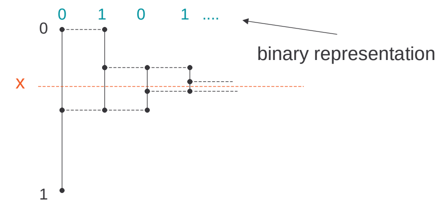
CSE 490 GZ
Introduction to Data Compression
Winter 2004

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.



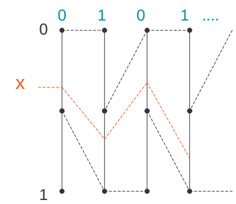
First Conversion

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

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

Proof of the Invariant

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

    $.b_1b_2 ... b_i b_{i+1} + y'/2^{i+1} = .b_1b_2 ... b_i 0 + 2y/2^{i+1}$
    $= .b_1b_2 ... b_i + y/2^i$
    $= x$
  - Case 2. $y \geq 1/2$. $b_{i+1} = 1$ and $y' = 2y – 1$

    $.b_1b_2 ... b_i b_{i+1} + y'/2^{i+1} = .b_1b_2 ... b_i 1 + (2y-1)/2^{i+1}$
    $= .b_1b_2 ... b_i + 1/2^{i+1} + 2y/2^{i+1} - 1/2^{i+1}$
    $= .b_1b_2 ... b_i + y/2^i$
    $= x$

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

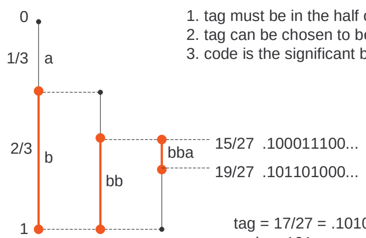
Arithmetic Coding

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 $k \approx -\log_2(1/(R-L))$.

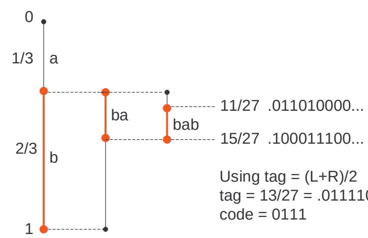
Example of Arithmetic Coding (1)

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

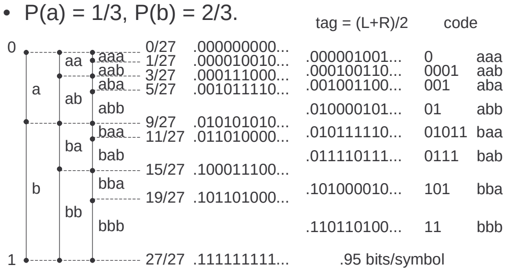


Using tag = (L+R)/2
tag = 13/27 = .011110110...
code = 0111

Alternative tag = 14/37 = .100001001...
code = 1

Example of Codes

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



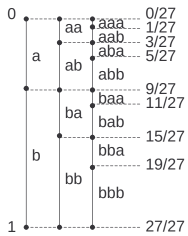
.95 bits/symbol
.92 entropy lower bound

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 $L \le .t_1t_2...t_k000... < R$.
- Guaranteed code:
  - choose $k = \lceil \log_2 (1/(R-L)) \rceil + 1$
  - $L \le .t_1t_2...t_kb_1b_2b_3... < R$ 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 $x_1x_2 \ldots x_n$

```
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

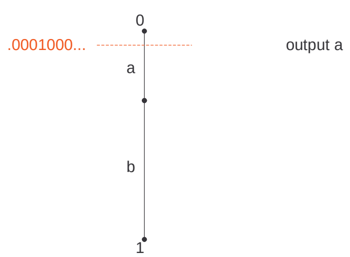
W := R - L;
L := L + W C(x);
R := L + W P(x)

symbol	W	L	R
		0	1
b	1		
b			
b			
b			

tag =
L =
R =
code =
prefix code =

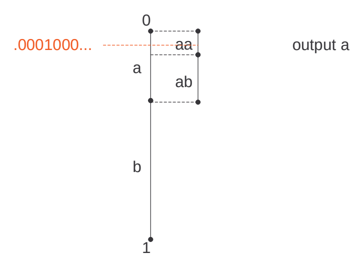
Decoding (1)

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



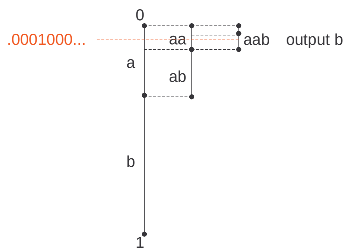
Decoding (2)

- 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 $b_1b_2...b_m$, number of symbols is n.

```
Initialize L := 0 and R := 1;
t := .b1b2...bm000...
for i = 1 to n do
   W := R - L;
    find j such that L + W * C(aj) ≤ t < L + W * (C(aj)+P(aj))
    output aj;
    L := L + W * C(aj);
    R := L + W * P(aj);
```


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