COMP 9319 Web Data Compression and Search

Recap for Compression;

Preview on Search;

Q&A for a1

Agenda for today

Where we are?

- Recap for Huffman & AC
- LZW, Adaptive Huffman & BWT overview
- Roadmap: Compression -> Search

Other course-related matters

- Reference papers on WebCMS3
- Q&As in Ed Forum & Consultations
- Regular exercises (started this week)
- Assignment 1 spec (how to start / Q&A)

Compression

- Minimize amount of information to be stored / transmitted
- Transform a sequence of characters into a new bit sequence
  - same information content (for lossless)
  - as short as possible

Run-length coding

- Run-length (encoding) is a very widely used mple compression technique
  - does not assume a memoryless source
  - replace runs of symbols (possibly of length one) with pairs of (symbol, run-length)

Uniquely decodable

- Uniquely decodable is a prefix free code
  if no codeword is a proper prefix of any other

- For example {1, 100000, 00} is uniquely decodable, but is not a prefix code
  – consider the codeword {…1000000001…}

- In practice, we prefer prefix code (why?)

Static codes

- Mapping is fixed before transmission
  - E.g., Huffman coding
- probabilities known in advance

Dynamic codes

- Mapping ch[illegible] over time
  - i.e. adaptiv[illegible]g
- Attempts to exploit locality of reference
  - periodic, frequent occurrences of messages
  - e.g., dynamic Huffman

Variable length coding

- Also known as entropy coding
  - The number of bits used to code symbols in the alphabet is variable
  - E.g. Huffman coding, Arithmetic coding

Entropy

- What is the minimum number of bits per symbol?
- Answer: Shannon's result – theoretical minimum average number of bits per code word is known as Entropy (H)

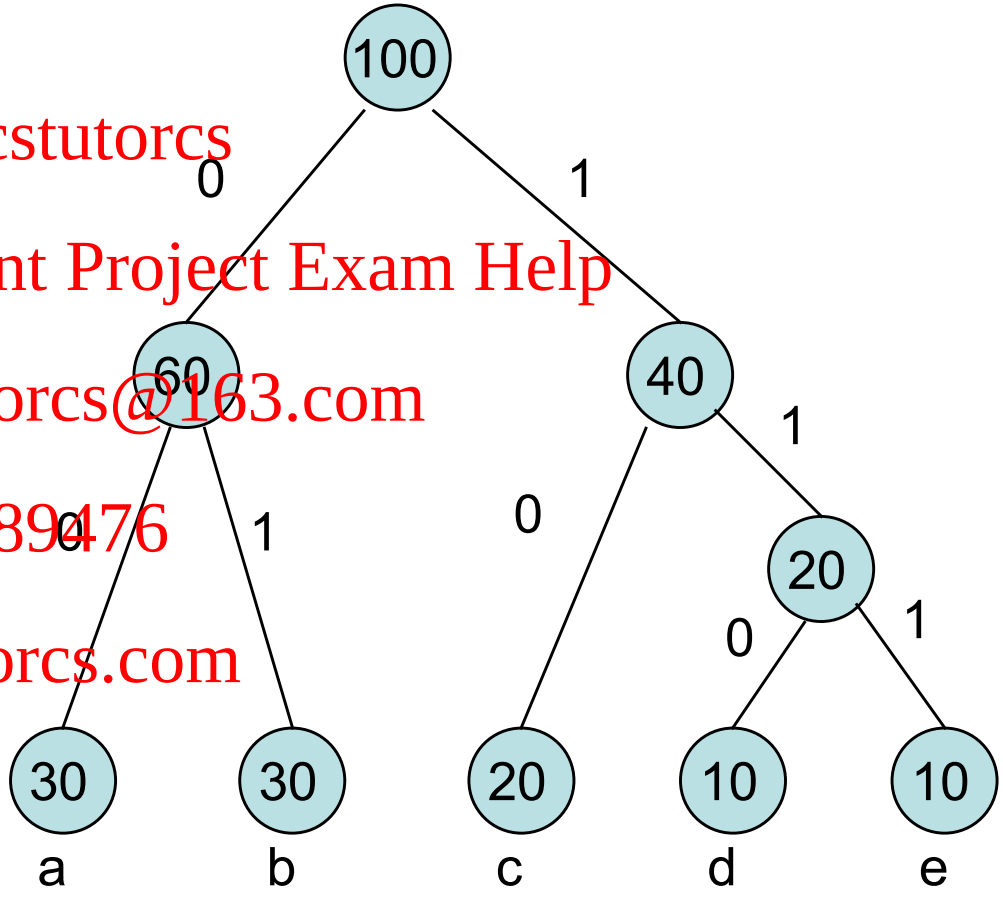
$$-\sum_{i=1}^{n} p(s_i)\log_2 p(s_i)$$

Huffman coding algorithm

1. Take the two least probable symbols in the alphabet
   (longest code words, equal length, differing in last digit)
2. Combine these two symbols into a single symbol
3. Repeat

Example

S	Freq	Huff
a	30	00
b	30	01
c	20	10
d	10	110
e	10	111



Another example

- S={a, b, c, d} with {4, 2, 1, 1}
- $H = 4/8*\log_2 2 + 2/8*\log_2 4 + 1/8*\log_2 8 + 1/8*\log_2 8$

- H = 1/2 + 1/2 + 3/8 + 3/8 = 1.75

- a => 0 b => 10 c => 110 d => 111
- Message: {abcdabaa} => {0 10 110 111 0 10 0 0}

- Average length L = 14 bits / 8 chars = 1.75
- If equal probability, i.e. fixed length, need $\log_2 4 = 2$ bits

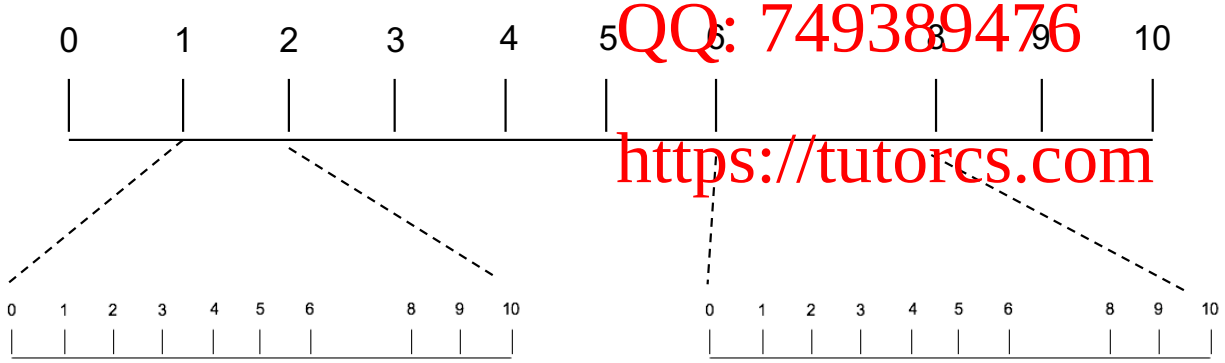
Problems of Huffman coding

- Huffman codes have an integral # of bits.
  - E.g., log (3) = 1.5 while Huffman may need 2 bits
- Noticeable non-optimality when prob of a symbol is high.

=> Arithmetic coding

Arithmetic coding

Character	Probability	
SPACE	1/10	0
A	1/10	0
B	1/10	0.20 - 0.30
E	1/10	0.30 - 0.40
G	1/10	0.40 - 0.50
I	1/10	0.50 - 0.60
L	2/10	0.60 - 0.80
S	1/10	0.80 - 0.90
T	1/10	0.90 - 1.00



Arithmetic coding

New Charact	w value	High Value
	0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

COMP 9319 Web Data Compression and Search

LZW, Adaptive Huffman

Dictionary coding

- Patterns: co[illegible]ons between part of the data
- Idea: replace recurring patterns with references to dictionary
- LZ algorithms are adaptive:
  - Universal coding (the prob. distr. of a symbol is unknown)
  - Single pass (dictionary created on the fly)
  - No need to transmit/store dictionary

Lempel-Ziv-Welch (LZW) Algorithm

- Most popular modification to LZ78
- Very common, e.g., Unix compress, TIFF, GIF, PDF (until recently)
- Read http://en.wikipedia.org/wiki/LZW regarding its patents
- Fixed-length references (12bit 4096 entries)
- Static after max entries reached

Problems of Huffman coding

Need statistics of data: e.g., single pass over the data to collect stat & stat unchanged during encoding

To decode, the stat table need to be transmitted. Table size can be significant for small msg.

=> Adaptive compression e.g., adaptive huffman

Adaptive Huffman Coding (dummy)

Encoder

Reset the stat

Repeat for each input char

(

Encode char

Update the stat

Rebuild huffman tree

)

Decoder

Reset the stat

Repeat for each input char

(

Decode char

Update the stat

Rebuild huffman tree

)

This works but too slow!

Terminology (Types)

- Block-block
  - source message: fixed; codeword: fixed length
  - e.g., ASCII
- Block-variable
  - source message: fixed; codeword: variable
  - e.g., Huffman coding
- Variable-block
  - source message: variable; codeword: fixed
  - e.g., LZW
- Variable-variable
  - source message and codeword: variable
  - e.g., Arithmetic coding

Summarised schedule

0. Information representation (today)
1. Compression
2. Search
3. Compression + Search on plain text
4. "Compression + Search" on Web text
5. Selected advanced topics (if time allows)

COMP 9319 Web Data Compression and Search

Basic BWT

Basic BWT

(to be discussed more detailed next week)

Recall from Lecture 1's RLE and BWT example

rabcabcababa$ bcabcabcababaa$

aabbbbccaccrcbaaaaaaaaaabbbbba$

aab4ccac3rcba10b5a$

A simple example

Input:

#BANANAS

All rotations

NANAS
ANANA
NANAS
ANAS#BANAN
NAS#BANA
ANAS#BAN
NANAS#BA
ANANAS#B
BANANAS#

Sort the rows


```
NANAS
ANANAS#B
NANAS#BAN
ANAS#BANA
NAS#BANAN
AS#BANANA
S#BANANAS
#BANANAS
BANANAS#
ANANAS#BA
NANAS#BAN
ANAS#BANA
NAS#BANAN
AS#BANANA
S#BANANAS
#BANANAS
```

Output

```
NANAS
ANANAS#B
NANAS#BA
ANAS#BAN
AS#BANAN
S#BANANAS#
NANAS#BA
ANAS#BAN
NAS#BANA
S#BANANA
```

Exercise: you can try this example

rabcabcababa  bcabcabcababaa$

aabbbbccaccbcbaaaaaaaaaaabbbbba$

Now the inverse, for decoding…

Input:

S

B

N

N

#

A

A

A

First add

Then sort

Add again


```
S#
BA
NA
NA
#B
AN
AN
AS
```

Then sort


```
#B
AN
AN
AS
BA
NA
NA
S#
```

Thread


```
S#B
BAN
NAN
NAS
#BA
ANA
ANA
AS#
```

Then sort


```
#BA
ANA
ANA
AS#
BAN
NAN
NAS
S#B
```

Thenadd


```
S#BA
BANA
NANA
NAS#
#BAN
ANAN
ANAS
AS#B
```

Then sort


```
#BAN
ANAN
ANAS
AS#B
BANA
NANA
NAS#
S#BA
```

Thenadd


```
S#BAN
BANAN
NANAS
NAS#B
#BANA
ANANA
ANAS#
AS#BA
```

Then sort


```
#BANA
ANANA
ANAS#
AS#BA
BANAN
NANAS
NAS#B
S#BAN
```

Thread


```
S#BANA
BANANA
NANAS#
NAS#BA
#BANAN
ANANAS
ANAS#B
AS#BAN
```

Then sort


```
#BANAN
ANANAS
ANAS#B
AS#BAN
BANANA
NANAS#
NAS#BA
S#BANA
```

Then add


```
S#BANAN
BANANAS
NANAS#B
NAS#BAN
#BANANA
ANANAS#
ANAS#BA
AS#BANA
```

Then sort


```
#BANANA
ANANAS#
NAS#BA
AS#BANA
BANANAS
NANAS#B
NAS#BAN
S#BANAN
```

Thenadd


```
S#BANANA
BANANAS#
NANAS#BA
NAS#BANA
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
```

Then sort (???)


```
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
BANANAS#
NANAS#BA
NAS#BANA
S#BANANA
```

Exercise: you can try this example

rabcabcababa$bcabcabcababaa$

aabbbbccaccbcbaaaaaaaaaaabbbbba$

Reference Papers on WebCMS3

A Method for the Construction of Minimum-Redundancy Codes*

DAVID A. HUFFMAN+, ASSOCIATE, IRE

Summary—An optimum method of coding an ensemble of messages consisting of a finite number of members is developed. A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.

Introduction

ONE IMPORTANT METHOD of transmitting messages is to transmit in their place sequences of symbols. If there are more messages which might be sent than there are kinds of symbols available, then some of the messages must use more than one symbol. If it is assumed that each symbol requires the same time for transmission, then the time for transmission (length) of a message is directly proportional to the number of symbols associated with it. In this paper, the symbol or sequence of symbols associated with a given message will be called the "message code." The entire number of messages which might be transmitted will be ...ined and here as an ensemble code which, for a message ensemble consisting of a finite number of members, N, and for a given number of coding digits, D, yields the lowest possible average message length. In order to avoid the use of the lengthy term "minimum-redundancy," this term will be replaced here by "optimum." It will be understood then that, in this paper, "optimum code" means "minimum-redundancy code."

The following basic restrictions will be imposed on an ensemble code:

(a) No two messages will consist of identical arrangements of coding digits.

(b) The message codes will be constructed in such a way that no additional indication is necessary to specify where a message code begins and ends once the starting point of a sequence of messages is known.

Q&As in Ed Forum

Hey all,

I just finished the arithmetic coding l wondering why we need to worry about using different probabilities for encoding?

E.g. if we knew we were just encoding English alphabet letters and spaces, is there a strong downside to just using a $\frac{1}{27}$ split of the $[0, 1]$ range and running the algorithm? I assume it has something to do with switching from pure real numbers to binary representations that the probabilities come in to play, but just wanted to ask in case there is a different explanation as well?

Thanks!

This is my somewhat limited understanding from reading parts of the AC paper:

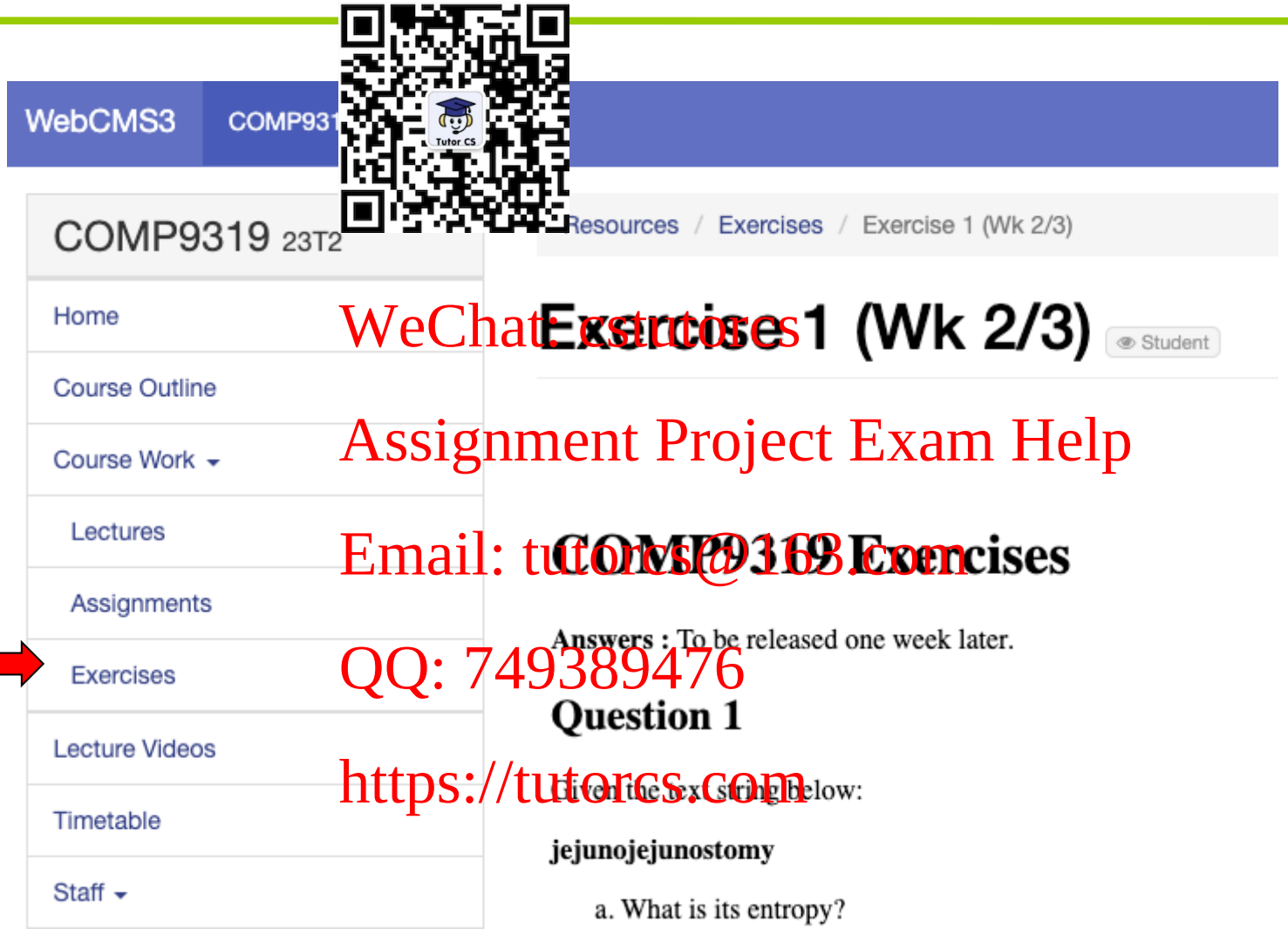
If you allocate a *smaller* range for a symbol, then you must transmit *more* bits in order to encode that symbol. This is because a smaller (i.e. narrower) range requires more decimal places to represent a number that lies in that range. More decimal places => longer binary representation.

For the English language, you would be using unnecessary extra bits to encode those characters that appear more frequently than others (vowels, for example).

Q&As from Consultations

For Huffman, AC, LZW we covered so far, what if we consider source messages in UTF8 instead of ASCII?

Exercises on WebCMS3



Assignment 1

COMP9319 2023T2 Assignment 1: LZW Encoding and Decoding

Your task in this assignment is to implement an LZW encoder and its decoder with 15-bit 32768 dictionary entries (excluding those entries for the individual ASCII characters), called `lencode` and `ldecode`, in C or C++. After the dictionary is full, no new entries can be added. You may assume the source file only contain any 7-bit ASCII characters.


```
%grieg> lencode ~cs9319/a1/test1.txt test1.encoded
%grieg> ldecode test1.encoded test1.decoded
%grieg> diff ~cs9319/a1/test1.txt test1.decoded
%grieg>
```


test1.lzw using xxd:


```
cs9319@grieg:~/a1$ xxd -b test1.txt
00000000: 01011110 01010111 01000101 01000100 01011110 01010111  ^WED^W
00000006: 01000101 01011110 01010111 01000101 01000101 01011110  E^WEE^
0000000c: 01010111 01000101 01000010 01011110 01010111 01000101  WEB^WE
00000012: 01010100                                               T
cs9319@grieg:~/a1$ xxd -b test1.lzw
00000000: 01011110 01010111 01000101 01000100 01011110 01010111  ^WED^W
00000006: 01000101 10000000 00000100 01000101 01011110 01010111  E..E^W
0000000c: 01000101 01000010 10000000 00000100 01010100           EB..T
cs9319@grieg:~/a1$
```

Hint: Setting MSB to 1

```
Unsigned_T val;

val = …

msb = ((Unsigned_T)-1 >> 1) + 1;

val |= msb;
```

Important: if you use your PC to code a1

Make sure you reserve time to:

- port & compile
- test & debug

on *grieg.cse.unsw.edu.au* before you submit.