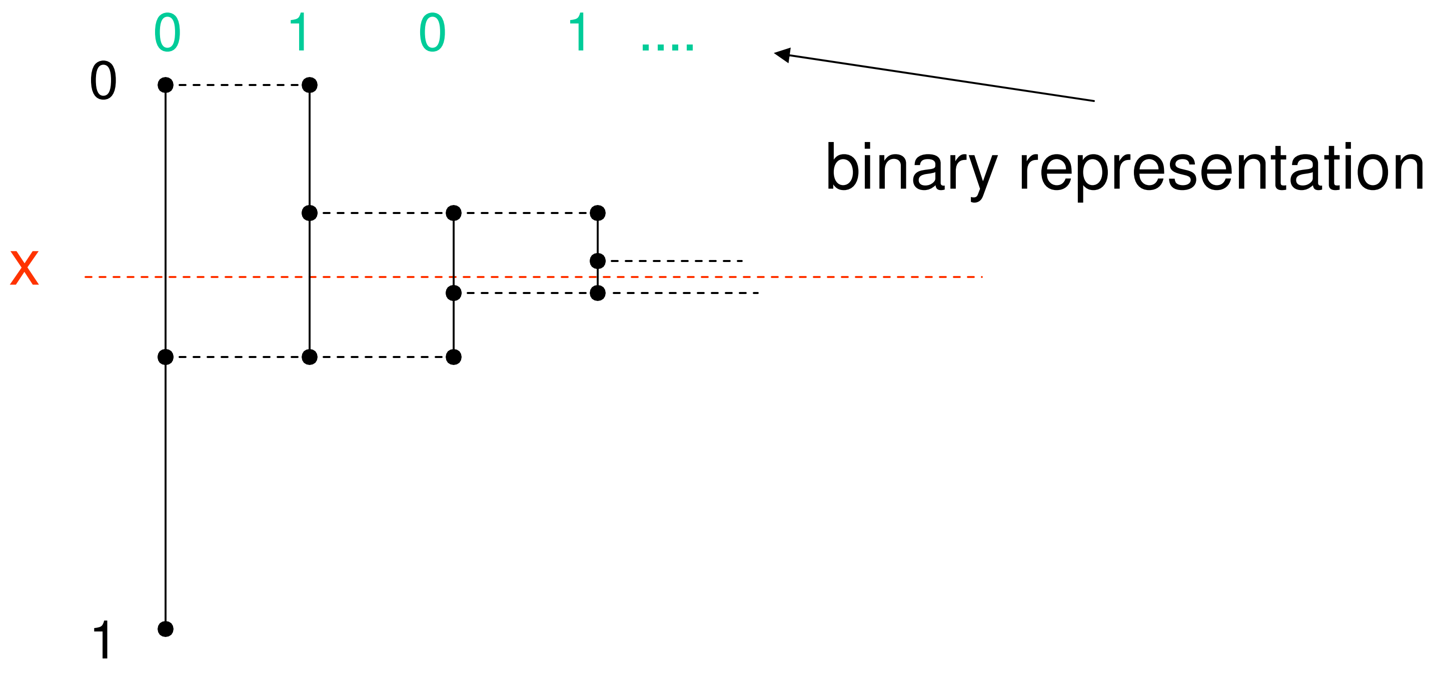
CSEP 590
Data Compression

Autumn 2007

Arithmetic Coding

Reals in Binary

- Any real number x in the interval [0,1) can be represented in binary as $.b_1b_2...$ where b_i is a bit.



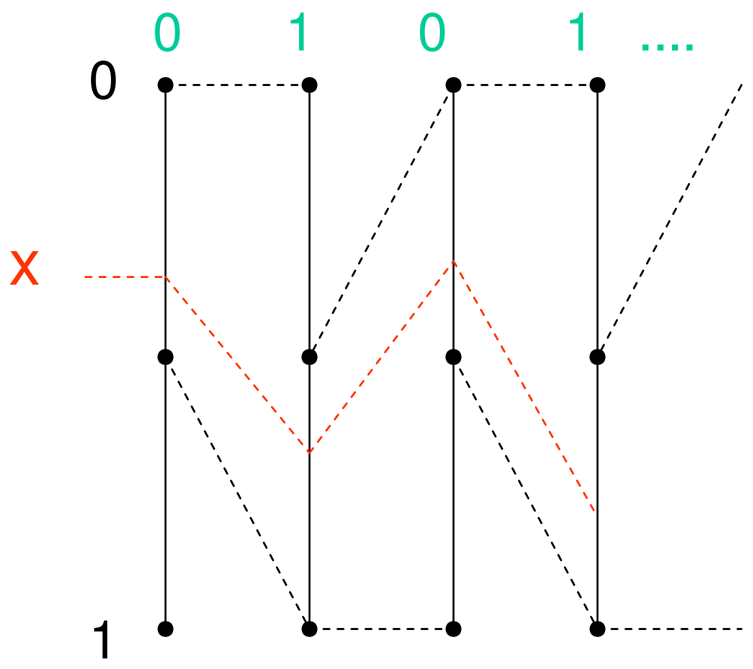
First Conversion

```
L := 0; R :=1; i := 1
while x > L *
    if $x < (L+R)/2$ then $b_i$ := 0 ; R := (L+R)/2;
    if $x \geq (L+R)/2$ then $b_i$ := 1 ; L := (L+R)/2;
    i := i + 1
end{while}
$b_j$ := 0 for all $j \geq i$
```

* Invariant: x is always in the interval [L,R)

Conversion using Scaling

- Always scale the interval to unit size, but x must be changed as part of the scaling.



Binary Conversion with Scaling

```
$y := x$; $i := 0$
while $y > 0$ *
    $i := i + 1$;
    if $y < 1/2$ then $b_i := 0$; $y := 2y$;
    if $y \geq 1/2$ then $b_i := 1$; $y := 2y - 1$;
end{while}
$b_j := 0$ for all $j \geq i + 1$
```

* Invariant: $x = .b_1b_2 \ldots b_i + y/2^i$

Proof of the Invariant

- Initially $x = 0 + y/2^0$
- Assume $x = .b_1b_2 \ldots b_i + y/2^i$
  - Case 1. $y < 1/2$. $b_{i+1} = 0$ and $y' = 2y$

    $$\begin{aligned} .b_1b_2 \ldots b_i b_{i+1} + y'/2^{i+1} &= .b_1b_2 \ldots b_i 0 + 2y/2^{i+1} \\ &= .b_1b_2 \ldots b_i + y/2^i \\ &= x \end{aligned}$$

  - Case 2. $y \geq 1/2$. $b_{i+1} = 1$ and $y' = 2y - 1$

    $$\begin{aligned} .b_1b_2 \ldots b_i b_{i+1} + y'/2^{i+1} &= .b_1b_2 \ldots b_i 1 + (2y-1)/2^{i+1} \\ &= .b_1b_2 \ldots b_i + 1/2^{i+1} + 2y/2^{i+1} - 1/2^{i+1} \\ &= .b_1b_2 \ldots b_i + y/2^i \\ &= x \end{aligned}$$

Example and Exercise

x = 1/3

y	i	b
1/3	1	0
2/3	2	1
1/3	3	0
2/3	4	1
...	...	...

x = 17/27

y	i	b
17/27	1	1

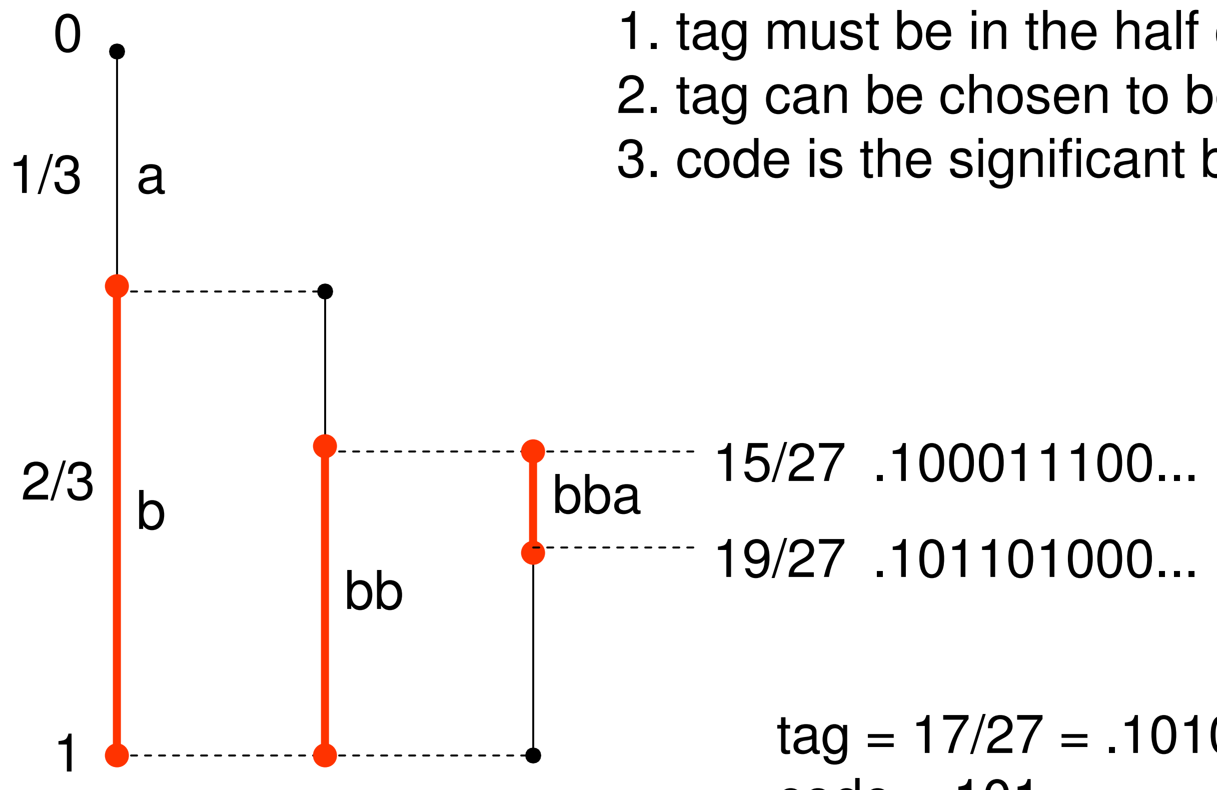
Arithmetic Coding

Basic idea in arithmetic coding:

- represent each string x of length n by a unique interval [L,R) in [0,1).
- The width R-L of the interval [L,R) represents the probability of x occurring.
- The interval [L,R) can itself be represented by any number, called a tag, within the half open interval.
- The k significant bits of the tag $.t_1t_2t_3...$ is the code of x. That is, . $.t_1t_2t_3...t_k000...$ is in the interval [L,R).
  - It turns out that $k \approx \log_2(1/(R-L))$.

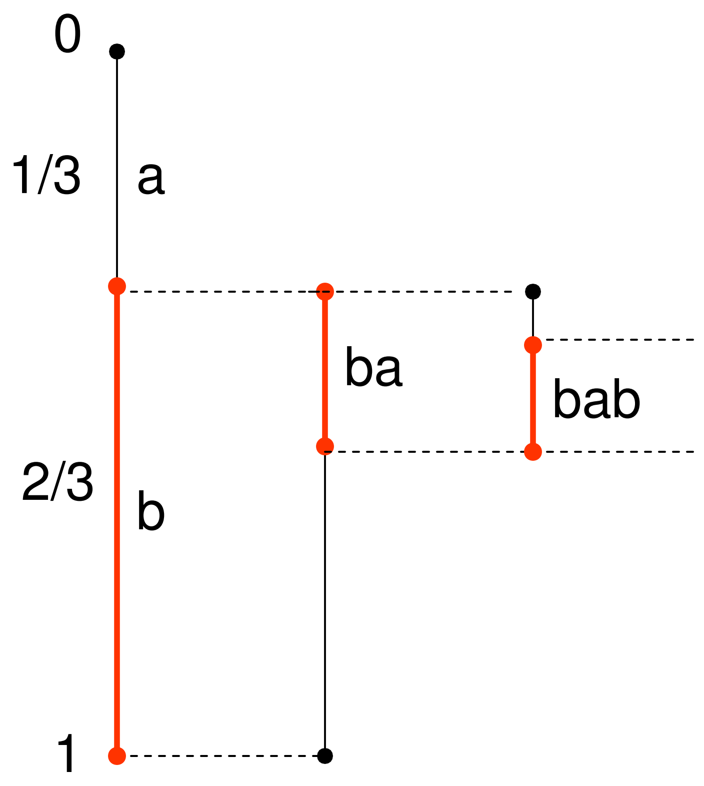
Example of Arithmetic Coding (1)

1. tag must be in the half open interval.
2. tag can be chosen to be (L+R)/2.
3. code is the significant bits of the tag.



tag = 17/27 = .101000010...
code = 101

Some Tags are Better than Others



11/27 .011010000...

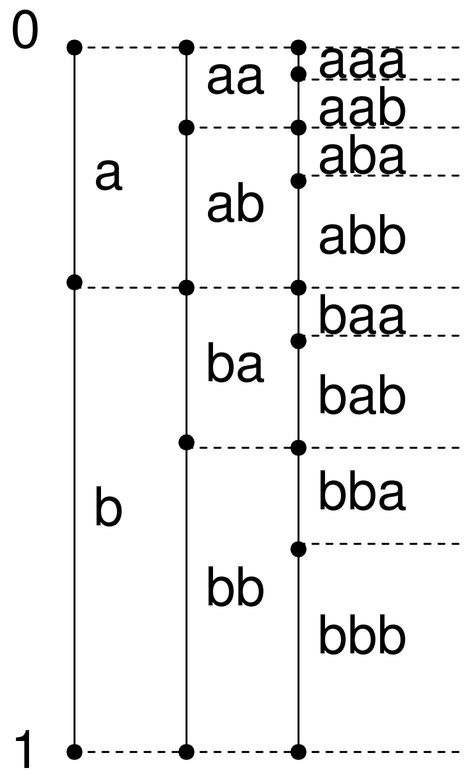
15/27 .100011100...

Using tag = (L+R)/2
tag = 13/27 = .011110110...
code = 0111

Alternative tag = 14/27 = .100001001...
code = 1

Example of Codes

- P(a) = 1/3, P(b) = 2/3.



0/27	.000000000...
1/27	.000010010...
3/27	.000111000...
5/27	.001011110...
9/27	.010101010...
11/27	.011010000...
15/27	.100011100...
19/27	.101101000...
27/27	.111111111...

tag = (L+R)/2	code	
.000001001...	0	aaa
.000100110...	0001	aab
.001001100...	001	aba
.010000101...	01	abb
.010111110...	01011	baa
.011110111...	0111	bab
.101000010...	101	bba
.110110100...	11	bbb

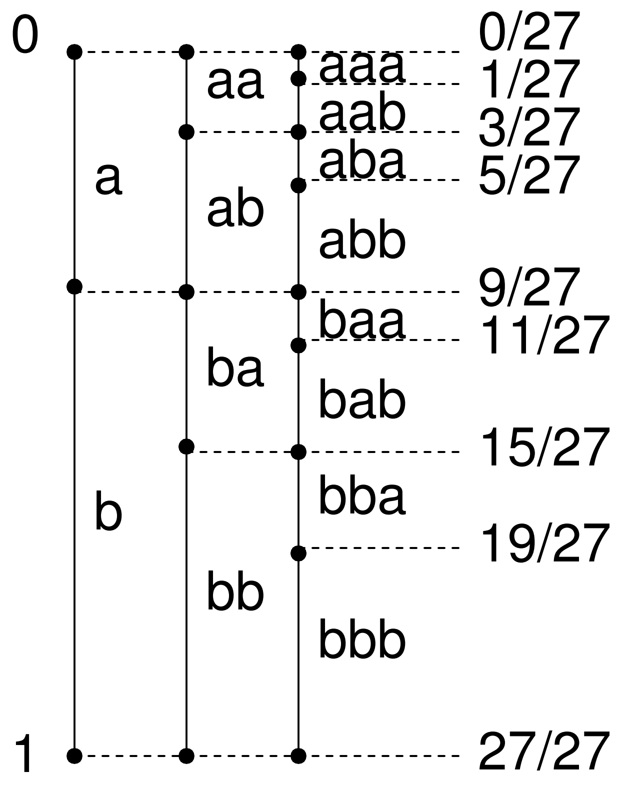
.95 bits/symbol
.92 entropy lower bound

Code Generation from Tag

- If binary tag is $.t_1t_2t_3... = (L+R)/2$ in $[L,R)$ then we want to choose k to form the code $t_1t_2...t_k$.
- Short code:
  - choose k to be as small as possible so that $L \leq .t_1t_2...t_k000... < R$.
- Guaranteed code:
  - choose $k = \lceil \log_2 (1/(R-L)) \rceil + 1$
  - $L \leq .t_1t_2...t_kb_1b_2b_3... < R$ for any bits $b_1b_2b_3...$
  - for fixed length strings provides a good prefix code.
  - example: [.000000000..., .000010010...), tag = .000001001...
    Short code: 0
    Guaranteed code: 000001

Guaranteed Code Example

- P(a) = 1/3, P(b) = 2/3.



tag = (L+R)/2	short code	Prefix code	
.000001001...	0	0000	aaa
.000100110...	0001	0001	aab
.001001100...	001	001	aba
.010000101...	01	0100	abb
.010111110...	01011	01011	baa
.011110111...	0111	0111	bab
.101000010...	101	101	bba
.110110100...	11	11	bbb

Arithmetic Coding Algorithm

- $P(a_1), P(a_2), \ldots, P(a_m)$
- $C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1})$
- Encode $x_1x_2...x_n$

```
Initialize L := 0 and R:= 1;
for i = 1 to n do
   W := R - L;
    L := L + W * C(x_i);
    R := L + W * P(x_i);
t := (L+R)/2;
choose code for the tag
```

Arithmetic Coding Example

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- abca

```
W := R - L;
L := L + W C(x);
R := L + W P(x)
```

symbol	W	L	R
		0	1
a	1	0	1/4
b	1/4	1/16	3/16
c	1/8	5/32	6/32
a	1/32	5/32	21/128

tag = (5/32 + 21/128)/2 = 41/256 = .001010010...
L = .001010000...
R = .001010100...
code = 00101
prefix code = 00101001

Arithmetic Coding Exercise

- $P(a) = 1/4$, $P(b) = 1/2$, $P(c) = 1/4$
- $C(a) = 0$, $C(b) = 1/4$, $C(c) = 3/4$
- bbbb

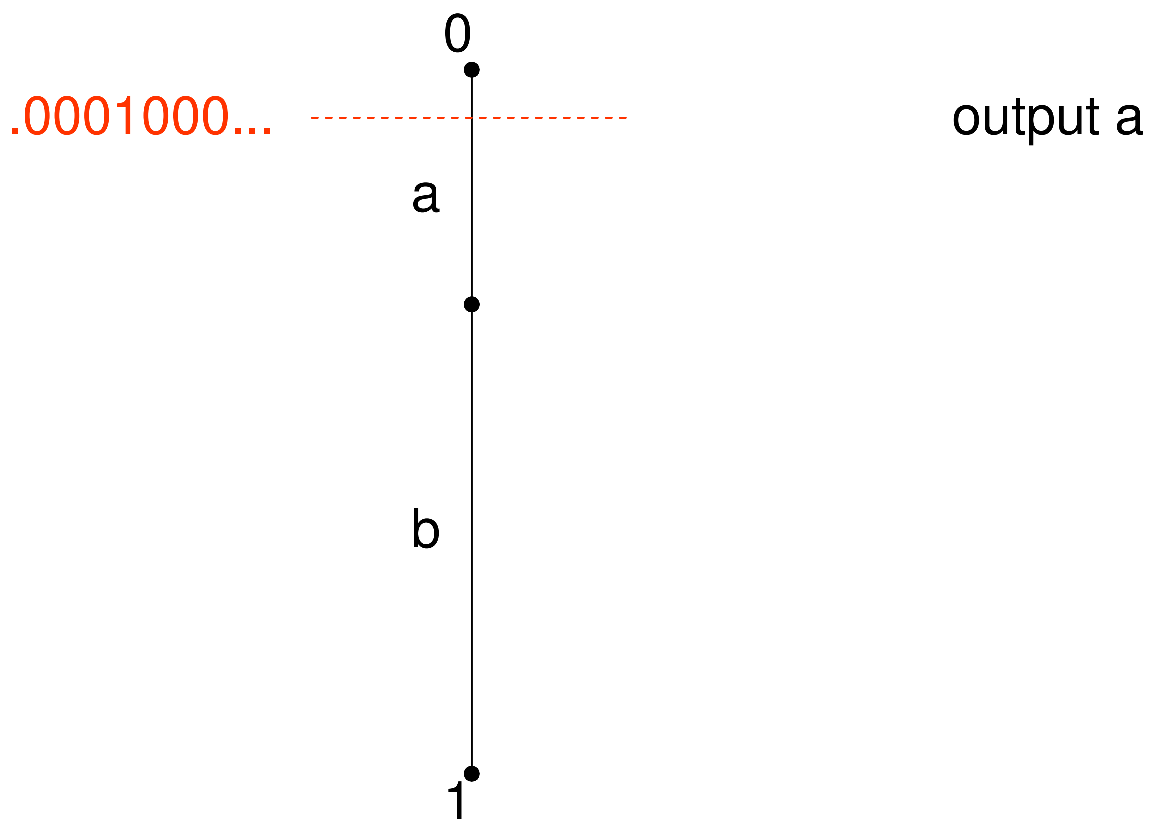
```
W := R - L;
L := L + W C(x);
R := L + W P(x)
```

symbol	W	L	R
		0	1
b	1		
b			
b			
b			

tag =
L =
R =
code =
prefix code =

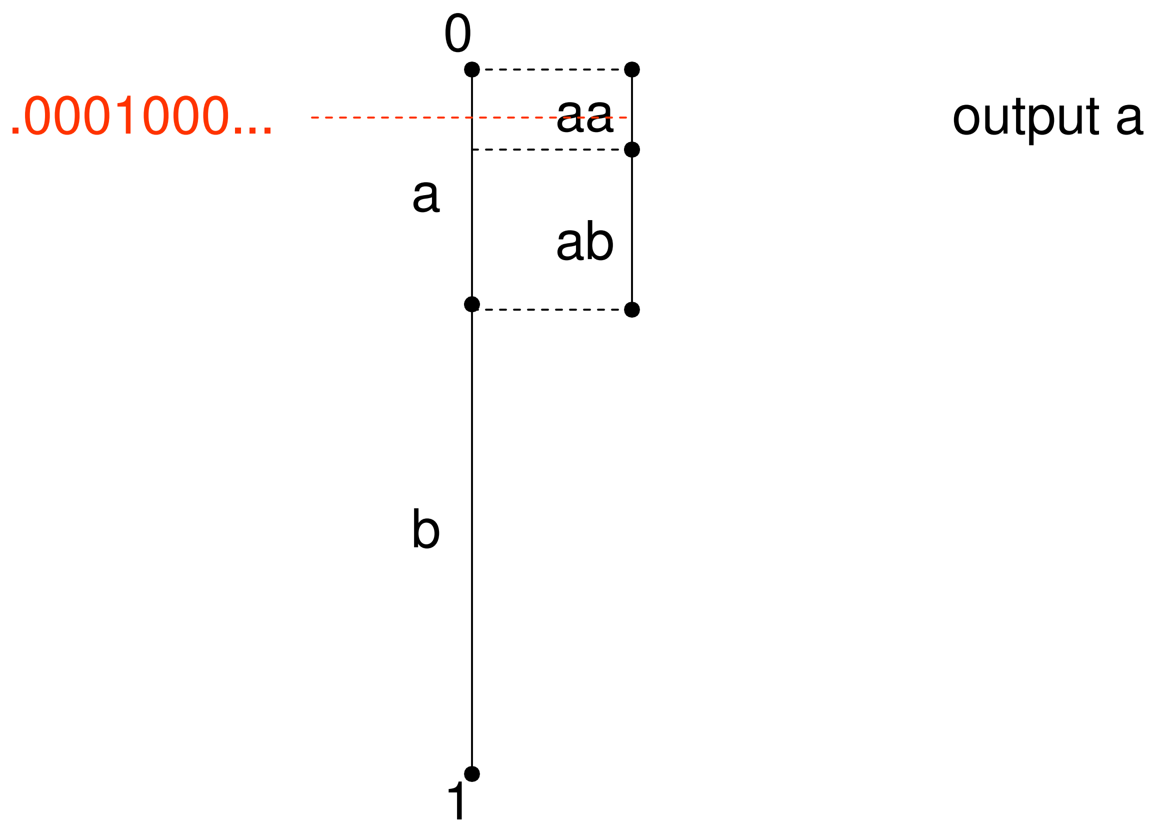
Decoding (1)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...



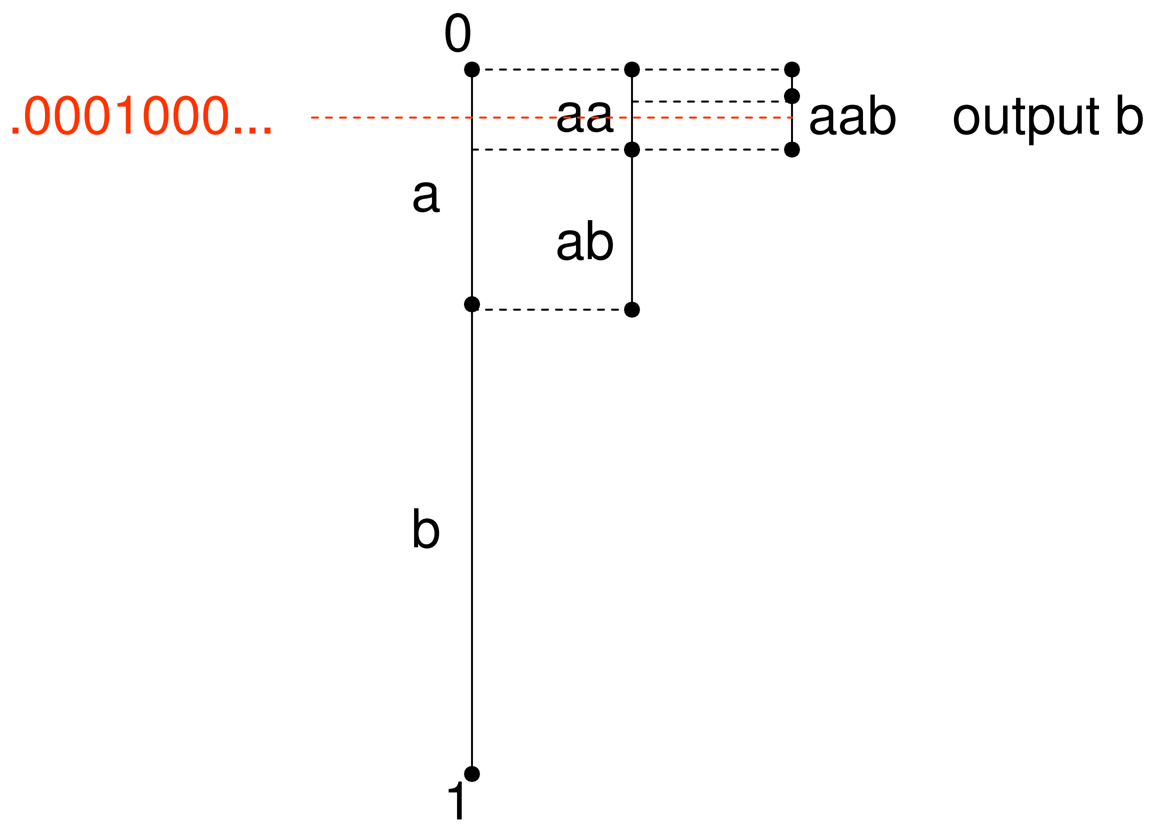
Decoding (2)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...



Decoding (3)

- Assume the length is known to be 3.
- 0001 which converts to the tag .0001000...



Arithmetic Decoding Algorithm

- $P(a_1), P(a_2), \ldots , P(a_m)$
- $C(a_i) = P(a_1) + P(a_2) + \ldots + P(a_{i-1})$
- Decode $b_1b_2...b_k$, number of symbols is n.

```
Initialize L := 0 and R := 1;
t := $.b_1b_2...b_k000...$
for i = 1 to n do
   W := R - L;
    find j such that $L + W * C(a_j) \leq t < L + W * (C(a_j)+P(a_j))$
    output $a_j$;
    L := $L + W * C(a_j)$;
    R := $L + W * P(a_j)$;
```

Decoding Example

- P(a) = 1/4, P(b) = 1/2, P(c) = 1/4
- C(a) = 0, C(b) = 1/4, C(c) = 3/4
- 00101

tag = .00101000... = 5/32

W	L	R	output
	0	1	
1	0	1/4	a
1/4	1/16	3/16	b
1/8	5/32	6/32	c
1/32	5/32	21/128	a

Decoding Issues

- There are at least two ways for the decoder to know when to stop decoding.

1. Transmit the length of the string
2. Transmit a unique end of string symbol

Practical Arithmetic Coding

- Scaling:
  - By scaling we can keep L and R in a reasonable range of values so that W = R - L does not underflow.
  - The code can be produced progressively, not at the end.
  - Complicates decoding some.
- Integer arithmetic coding avoids floating point altogether.

More Issues

- Context
- Adaptive
- Comparison with Huffman coding

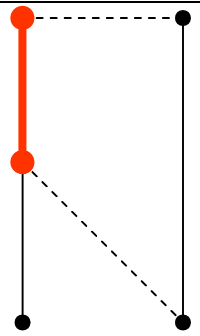
Scaling

- Scaling:
  - By scaling we can keep L and R in a reasonable range of values so that W = R – L does not underflow.
  - The code can be produced progressively, not at the end.
  - Complicates decoding some.

Scaling during Encoding

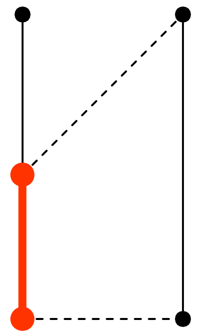
Lower half

```
If  [L,R) is contained in [0,.5) then
    L := 2L; R := 2R
    output 0, followed by  C 1's
    C := 0.
```



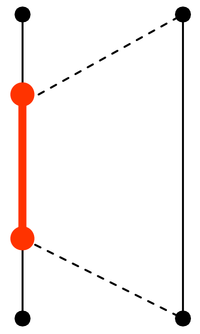
Upper half

```
If [L,R) is contained in  [.5,1) then
   L := 2L –1, R := 2R - 1
   output 1, followed by C 0's
   C := 0
```



Middle Half

```
If [L,R) is contained in  [.25,.75)  then
   L := 2L –.5, R := 2R -.5
   C := C + 1.
```

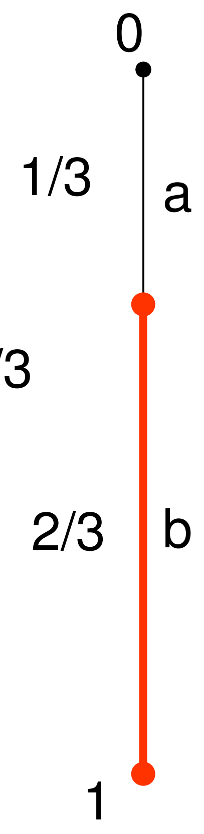


Example

- <u>b</u>aa

C = 0

L = 1/3 R = 3/3

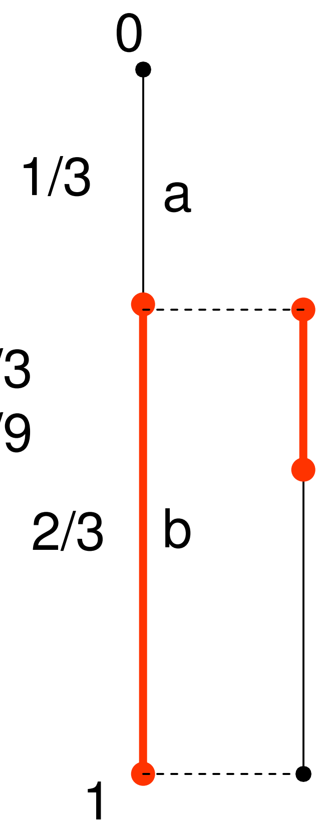


Example

- b<u>a</u>a

C = 0

L = 1/3 R = 3/3
L = 3/9 R = 5/9



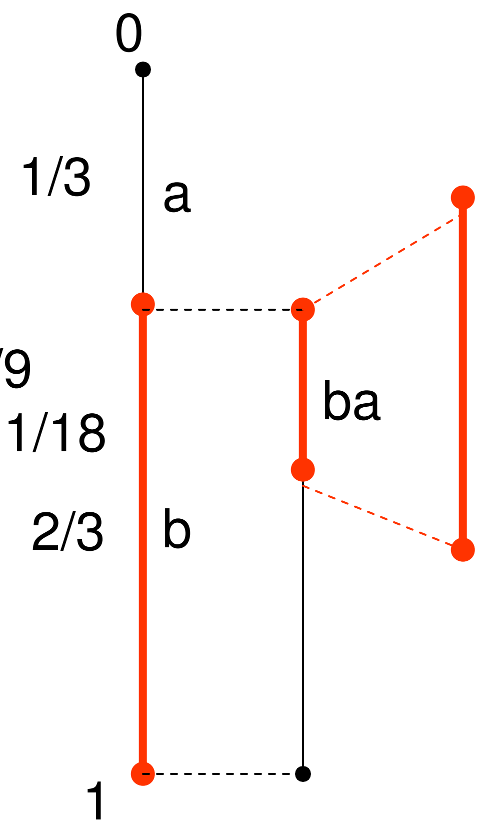
Scale middle half

Example

- b<u>a</u>a

C = 1

L = 3/9 R = 5/9
L = 3/18 R = 11/18

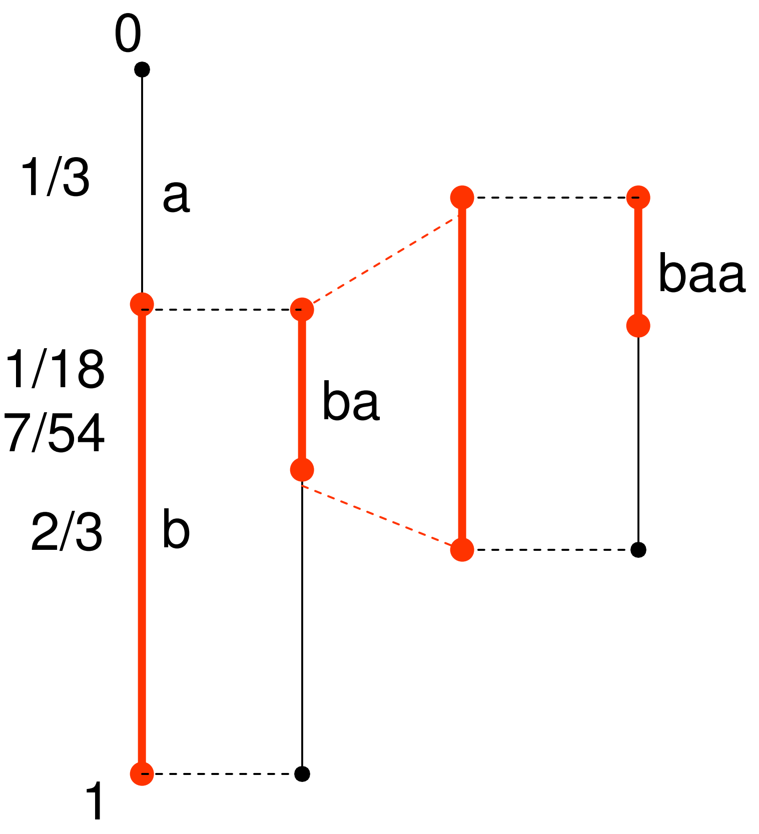


Example

- ba<u>a</u>

C = 1

L = 3/18 R = 11/18
L = 9/54 R = 17/54



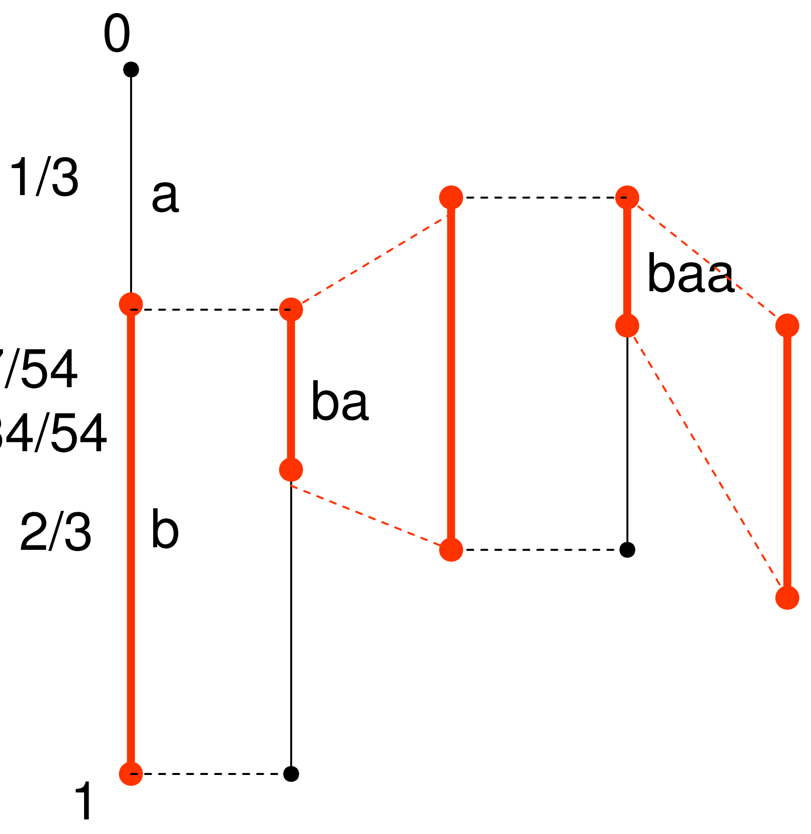
Scale lower half

Example

- baa 01

C = 0

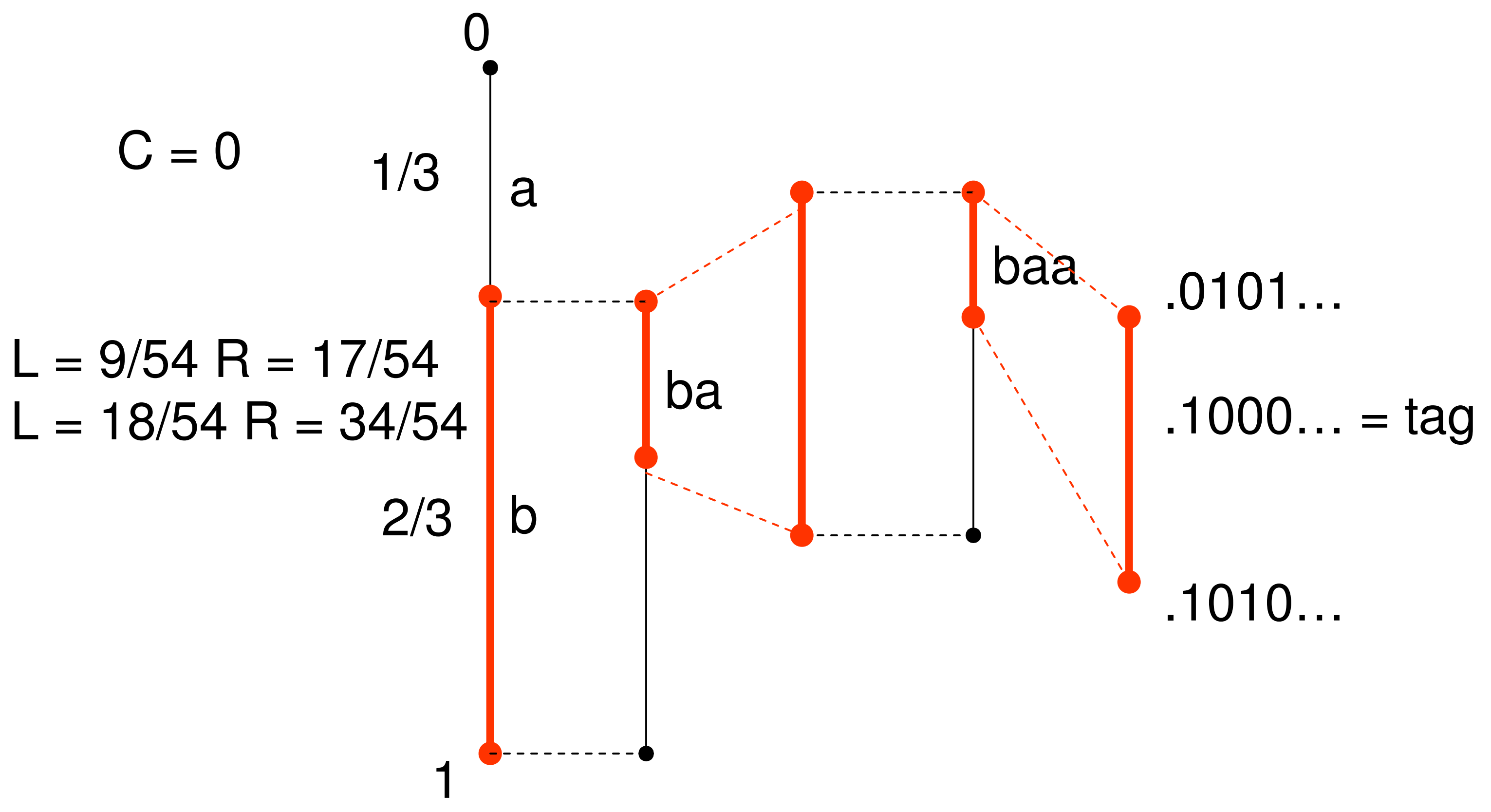
L = 9/54 R = 17/54
L = 18/54 R = 34/54



Example

- baa 01$\underline{1}$

In end L < ½ < R, choose tag to be 1/2



Exercise

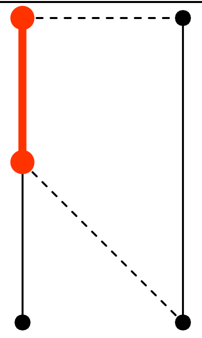
Model: a: 1/4; b: 3/4
Encode: bba

Decoding

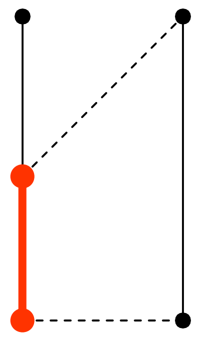
- The decoder behaves just like the encoder except that C does not need to be maintained.
- Instead, the input stream is consumed during scaling.

Scaling during Decoding

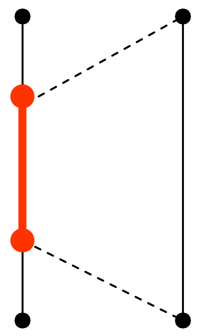
Lower half
If [L,R) is contained in [0,.5) then
    L := 2L; R := 2R
    consume 0 from the encoded stream



Upper half
If [L,R) is contained in [.5,1) then
   L := 2L –1, R := 2R - 1
   consume 1 from the encoded stream



Middle half
If [L,R) is contained in [.25,.75) then
   L := 2L –.5, R := 2R -.5
   Replace 01 with 0 on stream
   Replace 10 with 1 on stream



Scaling Math for the Tag

- Lower Half
  - $.0b_1b_2\ldots \times 10 = .b_1b_2$
- Upper Half
  - $.1b_1b_2\ldots \times 10 - 1 = .b_1b_2$
- Middle Half
  - $.01b_2b_3\ldots \times 10 - .1 = .0b_2b_3$
  - $.10b_2b_3\ldots \times 10 - .1 = .1b_2b_3$

Exercise

Model: a: 1/4; b: 3/4
Decode: 001 to 3 symbols

Integer Implementation

- m bit integers
  - Represent 0 with 000…0 (m times)
  - Represent 1 with 111…1 (m times)
- Probabilities represented by frequencies
  - n_i is the number of times that symbol a_i occurs
  - $C_i = n_1 + n_2 + \ldots + n_{i-1}$
  - $N = n_1 + n_2 + \ldots + n_m$

$$W := R - L + 1$$

$$L' := L + \left\lfloor \frac{W \cdot C_i}{N} \right\rfloor$$

$$R := L + \left\lfloor \frac{W \cdot C_{i+1}}{N} \right\rfloor - 1$$

$$L := L'$$

Coding the i-th symbol using integer calculations.
Must use scaling!

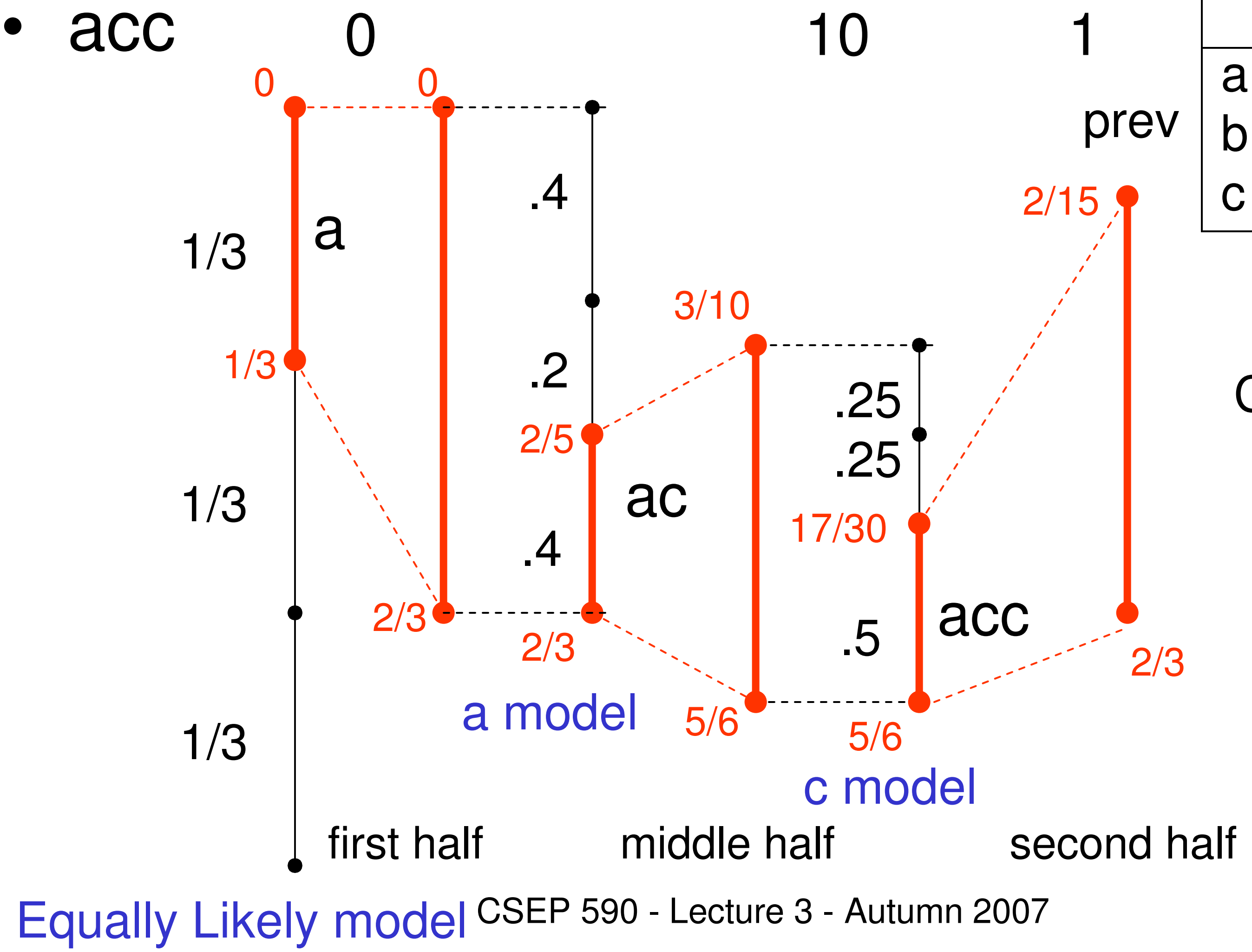
Context

- Consider 1 symbol context.
- Example: 3 contexts.

		next		
		a	b	c
prev	a	.4	.2	.4
	b	.1	.8	.1
	c	.25	.25	.5

Example with Scaling

- acc



next

prev	a	b	c
a	.4	.2	.4
b	.1	.8	.1
c	.25	.25	.5

Code = 0101

Arithmetic Coding with Context

- Maintain the probabilities for each context.
- For the first symbol use the equal probability model
- For each successive symbol use the model for the previous symbol.

Adaptation

- Simple solution – Equally Probable Model.
  - Initially all symbols have frequency 1.
  - After symbol x is coded, increment its frequency by 1
  - Use the new model for coding the next symbol
- Example in alphabet a,b,c,d

		a	a	b	a	a	c
a	1	2	3	3	4	5	5
b	1	1	1	2	2	2	2
c	1	1	1	1	1	1	2
d	1	1	1	1	1	1	1

After aabaac is encoded
The probability model is

a 5/10	b 2/10
c 2/10	d 1/10

Zero Frequency Problem

- How do we weight symbols that have not occurred yet.
  - Equal weights? Not so good with many symbols
  - Escape symbol, but what should its weight be?
  - When a new symbol is encountered send the <esc>, followed by the symbol in the equally probable model. (Both encoded arithmetically.)

		a	a	b	a	a	c
a	0	1	2	2	3	4	4
b	0	0	0	1	1	1	1
c	0	0	0	0	0	0	1
d	0	0	0	0	0	0	0
<esc>	1	1	1	1	1	1	1

After aabaac is encoded
The probability model is

a 4/7	b 1/7
c 1/7	d 0
<esc> 1/7	

PPM

- Prediction with Partial Matching
  - Cleary and Witten (1984)
- State of the art arithmetic coder
  - Arbitrary order context
  - Adaptive
- Needs good data structures to be efficient.

PPM Example

- abracadabra

0-order context		
	a	3
	b	1
	r	1
	c	1
	<esc>	1

1st-order context		
a		
	b	1
	c	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1

PPM Example

- abracad**d**abra

0-order context	
a	3
b	1
r	1
c	1
<esc>	1

Output
1-order <esc>
0-order <esc>
(-1)-order d.
Update tables

1st-order context		
a		
	b	1
	c	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1

- abraca**d**abra

0-order context		
	a	3
	b	1
	r	1
	c	1
	d	1
	<esc>	1

1st-order context		
a		
	b	1
	c	1
	d	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1
ca		
	d	1
	<esc>	1

- <u>abracad**a**</u>bra

0-order context	
a	3
b	1
r	1
c	1
d	1
<esc>	1

0-order d
Update tables

1st-order context		
a		
	b	1
	c	1
	d	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1
ca		
	d	1
	<esc>	1

- abracad**a**bra

0-order context		
	a	4
	b	1
	r	1
	c	1
	d	1
	<esc>	1

1st-order context		
a		
	b	1
	c	1
	d	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1
d		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1
ca		
	d	1
	<esc>	1
ad		
	a	1
	<esc>	1

- <u>abracada**b**</u>ra

0-order context	
a	4
b	1
r	1
c	1
d	1
<esc>	1

1st order b in
context a
Update tables

1st-order context		
a		
	b	1
	c	1
	d	1
	<esc>	1
b		
	r	1
	<esc>	1
r		
	a	1
	<esc>	1
c		
	a	1
	<esc>	1
d		
	a	1
	<esc>	1

2nd-order context		
ab		
	r	1
	<esc>	1
br		
	a	1
	<esc>	1
ra		
	c	1
	<esc>	1
ac		
	a	1
	<esc>	1
ca		
	d	1
	<esc>	1
ad		
	a	1
	<esc>	1

Arithmetic vs. Huffman

- Both compress very well. For m symbol grouping.
  - Huffman is within 1/m of entropy.
  - Arithmetic is within 2/m of entropy.
- Context
  - Huffman needs a tree for every context.
  - Arithmetic needs a small table of frequencies for every context.
- Adaptation
  - Huffman has an elaborate adaptive algorithm
  - Arithmetic has a simple adaptive mechanism.
- Bottom Line – Arithmetic is more flexible than Huffman.