Bitwise Representations & Compression & Huffman Coding

This will be handy for the Huffman assignment...

How are primitives stored?

m a c

"Primitive" includes "pointer" here.

How are primitives stored?

m	a	c
109	97	99

"Primitive" includes "pointer" here.

How are primitives stored?

m	a	c
109	97	99
01101101	01100001	01100011

One *binary digit* (“bit”)

“Primitive” includes “pointer” here.

How are primitives stored?

01101101011000010110011

"Primitive" includes "pointer" here.

How are primitives stored?

01101101011000010110100011

0110 1101 0110 0001 0110 0011

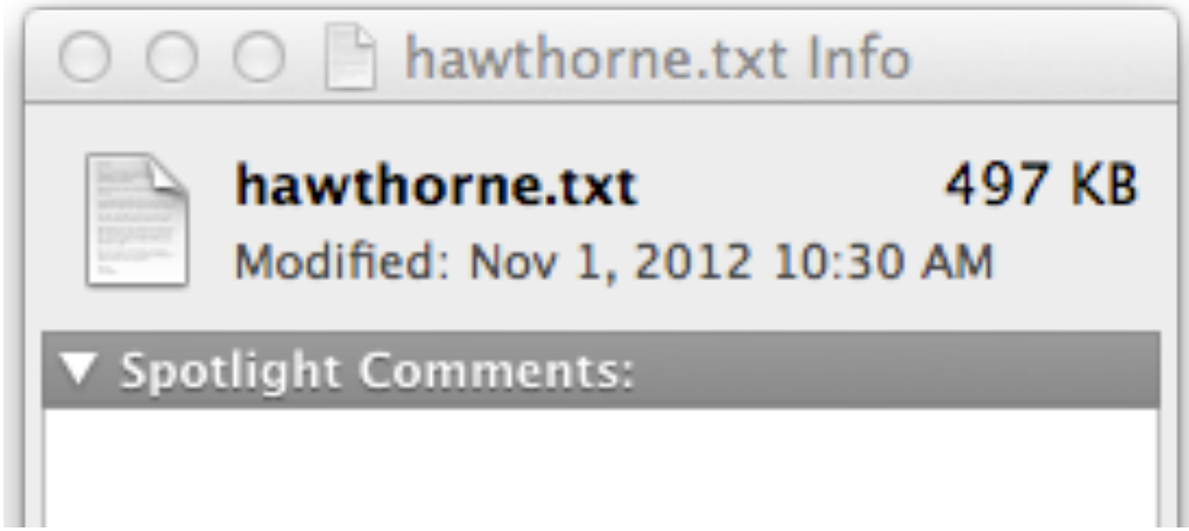
6 13 6 1 6 3

You have to know your *encoding*.

"Primitive" includes "pointer" here.

Text data (ASCII)

8 bits per character (one *byte*).



← *K* for kilo

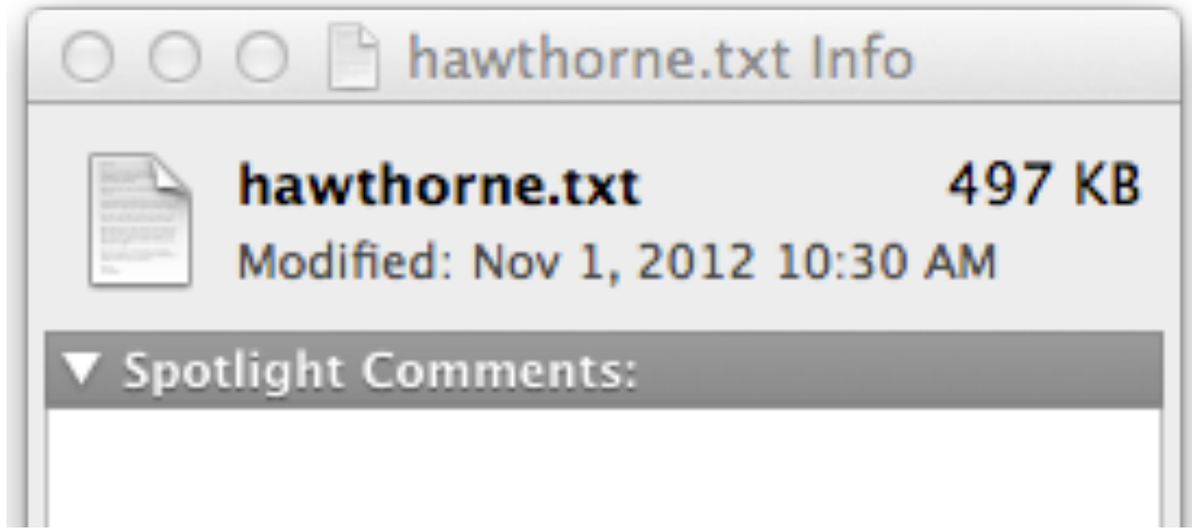
B for byte

≈ 508,928 characters

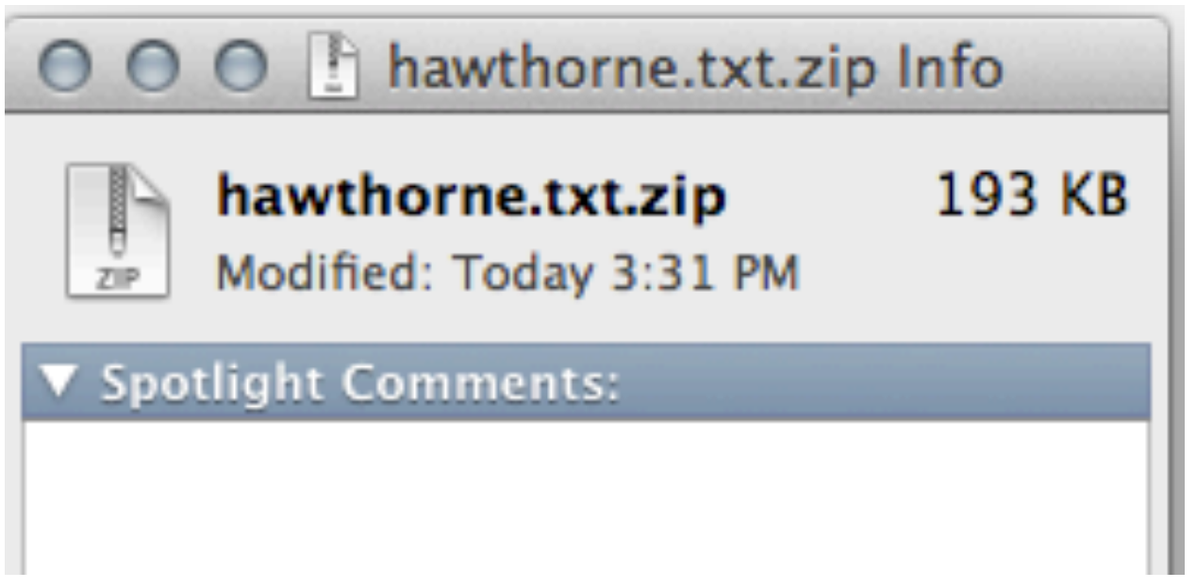
"American Standard Code for Information Interchange", I've just learned.

Text data (ASCII)

8 bits per character (one *byte*).



← *K* for kilo
B for byte
≈ 508,928 characters



← *But how?*

"American Standard Code for Information Interchange", I've just learned.

(Lossless) data compression

8 bits per character (one *byte*).

```
a : 29380        j : 282          s : 24265
b : 5672         k : 2351         t : 36509
c : 9571         l : 16336        u : 9861
d : 15972        m : 10312        v : 3680
e : 50853        n : 25435        w : 8839
f : 9317         o : 27513        x : 461
g : 7091         p : 6840         y : 6534
h : 26624        q : 337          z : 152
i : 26445        r : 24075
```

From hawthorne.txt (from Markov)

(Lossless) data compression

8 bits per character (one *byte*).

```
a : 29380
b : 5672
c : 9571
d : 15972
e : 50853
f : 9317
g : 7091
h : 26624
i : 26445
j : 282
k : 2351
l : 16336
m : 10312
n : 25435
o : 27513
p : 6840
q : 337
r : 24075
s : 24265
t : 36509
u : 9861
v : 3680
w : 8839
x : 461
y : 6534
z : 152
```

What if we use *variable-length* encoding?

http://goo.gl/8FQbY

From hawthorne.txt (from Markov)

Encoding Design

AACCCAAAABAADAE

```
a : 8
c : 3
b : 1
d : 1
e : 1
```

Encoding Design

AACCCAAAABAADAE

```
a : 8
c : 3
b : 1
d : 1
e : 1
```

http://goo.gl/oPVkR

Encoding Design

AACCCAAAABAADAE

```
a : 8
c : 3
b : 1
d : 1
e : 1
```

http://goo.gl/oPVkR

http://goo.gl/vBffI

Prefix Codes

AACCCAAABAADAE

```
a : 8
c : 3
b : 1
d : 1
e : 1
```

No encoding can be a *prefix* of any other: if 'a' is 01, nothing else can start with 01.

A valid (but inefficient!) prefix code for this string:

```
a : 01
c : 100
b : 101
d : 110
e : 111
```

(Gets us to 34 bits)

```
01011001001000101011010111001111
```

Prefix Codes

AACCCAAABAADAE

```
a : 8
c : 3
b : 1
d : 1
e : 1
```

No encoding can be a *prefix* of any other: if 'a' is 01, nothing else can start with 01.

A valid (but inefficient!) prefix code for this string:

```
a : 01
c : 100
b : 101
d : 110
e : 111
```

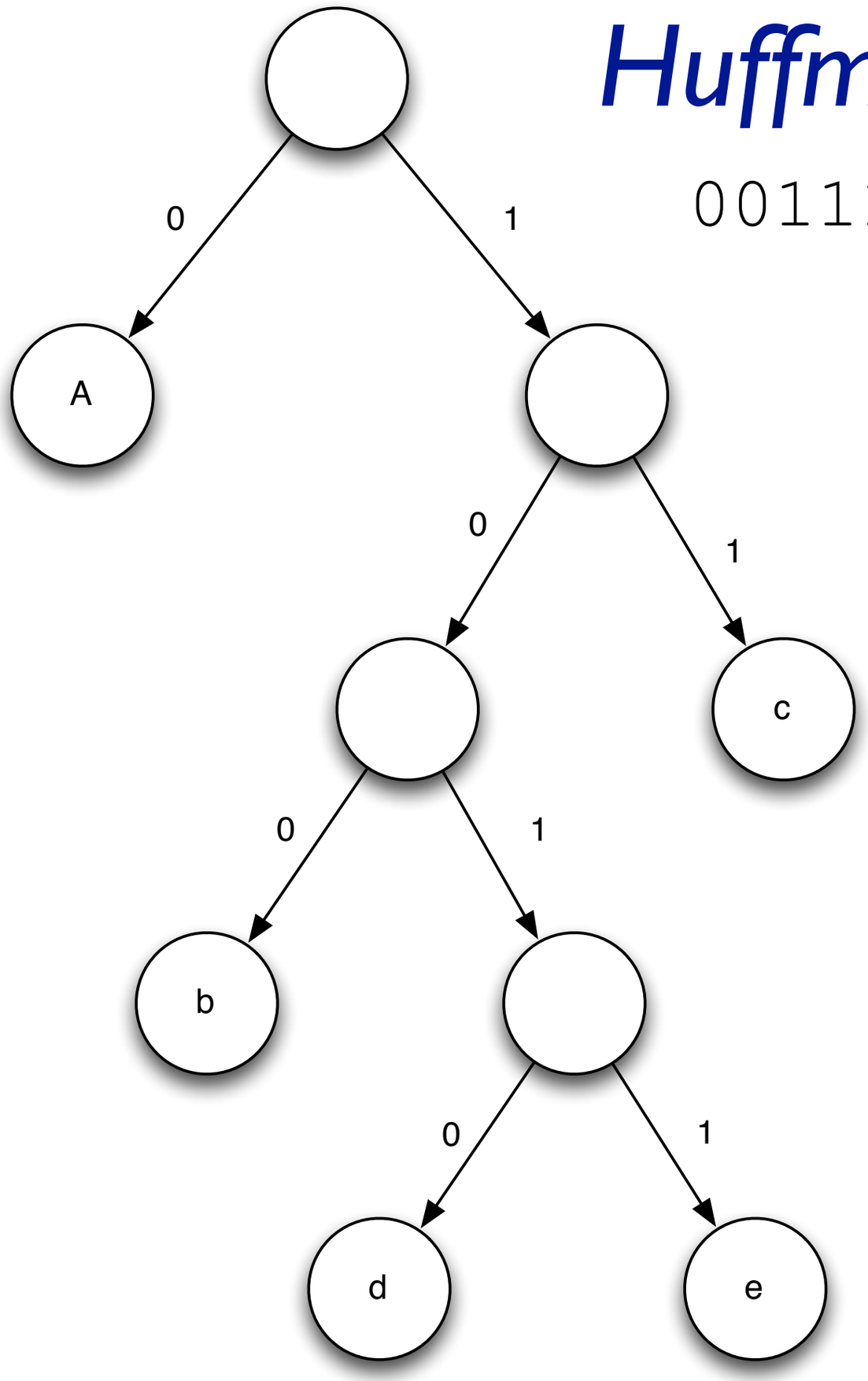
(Gets us to 34 bits)

http://goo.gl/KBk9j

01011001001000101011010111001111

Huffman Coding

00111 11100 01000 01010 01011



Sometimes, it seems like everything is a tree. That's because it's pretty much true.

Huffman Coding

she sells sea shells by the sea shore

- Generate your *frequencies*.
- Make *k* trees, one for each character.
- while (more than one tree)
- remove the two smallest trees *cough cough priority queue*
- merge them under a new root
- add the new tree

The per-character encoding is the path from root to leaf.

You use the tree to build the encoding, and then to decode data. For encoding, just make a map.

What if it isn't text?

Optimality

Optimal for per-character encoding schemes.

In practice...trickier.

File format

File starts
Magic Number
Frequency data *(or huff tree!)*
Compressed data
Pseudo-EOF
File ends