CSE 589
Applied Algorithms
Autumn 2001

Golomb Coding
Arithmetic Coding
LZW
Sequitur

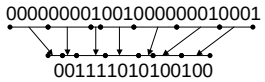
Binary Golomb Codes

- Binary source with 0's much more frequent than 1's.
- Variable-to-variable length code
- Prefix code.
- Golomb code of order 4

input	output
0000	0
0001	100
001	101
01	110
1	111

Example

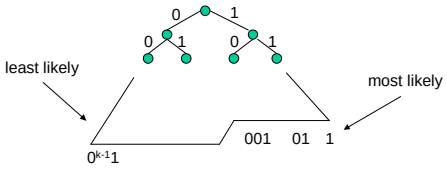
input	output
0000	0
0001	100
001	101
01	110
1	111



Constructing a Binary Golomb Code

- Let p be the probability of 0
- Choose the order k such that

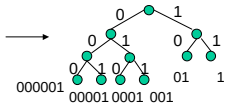
$$p^{k+1} \le 1 - p^k < p^{k-1}$$



Example

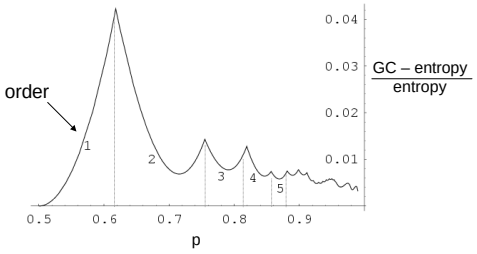
p = .89
p^7 = .442
1- p^6 = .503
p^5 = .558

Order = 6



input	output
000000	0
000001	1000
00001	1001
0001	1010
001	1011
01	110
1	111

Comparison of GC with Entropy



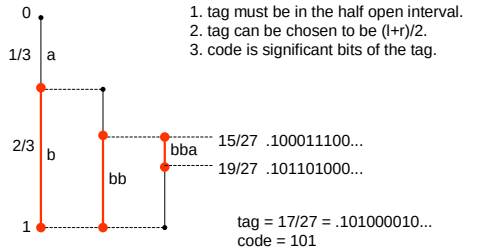
Notes on Binary Golomb Code

- Very effective as a binary run-length code
- As p goes to 1 effectiveness near entropy
- There are adaptive Golomb codes, so that the order can change according to observed frequency of 0's.

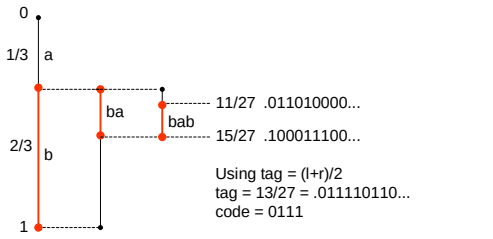
Arithmetic Coding

- Huffman coding works well for larger alphabets and gets to within one bit of the entropy lower bound. Can we do better. Yes
- Basic idea in arithmetic coding:
  - represent each string x of length n by an interval [l,r) in [0,1). We assume n is known.
  - 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).

Example of Arithmetic Coding (1)

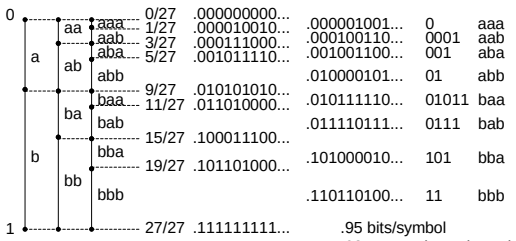


Some Tags are Better than Others



Example of Codes

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

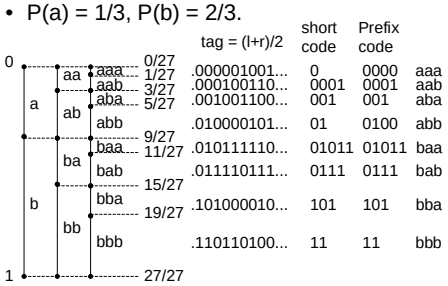


Code Generation from Tag

- If binary tag is $.t_1t_2t_3...$ = (r+l)/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 $l \le .t_1t_2...t_k000... < r$.
  - Good if n is known.
- Guaranteed code:
  - choose $k = \lceil -\log_2(r-l) \rceil + 1$
  - $l \le .t_1t_2...t_kb_1b_2b_3... < r$ 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	
aaa	.000001001...	0	0000	aaa
aab	.000100110...	0001	0001	aab
aba	.001001100...	001	001	aba
abb	.010000101...	01	0100	abb
baa	.010111110...	01011	01011	baa
bab	.011110111...	0111	0111	bab
bba	.101000010...	101	101	bba
bbb	.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 $x_1 x_2 ... x_n$

```
Initialize l := 0 and r := 1;
for i = 1 to n do
   w := r - l;
   l := l + w * C(xi);
   r := l + w * P(xi);
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

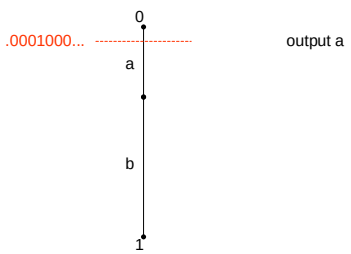
```
w := r - l;
l := l + w C(x);
r := l + w P(x)
```

symbol	w	l	r
		0	1
b	1		
b			
b			
b			

tag =
l =
r =
code =
prefix code =

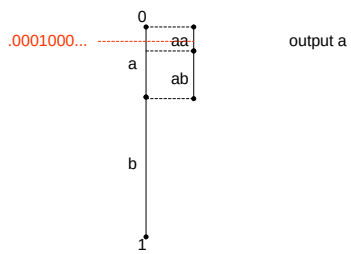
Decoding (1)

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



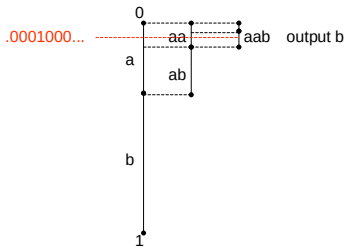
Decoding (2)

- 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 $b_1 b_2 \ldots b_m$, 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

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.

Notes on Arithmetic Coding

- Arithmetic codes come close to the entropy lower bound.
- Grouping symbols is effective for arithmetic coding.
- Arithmetic codes can be used effectively on small symbol sets. Advantage over Huffman.
- Context can be added so that more than one probability distribution can be used.
  - The best coders in the world use this method.
- There are very effective adaptive arithmetic coding methods.

Dictionary Coding

- Does not use statistical knowledge of data.
- Encoder: As the input is processed develop a dictionary and transmit the index of strings found in the dictionary.
- Decoder: As the code is processed reconstruct the dictionary to invert the process of encoding.
- Examples: LZW, LZ77, Unix Compress, gzip, GIF

LZW Encoding Algorithm

Repeat
  find the longest match w in the dictionary
  output the index of w
  put wa in the dictionary where a was the unmatched symbol

LZW Encoding Example (1)

Dictionary

0 a
1 b

a b a b a b a b a

LZW Encoding Example (2)

Dictionary

0 a
1 b
2 ab

a b a b a b a b a
0

LZW Encoding Example (3)

Dictionary

0 a
1 b
2 ab
3 ba

a b a b a b a b a
0 1

LZW Encoding Example (4)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba

a b a b a b a b a
0 1 2

LZW Encoding Example (5)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 abab

a b a b a b a b a
0 1 2 4

LZW Encoding Example (6)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 abab

a b a b a b a b a
0 1 2 4 3

LZW Decoding Algorithm

- Emulate the encoder in building the dictionary. Decoder is slightly behind the encoder.

```
initialize dictionary;
decode first index to w;
put w? in dictionary;
repeat
   decode the first symbol s of the index;
   complete the previous dictionary entry with s;
   finish decoding the remainder of the index;
   put w? in the dictionary where w was just decoded;
```


LZW Decoding Example (1)

Dictionary

0 a
1 b
2 a?

0 1 2 4 3 6
a

LZW Decoding Example (2a)

Dictionary

0 a
1 b
2 ab

0 1 2 4 3 6
a b

LZW Decoding Example (2b)

Dictionary

0 a
1 b
2 ab
3 b?

0 1 2 4 3 6
a b

LZW Decoding Example (3a)

Dictionary

0 a
1 b
2 ab
3 ba

0 1 2 4 3 6
a b a

LZW Decoding Example (3b)

Dictionary

0 a
1 b
2 ab
3 ba
4 ab?

0 1 2 4 3 6
a b ab

LZW Decoding Example (4a)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba

0 1 2 4 3 6
a b ab a

LZW Decoding Example (4b)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 aba?

0 1 2 4 3 6
a b ab aba

LZW Decoding Example (5a)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 abab

0 1 2 4 3 6
a b ab aba b

LZW Decoding Example (5b)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 abab
6 ba?

0 1 2 4 3 6
a b ab aba ba

LZW Decoding Example (6a)

Dictionary

0 a
1 b
2 ab
3 ba
4 aba
5 abab
6 bab

0 1 2 4 3 6
a b ab aba ba b

LZW Decoding Example (6b)

Dictionary

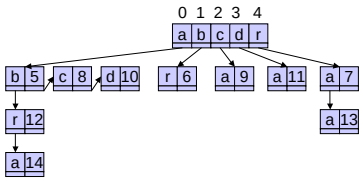
0	a
1	b
2	ab
3	ba
4	aba
5	abab
6	bab
7	bab?

0 1 2 4 3 6
a b ab aba ba bab

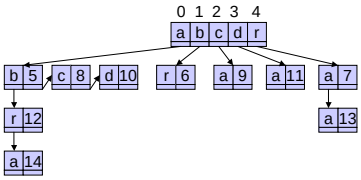
Trie Data Structure for Encoder's Dictionary

- Fredkin (1960)

0	a	9	ca
1	b	10	ad
2	c	11	da
3	d	12	abr
4	r	13	raa
5	ab	14	abra
6	br		
7	ra		
8	ac		

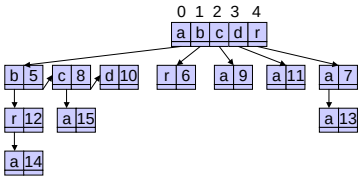


Encoder Uses a Trie (1)



a b r a c a d a b r a a b r a c a d a b r a
0 1 4 0 2 0 3 5 7 12

Encoder Uses a Trie (2)



a b r a c a d a b r a a b r a c a d a b r a
0 1 4 0 2 0 3 5 7 12 8

Decoder's Data Structure

- Simply an array of strings

0	a	9	ca
1	b	10	ad
2	c	11	da
3	d	12	abr
4	r	13	raa
5	ab	14	abr?
6	br		
7	ra		
8	ac		

0 1 4 0 2 0 3 5 7 12 8 ...
a b r a c a d ab ra abr

Notes on Dictionary Coding

- Extremely effective when there are repeated patterns in the data that are widely spread. Where local context is not as significant.
  - text
  - some graphics
  - program sources or binaries
- Variants of LZW are pervasive.

Sequitur

- Nevill-Manning and Witten, 1996.
- Uses a context-free grammar (without recursion) to represent a string.
- The grammar is inferred from the string.
- If there is structure and repetition in the string then the grammar may be very small compared to the original string.
- Clever encoding of the grammar yields impressive compression ratios.
- Compression plus structure!

Context-Free Grammars

- Invented by Chomsky in 1959 to explain the grammar of natural languages.
- Also invented by Backus in 1959 to generate and parse Fortran.
- Example:
  - terminals: b, e
  - nonterminals: S, A
  - Production Rules: S -> SA, S -> A, A -> bSe, A -> be
  - S is the start symbol

Context-Free Grammar Example

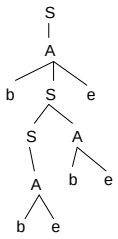
- S -> SA
  S -> A
  A -> bSe
  A -> be

Example: b and e matched as parentheses

derivation of bbebee

S
A
bSe
bSAe
bAAe
bbeAe
bbebee

hierarchical parse tree



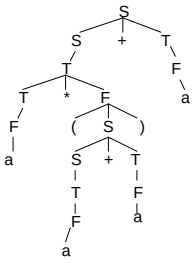
Arithmetic Expressions

- S -> S + T
  S -> T
  T -> T*F
  T -> F
  F -> a
  F -> (S)

derivation of a * (a + a) + a

S
S+T
T+T
T*F+T
F*F+T
a*F+T
a*(S)+F
a*(S+F)+T
a*(T+F)+T
a*(F+F)+T
a*(a+F)+T
a*(a+a)+T
a*(a+a)+F
a*(a+a)+a

parse tree



Sequitur Principles

- Digram Uniqueness:
  - no pair of adjacent symbols (digram) appears more than once in the grammar.
- Rule Utility:
  - Every production rule is used more than once.
- These two principles are maintained as an invariant while inferring a grammar for the input string.

Sequitur Example (1)

bbebeebebebbebee

S -> b

Sequitur Example (2)

bbebeebebebbebee

S -> bb

Sequitur Example (3)

bbebeebebebbebee

S -> bbe

Sequitur Example (4)

bbebeebebebbebee

S -> bbeb

Sequitur Example (5)

bbebeebebebbebee

S -> bbebe

Enforce digram uniqueness.
be occurs twice.
Create new rule A ->be.

Sequitur Example (6)

bbebeebebebbebee

S -> bAA
A -> be

Sequitur Example (7)

bbebeebebebbebee

S -> bAAe
A -> be

Sequitur Example (8)

bbebeebebebbebee

S -> bAAeb
A -> be

Sequitur Example (9)

bbebeebebebbebee

S -> bAAebe
A -> be

Enforce digram uniqueness.
be occurs twice.
Use existing rule A -> be.

Sequitur Example (10)

bbebeebebebbebee

S -> bAAeA
A -> be

Sequitur Example (11)

bbebeebebebbebee

S -> bAAeAb
A -> be

Sequitur Example (12)

bbebeebebebbebee

S -> bAAeAbe
A -> be

Enforce digram uniqueness.
be occurs twice.
Use existing rule A -> be.

Sequitur Example (13)

bbebeebebebbebee

S -> bAAeAA
A -> be

Enforce digram uniqueness
AA occurs twice.
Create new rule B -> AA.

Sequitur Example (14)

bbebeebebebbebee

S -> bBeB
A -> be
B -> AA

Sequitur Example (15)

bbebeebebebbebee

S -> bBeBb
A -> be
B -> AA

Sequitur Example (16)

bbebeebebebbebee

S -> bBeBbb
A -> be
B -> AA

Sequitur Example (17)

bbebeebebebbebee

S -> bBeBbbe
A -> be
B -> AA

Enforce digram uniqueness.
be occurs twice.
Use existing rule A ->be.

Sequitur Example (18)

bbebeebebebbebee

S -> bBeBbA
A -> be
B -> AA

Sequitur Example (19)

bbebeebebebbebee

S -> bBeBbAb
A -> be
B -> AA

Sequitur Example (20)

bbebeebebebbebee

S -> bBeBbAbe	Enforce digram uniqueness.
A -> be	be occurs twice.
B -> AA	Use existing rule A -> be.

Sequitur Example (21)

bbebeebebebbebee

S -> bBeBbAA	Enforce digram uniqueness.
A -> be	AA occurs twice.
B -> AA	Use existing rule B -> AA.

Sequitur Example (22)

bbebeebebebbebee

S -> bBeBbB	Enforce digram uniqueness.
A -> be	bB occurs twice.
B -> AA	Create new rule C -> bB.

Sequitur Example (23)

bbebeebebebbebee

S -> CeBC
A -> be
B -> AA
C -> bB

Sequitur Example (24)

bbebeebebebbebee

S -> CeBCe	Enforce digram uniqueness.
A -> be	Ce occurs twice.
B -> AA	Create new rule D -> Ce.
C -> bB	

Sequitur Example (25)

bbebeebebebbebee

S -> DBD	Enforce rule utility.
A -> be	C occurs only once.
B -> AA	Remove C -> bB.
C -> bB	
D -> Ce	

Sequitur Example (26)

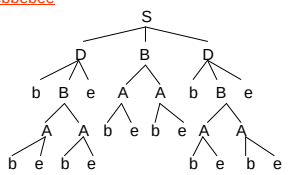
bbebeebebebbebee

S -> DBD
A -> be
B -> AA
D -> bBe

The Hierarchy

bbebeebebebbebee

S -> DBD
A -> be
B -> AA
D -> bBe



Is there compression? In this small example, probably not.

Sequitur Algorithm

Input the first symbol s to create the production S -> s;
repeat
  match an existing rule:
    A -> ….XY…
    B -> XY
    ⟶ A -> ….B….
      B -> XY
  create a new rule:
    A -> ….XY….
    B -> ….XY….
    ⟶ A -> ….C….
      B -> ….C….
      C -> XY
  remove a rule:
    A -> ….B….
    B -> $X_1X_2…X_k$
    ⟶ A -> …. $X_1X_2…X_k$ ….
  input a new symbol:
    S -> $X_1X_2…X_k$ ⟶ S -> $X_1X_2…X_k$s
until no symbols left

Complexity

- The number of non-input sequitur operations applied ≤ 2n where n is the input length.
- Amortized Complexity Argument
  - Let s = the sum of the right hand sides of all the production rules. Let r = the number of rules.
  - We evaluate 2s - r.
  - Initially 2s - r = 1 because s = 1 and r = 1.
  - 2s - r ≥ 0 at all times because each rule has at least 1 symbol on the right hand side.
  - 2s - r increases by 2 for every input operation.
  - 2s - r decreases by at least 1 for each non-input sequitur rule applied.

Sequitur Rule Complexity

- Digram Uniqueness - match an existing rule.

A -> ….XY… B -> XY	⟶	A -> ….B…. B -> XY	s r -1 0	2s - r -2

- Digram Uniqueness - create a new rule.

A -> ….XY…. B -> ….XY….	⟶	A -> ….C…. B -> ….C…. C -> XY	s r 0 1	2s - r -1

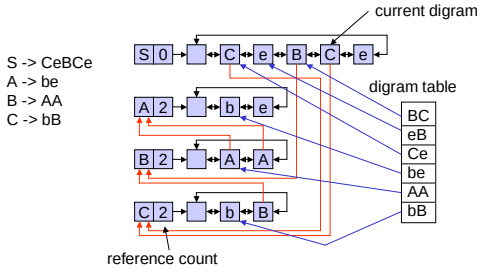
- Rule Utility - Remove a rule.

A -> ….B…. B -> $X_1X_2…X_k$	⟶	A -> …. $X_1X_2…X_k$ ….	s r -1 -1	2s - r -1

Linear Time Algorithm

- There is a data structure to implement all the sequitur operations in constant time.
  - Production rules in an array of doubly linked lists.
  - Each production rule has reference count of the number of times used.
  - Each nonterminal points to its production rule.
  - digrams stored in a hash table for quick lookup.

Data Structure Example



Basic Encoding a Grammar

Grammar

S -> DBD
A -> be
B -> AA
D -> bBe

Symbol Code

b 000
e 001
S 010
A 011
B 100
D 101
110

Grammar Code

D	B	D	#	b	e	#	A	A	#	b	B	e	
101	100	101	110	000	001	110	011	011	110	000	100	001	39 bits

$$|\text{Grammar Code}| = (s + r - 1)\lceil \log_2(r + a + 1) \rceil$$

r = number of rules
s = sum of right hand sides
a = number in original symbol alphabet

Better Encoding of the Grammar

- Nevill-Manning and Witten suggest a more efficient encoding of the grammar that resembles LZ77.
  - The first time a nonterminal is sent, its right hand side is transmitted instead.
  - The second time a nonterminal is sent the new production rule is established with a pointer to the previous occurrence sent along with the length of the rule.
  - Subsequently, the nonterminal is represented by the index of the production rule.

Compression Quality

- Neville-Manning and Witten 1997

	size	compress	gzip	sequitur	PPMC
bib	111261	3.35	2.51	2.48	2.12
book	768771	3.46	3.35	2.82	2.52
geo	102400	6.08	5.34	4.74	5.01
obj2	246814	4.17	2.63	2.68	2.77
pic	513216	0.97	0.82	0.90	0.98
progc	38611	3.87	2.68	2.83	2.49

Files from the Calgary Corpus
Units in bits per character (8 bits)
compress - based on LZW
gzip - based on LZ77
PPMC - adaptive arithmetic coding with context

Notes on Sequitur

- Very new and different from the standards.
- Yields compression and hierarchical structure simultaneously.
- With clever encoding is competitive with the best of the standards.
- Practical linear time encoding and decoding.
- Alternatives
  - Off-line algorithms – (i) find the most frequent digram, (ii) find the longest repeated substring