tr>
</tbody>
</table>

O1 = "B" or continue?
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>terrible</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>00</td>
<td>11</td>
<td>ambiguous</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
<td>uniquely decodable</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>011</td>
<td>0111</td>
<td>uniquely decodable but not instantaneous</td>
</tr>
</tbody>
</table>

0
  01
  11

\[\begin{array}{c}
\text{A} \\
\text{C} \\
\text{CCCCC...} \\
\text{BBBBBBBBBBBB} \\
\text{C C C C C}
\end{array}\]
How do we know if a code is uniquely decodable?

0 01 11

A B C

A C C C C C ...

0 1 1 1 1 1 1}

resolved at end

B C C C C
How do we know if a code is uniquely decodable?

0 01 11

A

B

C

0 01 10

? 

\{ resolved at end \\

A

\[ \]

0

11

B

C

C

C

C

C

C
How do we know if a code is uniquely decodable?

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{resolved at end} \]

\[ \text{Problem} \]
How do we know if a code is uniquely decodable?

If we encoded ACC... then decoding BCC... fails (and vice versa).

Problem

Whatever we encoded, both decodings work.
How do we know if a code is uniquely decodable?

$$A = c_1 c_2 c_3 \cdots c_K$$

$$B = c_1 c_2 c_3 \cdots c_K d_1 d_2 \cdots d_j$$

$$\{ A \text{ is a prefix of } B \}$$
How do we know if a code is uniquely decodable?

\[ A = c_1 c_2 c_3 \ldots c_k \]

\[ B = c_1 c_2 c_3 \ldots c_k d_1 d_2 \ldots d_j \]

\{ A \text{ is a prefix of } B \}

"dangling suffix"
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1c_2c_3 \ldots c_k \]
\[ B = c_1c_2c_3 \ldots c_k d_1d_2 \ldots d_j \]

\{ A \text{ is a prefix of } B \ldots \}
\{ \text{“dangling suffix”...is another codeword} \}

\[
\begin{array}{ccc}
A & B & C \\
0 & 01 & 1
\end{array}
\]
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1 c_2 c_3 \ldots c_k \]
\[ B = c_1 c_2 c_3 \ldots c_k d_1 d_2 \ldots d_j \]
\{ A is a prefix of B... and "dangling suffix"... is another codeword \}

(but also not a sufficient test)

\begin{align*}
0 & \rightarrow 01 \\
A & \rightarrow B \\
0 & \rightarrow 01 \\
\text{also ok?!}
\end{align*}
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1c_2c_3 \ldots c_k \]
\[ B = c_1c_2c_3 \ldots c_k d_1d_2 \ldots d_j \]
\{ \text{A is a prefix of B...} \}
\text{ and }
\text{"dangling suffix"...is another codeword}

\begin{align*}
0 & \quad 01 \\
A & \quad B \\
\hline
0 & \quad 01 \\
0 & \quad 01 \\
10 & \quad \hat{=} \\
\hline
\end{align*}

Extended test: if \exists \text{ dangling suffix}
How do we know if a code is uniquely decodable?

**Not good:**

\[ A = \ldots c_k \]

\[ B = \ldots c_k d_1 d_2 \ldots d_j \]

\( A \) is a prefix of \( B \)...

and

"dangling suffix"... is another codeword

<table>
<thead>
<tr>
<th>0</th>
<th>01</th>
<th>11</th>
<th>😊</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>😎</td>
</tr>
</tbody>
</table>

**Extended test:** if \( \exists \) dangling suffix add it to a list (set)
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1 c_2 c_3 \ldots c_k \]
\[ B = c_1 c_2 c_3 \ldots c_k d_1 d_2 \ldots d_j \]
\{ A is a prefix of B... and
"dangling suffix"... is another codeword

<table>
<thead>
<tr>
<th>0</th>
<th>01</th>
<th>11</th>
<th>😊</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
<td>😋</td>
</tr>
</tbody>
</table>

Extended test: if \exists dangling suffix
add it to a list (set)
& repeat original test
on combined set
until no new listwords
get generated.

0 01 11 1
  OK
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1 c_2 c_3 \ldots c_k \]
\[
B = c_1 c_2 c_3 \ldots c_k \overline{d_1 d_2 \ldots d_j} \}
\] \(A\) is a prefix of \(B\)...
and
"dangling suffix"... is another codeword.

Extended test: if \(\exists\) dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.

\[
\begin{array}{ccc}
0 & 01 & 11 \ \\
A & B & C \\
0 & 01 & 10 \ \\
\end{array}
\]
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1c_2c_3 \ldots c_k \]
\[ B = c_1c_2c_3 \ldots c_k d_1d_2 \ldots d_j \]
\{ A is a prefix of B ... and "dangling suffix" ... is another codeword \}

Extended test: if \exists dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.
How do we know if a code is uniquely decodable?

Not good: \[ A = c_1 c_2 c_3 \ldots c_k \] \[ B = c_1 c_2 c_3 \ldots c_k d_1 d_2 \ldots d_j \] \( \{ \) A is a prefix of B... and "dangling suffix"... is another codeword

Extended test: if \( \exists \) dangling suffix add it to a list (set) & repeat original test on combined set until no new listwords get generated.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>

STILL OK

NOT OK

PROOF OMITTED

OK

!!!
BINARY CHARACTER CODES

How to encode \(...ABACBBFAEFDACB...\) in binary, knowing full string.

alphabet

A: 45  
B: 13  
C: 12  
D: 16  
E: 9   
F: 5

3 bits/char.

impossible w/ 2 bits.
How to encode \( \ldots \text{ABACBBFAEFDACB} \ldots \) in binary, knowing full string

\textbf{alphabet}

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

\( \% \text{ frequencies} \)

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
</tr>
</thead>
</table>

\( \rightarrow \) 3 bits/char.

impossible w/ 2 bits

\( \texttt{0 101 100 111 1101 1100} \) : Variable length code
How to encode \( \ldots \text{ABACBBFAEFDACB} \ldots \) in binary, knowing full string

**alphabet**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ \sum (\text{freq. bits}) = 0.45 \cdot 1 + (0.13 + 0.12 + 0.16) \cdot 3 + (0.09 + 0.05) \cdot 4 \]

\[ \text{average 3 bits/char.} \]

impossible w/ 2 bits

\[ \text{Variable length code} \]
BINARY CHARACTER CODES

How to encode ...ABACBBFAEFDACB... in binary, knowing full string

alphabet

A B C D E F
45 13 12 16 9 5 : % frequencies
000 001 010 011 100 101 \rightarrow average 3 bits/char.
impossible w/ 2 bits

0 101 100 111 1101 1100 : Variable length code

\[ \sum (\text{freq.bits}) \cdot \begin{align*}
0.45 \cdot 1 &+ (0.13 + 0.12 + 0.16) \cdot 3 &+ (0.09 + 0.05) \cdot 4 \\
\end{align*} \rightarrow \text{average 2.24 bits/char.} \]
BINARY CHARACTER CODES

How to encode \( \ldots ABACBBFAEFDACB \ldots \) in binary, knowing full string

**alphabet**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13</td>
<td>17</td>
<td>16</td>
<td>19</td>
<td>15</td>
</tr>
</tbody>
</table>

: % frequencies

000 001 010 011 100 101 \( \rightarrow \) average 3 bits/char.

impossible w/ 2 bits

10 000 010 011 11 001 : Variable length code

\[ .2 \cdot 2 + (0.13+0.17+0.16) \cdot 3 + 0.19 \cdot 2 + 0.15 \cdot 3 \]\n
\( \rightarrow \) average 2.46 bits/char.
BINARY CHARACTER CODES

How to encode \ldots ABACBBFAEFDACB\ldots in binary, knowing full string

alphabet

\begin{align*}
A & \quad 95 \\
B & \quad 1 \\
C & \quad 1 \\
D & \quad 1 \\
E & \quad 1 \\
F & \quad 1 \\
\end{align*}

3rd example $\rightarrow 000 \quad 001 \quad 010 \quad 011 \quad 100 \quad 101 \Rightarrow \text{average 3 bits/char.}$

for fixed length: impossible w/ 2 bits

$1 \quad 000 \quad 001 \quad 0100 \quad 0101 \quad 011 : \text{Variable length code}$

$.95 \cdot 1 + (.01+.01) \cdot 3 + (.01+.01) \cdot 4 + .01 \cdot 3 \Rightarrow \text{average 1.12 bits/char.}$
How to encode AACBADDAEFDBACB...

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
</table>
| 45 | 13 | 12 | 16 | 9 | 5 | % frequencies
How to encode A A A C B A D A E F D A C B...

000 • 000 • 000 • 010 • 001 • 000 • 011 • 000 • 100 • 101 • 011 • 000 • 010 • 001...

0 • 0 • 0 • 100 • 101 • 0 • 110 • 1101 • 1100 • 111 • 0 • 100 • 101

A A A C B A D A E F D A C B...

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

000 001 010 011 100 101 : fixed length code

0 101 100 111 1101 1100 : variable length code
How to encode: A A A C B A D A E F D A C B...

Fixed: 000000000010001000011000100101011000010001

Variable: 00010010101010110011001100110100010001

A B C D E F
45 13 12 16 9 5 : % frequencies
000 001 010 011 100 101 : fixed length code
0 101 100 111 1101 1100 : Variable length code

why ok?
Prefix codes: no code is a prefix of another, e.g., only A starts with 0

Unlike, say, A=01, B=011, C=1 → 011 = B or AC?
We still need to know when each code ends

How do we know there is no 0001?

0001001010111010101110011110100101

A    B    C    D    E    F
45   13   12   16    9    5 : % frequencies
0    101   100   111  1101  1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

A

45 13 12 16 9 5 : % frequencies
0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

A
B

45 13 12 16 9 5 : % frequencies
0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

A

B

C

D

E

F

45 13 12 16 9 5 : % frequencies
0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends  ➔ use binary tree

A

C  B

D

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

| 0 | 101 | 100 | 111 | 1101 | 1100 |

: Variable length (prefix) code
We still need to know when each code ends $\rightarrow$ use binary tree

$00010010101101101001101010$

A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

$010110111111011100$ : Variable length (prefix) code
We still need to know when each code ends \( \rightarrow \) use binary tree

000100101001001101110101

AA

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

0 101 100 111 1101 1100

: Variable length (prefix) code
We still need to know when each code ends → use binary tree

```
00010010101101101101100110100101
AAA
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

```
0 101 100 111 1101 1100 : Variable length (prefix) code
```
We still need to know when each code ends → use binary tree

AAAA

A  B  C  D  E  F  
45 13 12 16  9  5 : % frequencies
0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>01</th>
<th>100</th>
<th>11</th>
<th>1101</th>
<th>1100</th>
</tr>
</thead>
</table>

: Variable length (prefix) code
We still need to know when each code ends → use binary tree

AAA C

A B C D E F
45 13 12 16 9 5 : % frequencies
0 101 100 111 1101 1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

AAA c

A  B  C  D  E  F
45  13  12  16  9  5 : % frequencies
0  101  100  111  1101  1100 : Variable length (prefix) code
We still need to know when each code ends. → Use binary tree.

AAA C B

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

% frequencies

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Variable length (prefix) code
We still need to know when each code ends → use binary tree

000100101011011011011110100101

AAA C BA

A  B  C  D  E  F
45  13  12  16  9  5 : % frequencies
0  101  100  111  1101  1100 : Variable length (prefix) code
We still need to know when each code ends → use binary tree

0001001010110110101101001100101
AAA C BA D etc

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

: % frequencies

0 101 100 111 1101 1100 : Variable length (prefix) code
Huffman code

developed in 1951 as a solution to a class assignment

typical compression 20-90%
Huffman code: Make n trees of size 1: one tree per character.

A  B  C  D  E  F
45 13 12 16 9   5 : %
Huffman code: Make \( n \) trees of size 1: one tree per character.

- The 2 roots with lowest frequencies become siblings.
Huffman code: Make n trees of size 1: one tree per character.

- The 2 roots with lowest frequencies become siblings.
Huffman code:
Make n trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.

A  45
B  13
C  12
D  16
E  9
F  5

14
Huffman code:

Make n trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.
Repeat
Huffman code:

Make n trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.
Repeat.

A 45
B 13
C 12
D 16
E 9
F 5

25

12

13

14

5

9

F

E
Huffman code: Make \( n \) trees of size 1: one tree per character. The 2 roots with lowest frequencies become siblings. Their new parent gets the sum of their frequencies. Repeat.
Huffman code:

Make \( n \) trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.
Repeat

\[
\begin{array}{ccccccc}
A & B & C & D & E & F & \% \\
45 & 13 & 12 & 16 & 9 & 5 & \\
\end{array}
\]
Huffman code:

- Make \( n \) trees of size 1: one tree per character.
- The 2 roots with lowest frequencies become siblings.
- Their new parent gets the sum of their frequencies.
- Repeat
Huffman code: Make $n$ trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.

Repeat
Huffman code:

Make $n$ trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.
Repeat.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

55

25

12

C

13

B

14

16

5

F

9

E

D
Huffman code:

Make $n$ trees of size 1: one tree per character.
The 2 roots with lowest frequencies become siblings.
Their new parent gets the sum of their frequencies.
Repeat until 1 root:

A  45
B  13
C  12
D  16
E  9
F  5 : %
Huffman code: (greedy algo)

Make \( n \) trees of size 1: one tree per character. The 2 roots with lowest frequencies become siblings. Their new parent gets the sum of their frequencies. Repeat until 1 root:

\[
\begin{array}{c}
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} \\
45 & 13 & 12 & 16 & 9 & 5
\end{array}
\]

\[
0 \quad 101 \quad 100 \quad 111 \quad 1101 \quad 1100 : \text{Huffman code}
\]
Huffman code: Make n trees of size 1: one tree per character. The 2 roots with lowest frequencies become siblings. Their new parent gets the sum of their frequencies. Repeat until 1 root:

- Optimal! to show
- Not unique
  - arbitrary sibling order
  - weights could be equal

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
<td>%</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
<td>:</td>
<td>Huffman code</td>
</tr>
</tbody>
</table>
Huffman code: time to build?
Huffman code: easy to implement in $O(n \log n)$ time & $\Theta(n)$ space

priority queue to identify 2 lowest frequencies

initialize with $n$ chars: $\Theta(n)$

extract twice, insert new parent node: $O(\log n)$

$n-1$ rounds
Huffman code: easy to implement in $O(n \log n)$ time & $\Theta(n)$ space

priority queue to identify 2 lowest frequencies

- initialize with $n$ chars: $\Theta(n)$
- extract twice, insert new parent node: $O(\log n)$
- $n-1$ rounds

$O(n \log n \log n)$ possible, using a van Emde Boas tree

CLRS ch. 20
Huffman code: Not unique

Must store representation, like BST, otherwise decoder can't work.

\[ \Rightarrow \text{costs extra space \( \ddagger \)} \]
OPTIMALITY

\[
\text{minimize } \sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum (\text{freq.} \cdot \text{node depth})
\]

- Not optimal number of bits in general (more later)

- We get OPT assuming one codeword per character
OPTIMALITY

\[
\text{minimize } \sum_{\text{all chars}} (\text{freq.} \cdot \text{bit length}) = \sum_{\text{all chars}} (\text{freq.} \cdot \text{node depth})
\]

Claim: there is always a prefix code that achieves OPT

- CLRS skips the proof
- See Sayood 2.4.3: 3-page proof
Claim 2: 
\text{OPT} \text{ tree can't have an internal node w/ 1 child}
Optimality

minimize $\sum (\text{freq} \cdot \text{bit length}) = \sum (\text{freq} \cdot \text{node depth})$

Claim 2:

OPT tree can't have an internal node w/ 1 child

$\Downarrow$ contract, improve $\Sigma$, still a prefix code: contradiction
**OPTIMALITY**  
minimize $\sum (\text{freq.} \cdot \text{node depth})$

✓ OPT tree can't have an internal node w/ 1 child

Claim 3:

∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level
**Optimality**

\[
\text{minimize } \sum (\text{freq.} \cdot \text{node depth})
\]

- OPT tree can't have an internal node with 1 child

**Claim 3:**

∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

\[\iff\] by contradiction: take "OPT", swap nodes, improve \(\Sigma\)
minimize \( \sum (\text{freq.} \times \text{node depth}) \)

- OPT tree can't have an internal node w/ 1 child
- \( \exists \) OPT tree w/ 2 lowest frequencies as siblings at lowest level
OPTIMALITY

minimize $\sum (\text{freq.} \cdot \text{node depth})$

- OPT tree can't have an internal node with 1 child
- $\exists$ OPT tree with 2 lowest frequencies as siblings at lowest level

Last claim:

if we merge 2 lowest frequencies $x,y$ in tree $T$
and new tree $T'$ is OPT'
then $T$ was OPT.
OPTIMALITY

\[ \text{minimize } \sum (\text{freq.} \cdot \text{node depth}) \]

✓ OPT tree can't have an internal node with 1 child

✓ ∃ OPT tree with 2 lowest frequencies as siblings at lowest level

Last claim:

if we merge 2 lowest frequencies \(x,y\) in tree \(T\) and new tree \(T'\) is OPT' (node \(z : x.\text{freq} + y.\text{freq}\))

then \(T\) was OPT.

Weight diff for \(T,T' = 1 \cdot (x.\text{freq} + y.\text{freq})\)  // all other nodes unchanged
OPTIMALITY

minimize \( \sum (\text{freq} \cdot \text{node depth}) \)

✓ OPT tree can't have an internal node w/ 1 child
✓ ∃ OPT tree w/ 2 lowest frequencies as siblings at lowest level

Last claim:

if we merge 2 lowest frequencies \( x, y \) in tree \( T \) and new tree \( T' \) is OPT' (node \( z : x.f\text{freq} + y.f\text{req} \))
then \( T \) was OPT.

Weight diff for \( T, T' = 1 \cdot (x.f\text{req} + y.f\text{req}) \) // all other nodes unchanged

Same for diff of any OPT tree that beats \( T \): \( x, y \) must be siblings so we can contract them & get a tree that beats \( T' \); contradiction
OPTIMALITY

minimize \( \sum (\text{freq.} \cdot \text{node depth}) \)

\( \checkmark \) OPT tree can’t have an internal node \( w/ 1 \) child

\( \checkmark \) \( \exists \) OPT tree \( w/ 2 \) lowest frequencies as siblings at lowest level

\( \checkmark \) if we merge \( 2 \) lowest frequencies \( x,y \) in tree \( T \) and new tree \( T' \) is optimal then \( T \) was OPT.

\( \checkmark \) merge 2 smallest, apply induction, uncontract \( \rightarrow \) get OPT

\( \checkmark \) incremental construction is equivalent
A few words about extensions & other ideas
Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue
Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue
  - place frequencies (single nodes) in queue A
  - queue B holds merged nodes
Huffman code with presorted frequencies

- use 2 queues instead of 1 priority queue
  - place frequencies (single nodes) in queue A
  - queue B holds merged nodes
- merge two smallest frequencies & insert in B, repeat

(min two elements are always at top 2+2 positions)
Huffman code with presorted frequencies \[ O(n) \] time

- Use 2 queues instead of 1 priority queue
  - Place frequencies (single nodes) in queue A
  - Queue B holds merged nodes
- Merge two smallest frequencies & insert in B, repeat
  (min two elements are always at top 2+2 positions)
- Proof of correctness is straightforward
Huffman code with minimum codeword length (variance)
Huffman code with minimum codeword length (variance)

- prioritize merging shorter trees
  - sort all frequencies, place in queue A
  - use queue B as for presorted data
Huffman code with minimum codeword length (variance)

- prioritize merging shorter trees
  - sort all frequencies, place in queue A
  - use queue B as for presorted data
  - when dequeuing min two elements, prioritize first queue if tied
Decompressing options (affects storage/transmission)

- store frequencies (could be really bad)
  - store tree
  - use "canonical tree"

: adaptive Huffman coding
0.8  0.02  0.18
0.8 + 0.04 + 0.36 = 1.2

Improve this by encoding blocks
A  B  C
0  11  10
0.8  0.02  0.18

0.8 + 0.04 + 0.36 = 1.2

AA  0.64
AB  0.016
AC  0.144
BA  0.016
BB  0.0004
BC  0.0036
CB  0.0036
CC  0.0324
\[0.8 + 0.04 + 0.36 = 1.2\]

<table>
<thead>
<tr>
<th>Pair</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>0.64</td>
</tr>
<tr>
<td>AB</td>
<td>0.016</td>
</tr>
<tr>
<td>AC</td>
<td>0.144</td>
</tr>
<tr>
<td>BA</td>
<td>0.016</td>
</tr>
<tr>
<td>BB</td>
<td>0.0004</td>
</tr>
<tr>
<td>BC</td>
<td>0.0036</td>
</tr>
<tr>
<td>CA</td>
<td>0.144</td>
</tr>
<tr>
<td>CB</td>
<td>0.0036</td>
</tr>
<tr>
<td>CC</td>
<td>0.0324</td>
</tr>
</tbody>
</table>
$0.8 + 0.04 + 0.36 = 1.2$

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>0</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>0.8</td>
<td>0.02</td>
<td>0.18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0.0004

0.0036

0.0324

AA 0.64
AB 0.016
AC 0.144
BA 0.016
BB 0.0004
BC 0.0036
CA 0.144
CB 0.0036
CC 0.0324
0.8 + 0.04 + 0.36 = 1.2

AA  0.64
AB  0.016
AC  0.144
BA  0.016
BB  0.0004
BC  0.0036
CA  0.144
CB  0.0036
CC  0.0324
$0.8 + 0.04 + 0.36 = 1.2$

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>O</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

0.8  0.02  0.18

AA 0.64
AB 0.016
AC 0.144
BA 0.016
BB 0.0004
BC 0.0036
CA 0.144
CB 0.0036
CC 0.0324

0.016
0.0236
0.0076
0.0004
0.0396
0.016
0.0036
0.004

AB
BA
BB
CB
CA
\[
0.8 + 0.04 + 0.36 = 1.2
\]
0.8 + 0.04 + 0.36 = 1.2

0.8  0.02  0.18
A  B  C
0  11  10

0.8  0.02  0.18

0.8 + 0.04 + 0.36 = 1.2

AA  0.64
AB  0.016
AC  0.144
BA  0.016
BB  0.0004
BC  0.0036
CA  0.144
CB  0.0036
CC  0.0324
0.8 + 0.04 + 0.36 = 1.2
A B C
0 11 10
0.8 0.02 0.18

0.8 + 0.04 + 0.36 = 1.2

<table>
<thead>
<tr>
<th>AA</th>
<th>0.64</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>0.016</td>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>AC</td>
<td>0.144</td>
<td>1 1</td>
</tr>
<tr>
<td>BA</td>
<td>0.016</td>
<td>1 0 1 0 0 0</td>
</tr>
<tr>
<td>BB</td>
<td>0.0004</td>
<td>1 0 1 0 0 1 0 1</td>
</tr>
<tr>
<td>BC</td>
<td>0.0036</td>
<td>1 0 1 0 0 1 1</td>
</tr>
<tr>
<td>CA</td>
<td>0.144</td>
<td>1 0 0</td>
</tr>
<tr>
<td>CB</td>
<td>0.0036</td>
<td>1 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>CC</td>
<td>0.0324</td>
<td>1 0 1 1</td>
</tr>
</tbody>
</table>

1.7228 bits/symbol
\[
\frac{1.7228}{2} = 0.8614
\]
errors do not propagate (Huffman: one incorrect bit can ruin everything)
TBC?
errors do not propagate (Huffman: one incorrect bit can ruin everything)