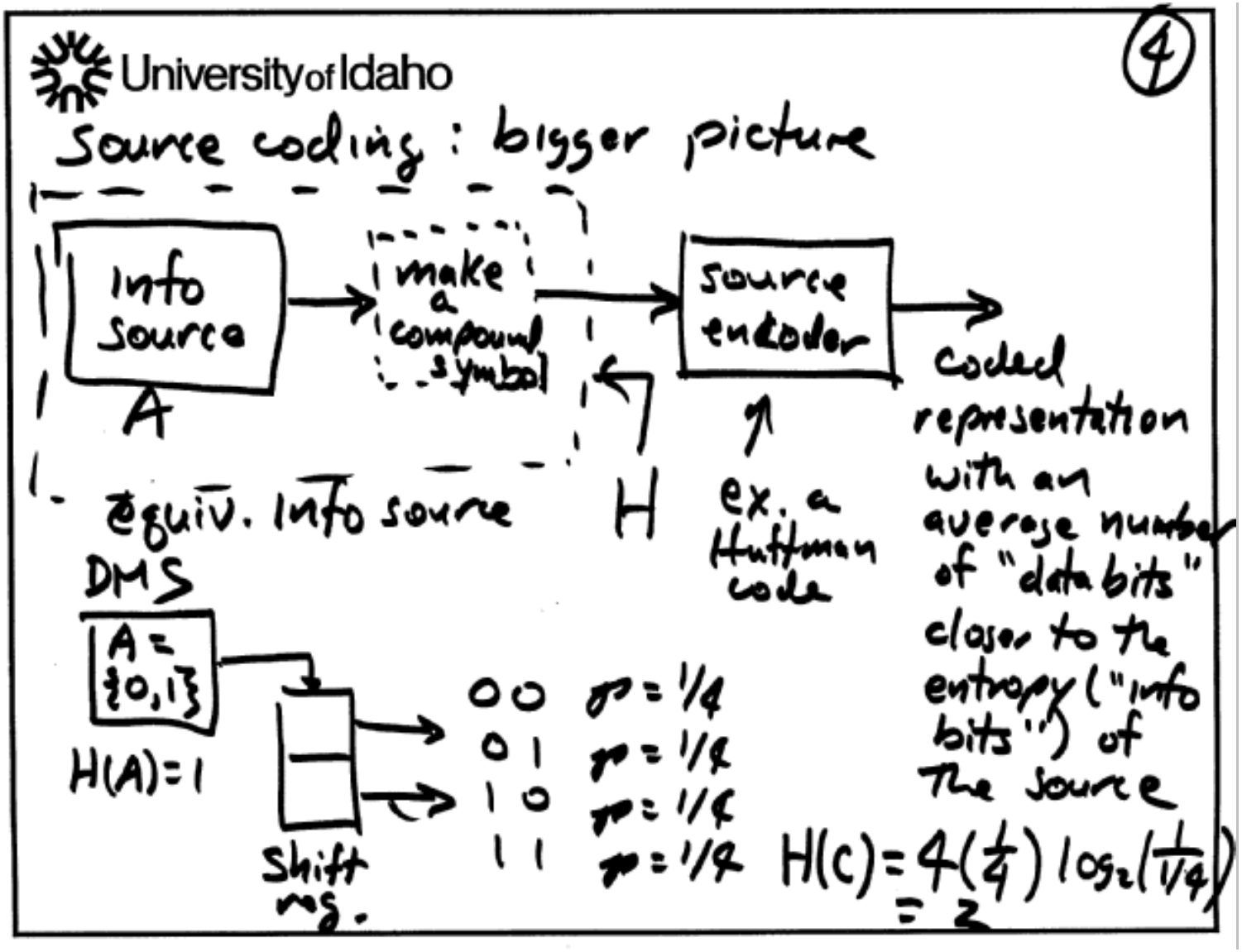
16.548 Notes III

Lossless and Lossy Compression Algorithms


University of Idaho
4
Source coding : bigger picture
info source
A
make a compound symbol
source encoder
coded representation with an average number of "data bits" closer to the entropy ("info bits") of the source
equiv. info source
H
ex. a Huffman code
DMS
A = {0,1}
H(A)=1
Shift reg.
00 p = 1/4
01 p = 1/4
10 p = 1/4
11 p = 1/4
H(C) = 4(1/4) log2(1/1/4)
= 2

Review: Joint Entropy

$$H(A,B) \quad \text{from eq.} \quad \sum_{a_i} \sum_{b_j} p_{i,j} \log_2\left(\frac{1}{p_{i,j}}\right)$$

$$H(A,B) = H(A) + H(B|A) = H(B) + H(A|B)$$

University of Idaho

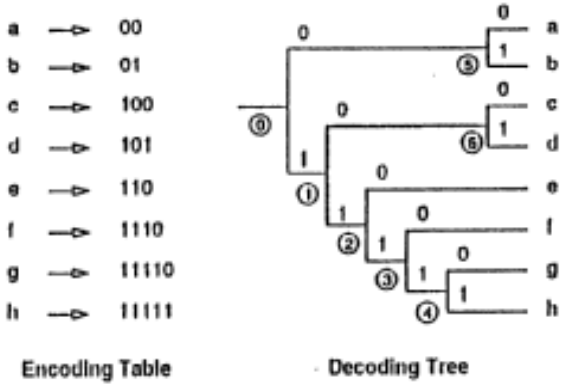


Figure 12.1 Example of a Huffman encoding table with the corresponding decoding tree.

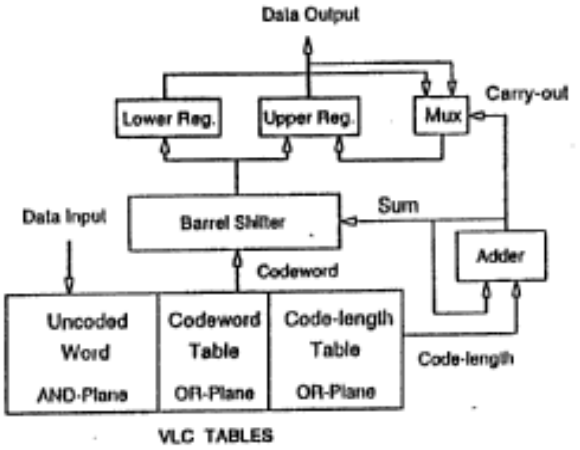


Figure 12.2 Block diagram of the Lei-Sun VLC encoder.

Input	Upper Register	Lower Register	Sum	Carry Out
g	1111000000000000	0000000000000000	5	0
b	1111001000000000	0000000000000000	7	0
a	1111001000000000	0000000000000000	9	0
c	1111001001000000	0000000000000000	12	0
b	1111001001000100	0000000000000000	14	0
f	1111001001000111	1000000000000000	2	1
d	1010100000000000	0000000000000000	5	0

Table 12.1 Example of operation of the VLC encoder.

University of Idaho

input stream

11110.01.00.100.01.

state machine

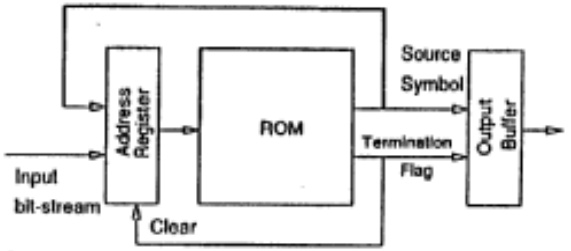


Figure 12.3 Block diagram of a bit-serial Huffman decoder.

Address		Output	
Node	Input	Next-Node/ Codeword	Termination Flag
0	0	5	0
0	1	1	0
1	0	6	0
1	1	2	0
2	0	e	1
2	1	3	0
3	0	f	1
3	1	4	0
4	0	g	1
4	1	h	1
5	0	a	1
5	1	b	1
6	0	c	1
6	1	d	1

Table 12.2 Example of ROM entries for bit-serial Huffman decoding; input is one bit at a time.

University of Idaho

5

Some issues w/ Huffman Code

1) Have to have the symbol Probabilities
2) oftentimes it is not too feasible that we will know the probabilities
   - Probabilities are time-varying
   - what if our info source has different kinds of info
     e.g. ASCII text and graphics
     bit maps and numerical data

What do we do about it?

1) make the encoder measure the probabilities "on the fly" and adapt the coding tree

2) use a code that does not need to know the probabilities

- Dictionary codes
  - build their own dictionary

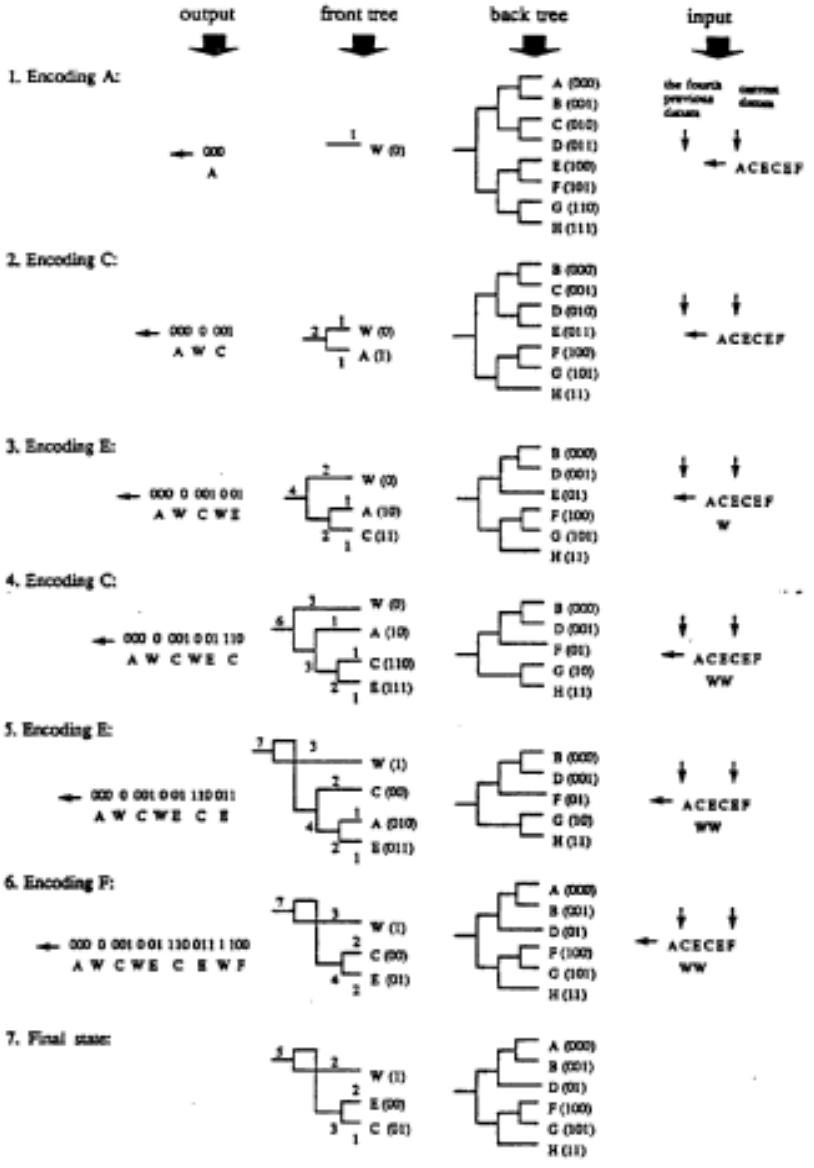
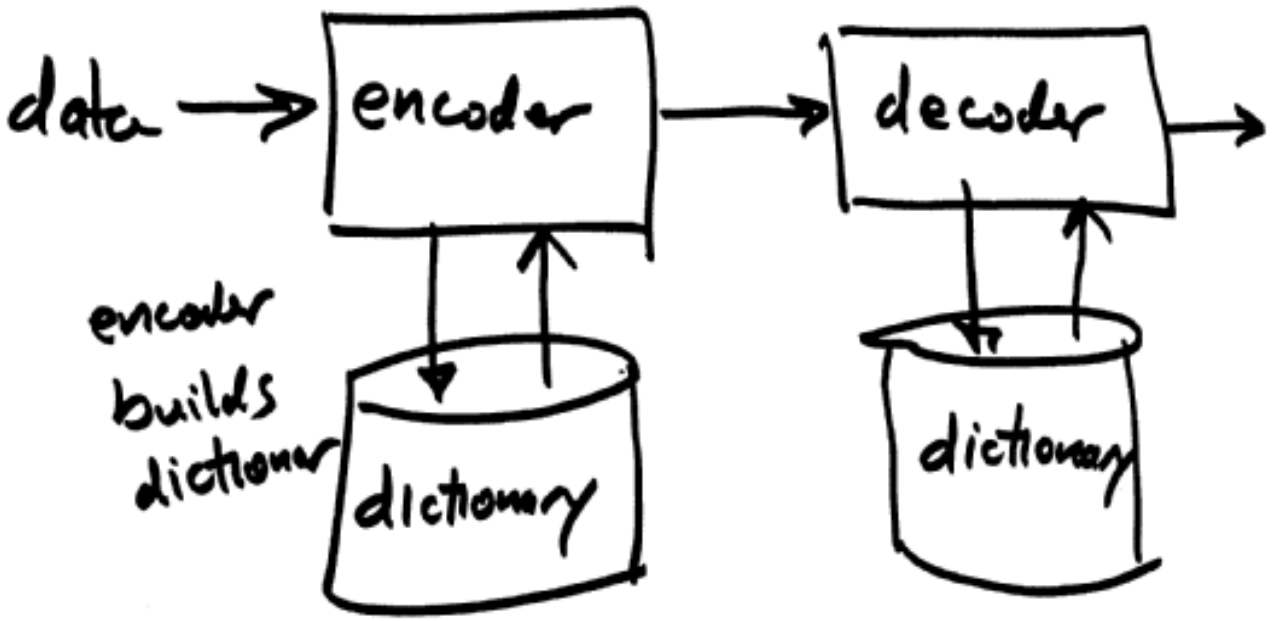


Fig. 1. An example of the fast-adaptive Huffman encoding.

University of Idaho



1) achieve compression

2) decoder must be able to build the dictionary "on-the-fly"

University of Idaho

9

Lempel-Ziv codes

family of dictionary codes

LZ codes are (or were)

the "compress" command

in Unix

The Inventors: Lempel and Ziv

Improvements in the field of data compression up until the late 1970's were chiefly directed towards creating better methodologies for Huffman Coding. In order to obtain the necessary frequencies of symbols in the data to be compressed, these methods either had to rely on the ability to predict such occurrences or would require that the text be read in beforehand.

In 1977, two Israeli information theorists, Abraham Lempel and Jacob Ziv, introduced a radically different way of compressing data - one which would avoid the uncertainty of making predictions or the wastefulness of pre-reading data. LZ77 and LZ78, two dictionary-based data compression techniques described by these two researchers, provided a whole new way of viewing the world of data compression.

Ziv-Lempel Codes

Lempel-Ziv Variants

The highly innovative LZ77 and LZ78 methods, although usually referred to as the singular Lempel-Ziv (LZ) or Ziv-Lempel method, are in fact considerably different. Both algorithms are based on the principle of storing pointers to the data. The pointers make references to phrases which have made previous appearances. In both cases, encoding proceeds "on the fly," so that as symbols are read in, more and more phrases are stored (in that respect, the LZ compression techniques are a bit like a suffix trie). It is important, however, to note that LZ77 and LZ78 are distinct. The method which will be discussed here, and which will later be treated as "the" Lempel-Ziv compression technique, is in reality the latter of the two.

Lempel and Ziv's compression techniques of '77 and '78 gave rise to a series of variants which form part of the LZ family. The variants are essentially identical to the method from which they originate (either LZ77 or LZ78). Variations have to do with factors such as establishing how many bits to use for certain boundaries. A list of these coding algorithms is provided below (Table 1).

LZ77 Variants	LZR	LZSS	LZB	LZH		
LZ78 Variants	LZW	LZC	LZT	LZMW	LZJ	LZFG

Table 1: LZ77 and LZ78 Variants

As an example, the zip and unzip methods use the LZH techique. UNIX compress methods, on the other hand, belong to the LZW and LZC classes.

A question that would naturally arise at this point is "Why so many variants?" The answer is...patents. Disputes over patenting have actually slowed down the evolutionary progress of the LZ method.

University of Idaho EE455 Lec 8 (1)

Lempel-Ziv codes

- family of codes
- dynamic dictionary codes

What do LZ codes do? They work by building a dictionary of "phrases"

for example

ä : 1 10 0 01 011 001 011 0

see text on pg 23

General Overview

To illustrate the LZ compression technique, we take a simple alphabet containing only two letters, a and b, and create a sample stream of text. An example of such a stream is shown below (Figure 1):

aaababbbaaabaaaaaaabaabb

Figure 1: A stream of a's and b's in a sample text

Rule: Separate this stream of characters into pieces of text so that the shortest piece of data is the string of characters that we have not seen so far.

According to the above rule, we see that the first piece of our sample text is **a**. The second piece must then be **aa**. If we go on like this, we obtain the breakdown of data illustrated in Figure 2:

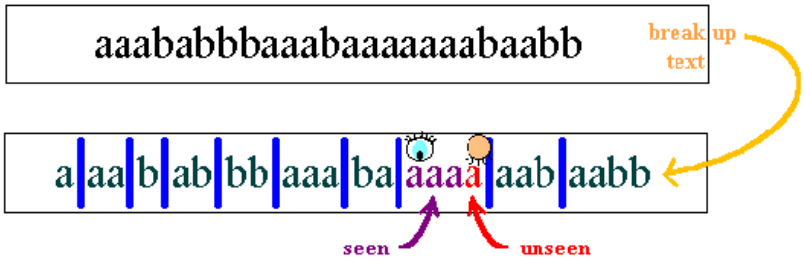


Figure 2: How a sample text gets broken up

Sender : The Compressor

Before compression, the pieces of text obtained in the breaking-down process must be indexed from 1 to n, as in Figure 3 below:

Figure 3: Indexing the pieces of our sample text

These indices are used to number the pieces of data. The empty string (start of text) has index 0. The piece indexed by 1 is **a**. Thus **a**, together with the initial string, must be numbered **0a**. String 2, **aa**, will be numbered **1a**, because it contains **a**, whose index is 1, and the new character **a**. In this way, we proceed to number all the pieces in terms of those that came before them. See Figure 4 for an illustration of this technique:

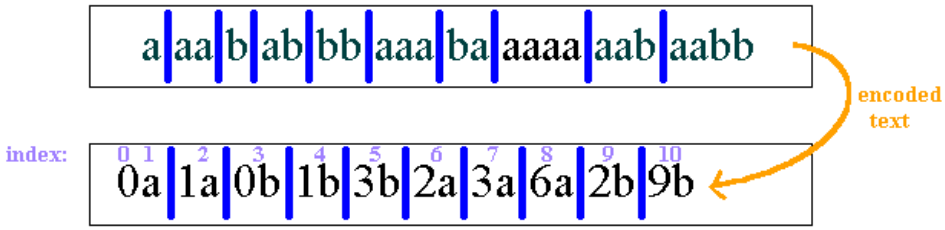


Figure 4: "Numbering" each phrase in the text

As we can see, this process of renaming pieces of text starts to pay off. Small integers replace what were once long strings of characters. We can now throw away our old stream of text and send the encoded information to the receiver, as will be discussed in the next section.

Bit Representation of Coded Information

Now, we wish to calculate the number of bits needed to represent this coded information. As we have seen before, each piece of text is composed of an integer and a letter from our alphabet.

The number of bits needed to represent each integer with index n is at most equal to the number of bits needed to represent the (n-1)th index. For example, the number of bits needed to represent 6 in piece 8 is equal to 3, because it takes three bits to express 7 (the (n-1)th index) in binary.

Every character will occupy 8 bits because it will be represented in US ASCII format. Figure 5 illustrates this thinking process:

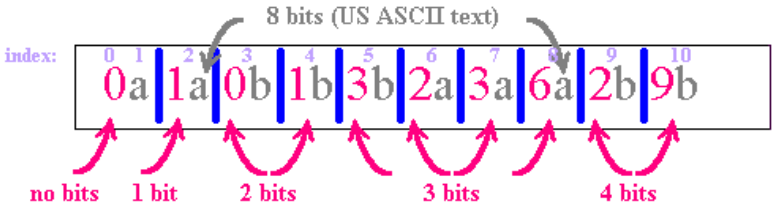


Figure 5: Calculating the number of bits occupied by each phrase

Receiver: The Decompressor (Implementation)

Since the receiver knows exactly where the boundaries are, there is no problem in reconstructing the stream of text.

It is preferrable to decompress the file in one pass; otherwise, we will encounter a problem with temporary storage. This feature allows you to process many megs of information at one time.

Given the code, the receiver constructs a trie "on the fly". This must be identical to the trie contructed by the sender. It is illustrated below (Figure 6):

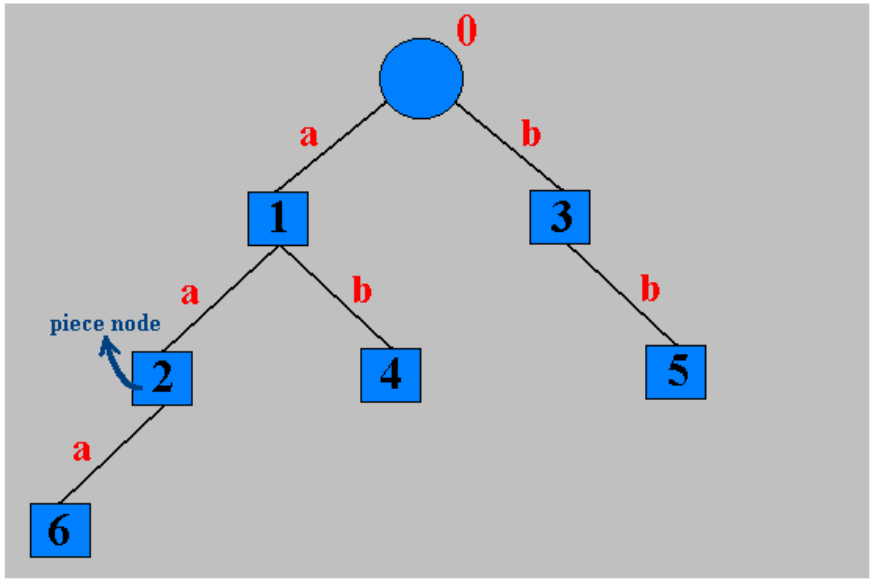


Figure 6: The trie built from our sample stream of a's and b's

Note: To find the contents of a particular piece, trace the path from the node to the root. For example, piece 2 will contain **1a**.

The structure shown above is called a digital search tree and is part of the trie family. It is different from the standard trie because the internal nodes do not live only in the leaves.

University of Idaho

- initialize dictionary
- n = 0

source symbol: a
linked list pointer: n
dictionary entries: ⟨n, a⟩

(2)

1. Fetch next source symbol a;
2. If the ordered pair $\langle n, a \rangle$ is already in the dictionary then

   n = dictionary address of entry $\langle n, a \rangle$;

   else

   transmit n

   create new dictionary entry $\langle n, a \rangle$ at dictionary address m

   $m = m + 1$

   n = dictionary address of entry $\langle 0, a \rangle$;
3. Return to step 1.

let $A = \{0, 1\}$

EX 1.5.1 initial dictionary

m	n	word
0	⟨0	null⟩
1	⟨0	0⟩
2	⟨0	1⟩

initial n = 0

University of Idaho $A = \{0, 1\}$ (3)

Example: α : 1 1 0. 0 0 1. 0 1 1

initial $n = 0$

1) $a = 1$

m	n	a	← initial dictionary
0	0	null	
1	0	0	
2	<0	1>	reset $n = 2$

$\langle n, a \rangle = \langle 0, 1 \rangle$

3	2	1

2) $a = 1$

$\langle n, a \rangle = \langle 2, 1 \rangle$ not in dictionary! Xmit 2

reset $n = 2$

3) a = 0

⟨n, a⟩ = ⟨2, 0⟩

not in dict.

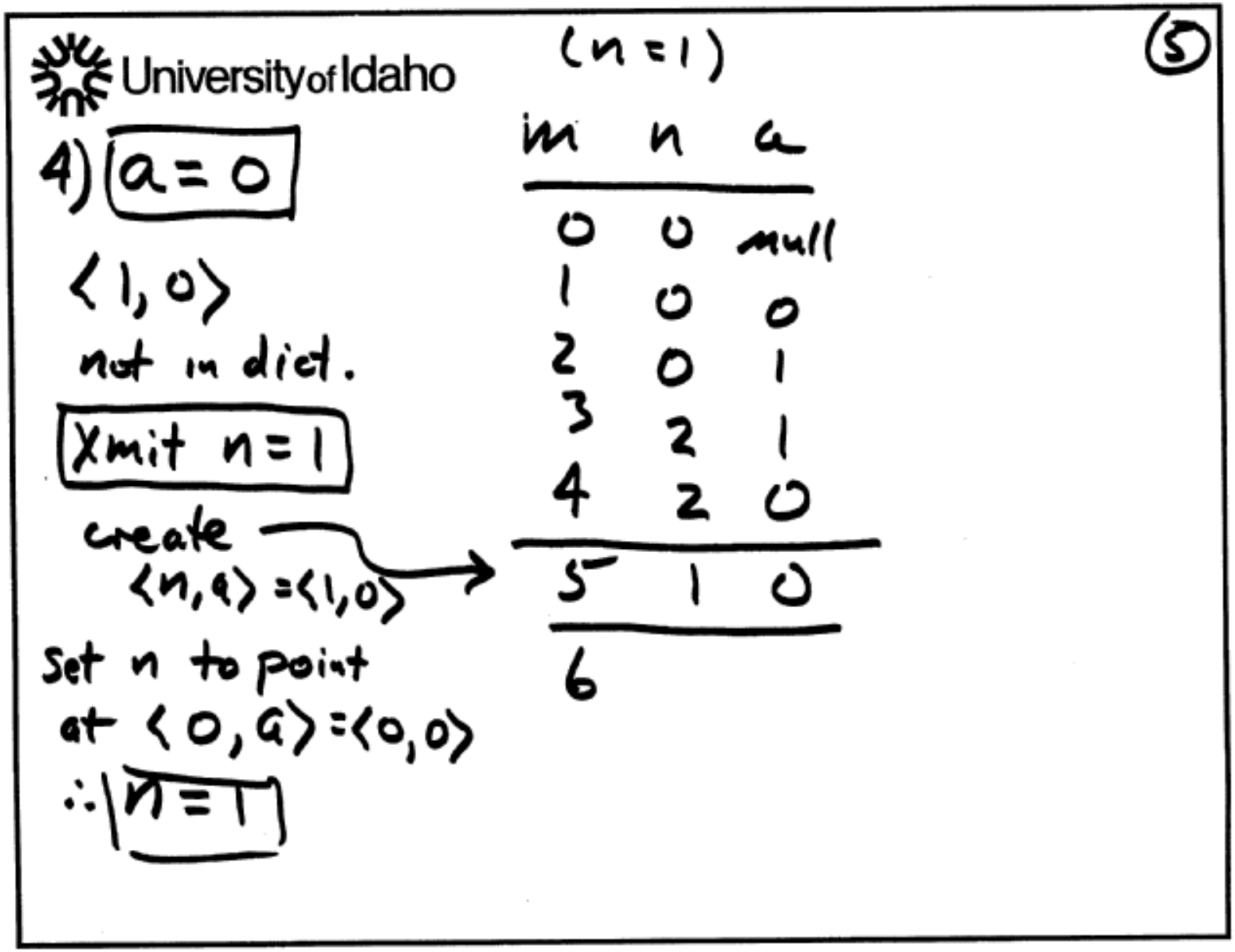
so: Xmit 2

create →

reset n to point at ⟨0, a⟩ = ⟨0, 0⟩

∴ n = 1

m	n	a
0	0	null
1	0	0
2	0	1
3	2	1
4	2	0


University of Idaho
(n = 1)
5
4) a = 0
⟨1, 0⟩
not in dict.
Xmit n = 1
create
⟨n, a⟩ = ⟨1, 0⟩
Set n to point
at ⟨0, a⟩ = ⟨0, 0⟩
∴ n = 1
in n a
0 0 null
1 0 0
2 0 1
3 2 1
4 2 0
5 1 0
6

University of Idaho $n = 1$ (6)

5) $a = 0$

$\langle 1, 0 \rangle$ "in there"

set $n = 5$

6) $a = 1$

$\langle 5, 1 \rangle$ not there

xmit $n = 5$

m	n	a
0	0	null
1	0	0
2	0	1
3	2	1
4	2	0
5	1	0
6	5	1

new entry

$\langle 0, 1 \rangle$ is at $n = 2$

University of Idaho

pg 24

source symbol	present n	present m	transmit	next n	dictionary entry
1	0	3		2	
1	2	3	2	2	2, 1
0	2	4	2	1	2, 0
0	1	5	1	1	1, 0
0	1	6		5	
1	5	6	5	2	5, 1
0	2	7		4	
1	4	7	4	2	4, 1
1	2	8		3	
0	3	8	3	1	3, 0
0	1	9		5	
1	5	9		6	
0	6	9	6	1	6, 0
1	1	10	1	2	1, 1
1	2	11		3	
1	3	11	3	2	3, 1
0	2	12		4	
0	4	12	4	1	4, 0
0	1	13		5	
1	5	13		6	
1	6	13	6	2	6, 1
1	2	14		3	
1	3	14		11	

The encoder's dictionary to this point is shown below.

dictionary address	dictionary entry
0	0, *null*
1	0, 0
2	0, 1
3	2, 1
4	2, 0
5	1, 0
6	5, 1
7	4, 1
8	3, 0
9	6, 0
10	1, 1
11	3, 1
12	4, 0
13	6, 1
14	no entry yet

At the decoder:

- we must also build the dictionary
- what we receive is not source symbols "a" but pointers m.

What does the decoder do?

1) must initialize its dictionary

m	n	a
0	0	null
1	0	0
2	0	1

University of Idaho

(10)

as the decoder receives "n" e, it must

1) build its dictionary
2) decode the transmission

Must do this in this order because its dictionary construction is "just in time to decode"

(11)

example

1) receive n=2

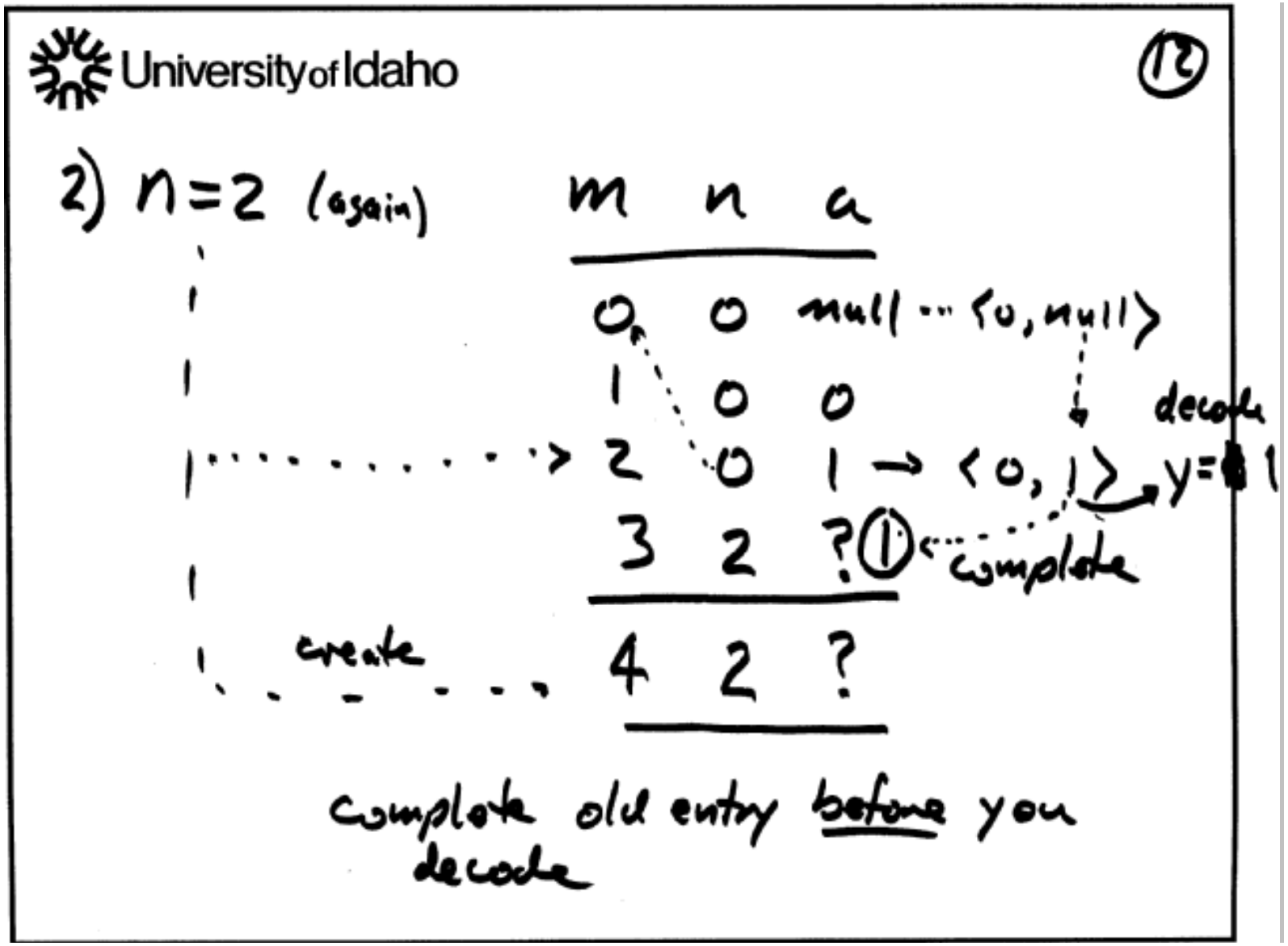
m	n	a	
0	0	null	→ <0, null>
1	0	0	
2	0	1	→ <0, 1>

Points ->

create ->

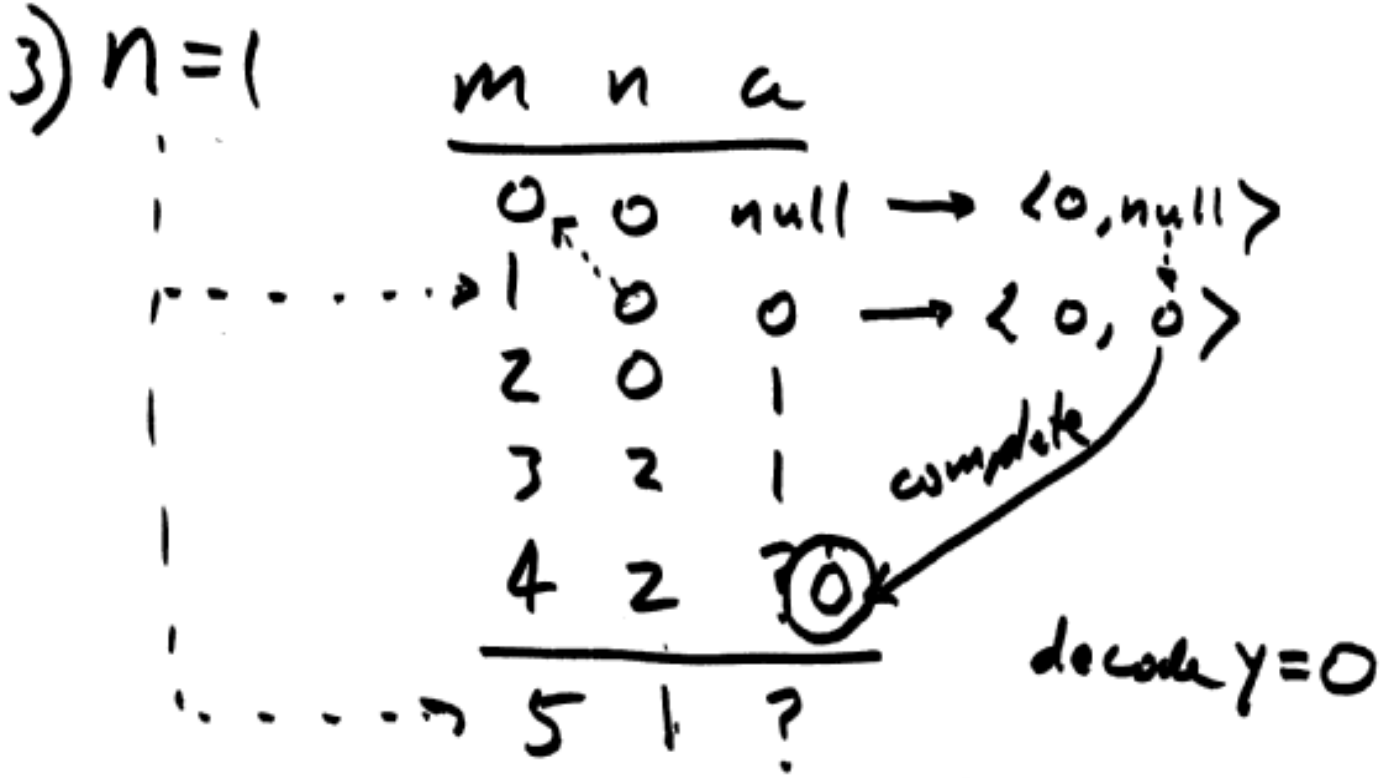
3	2	?

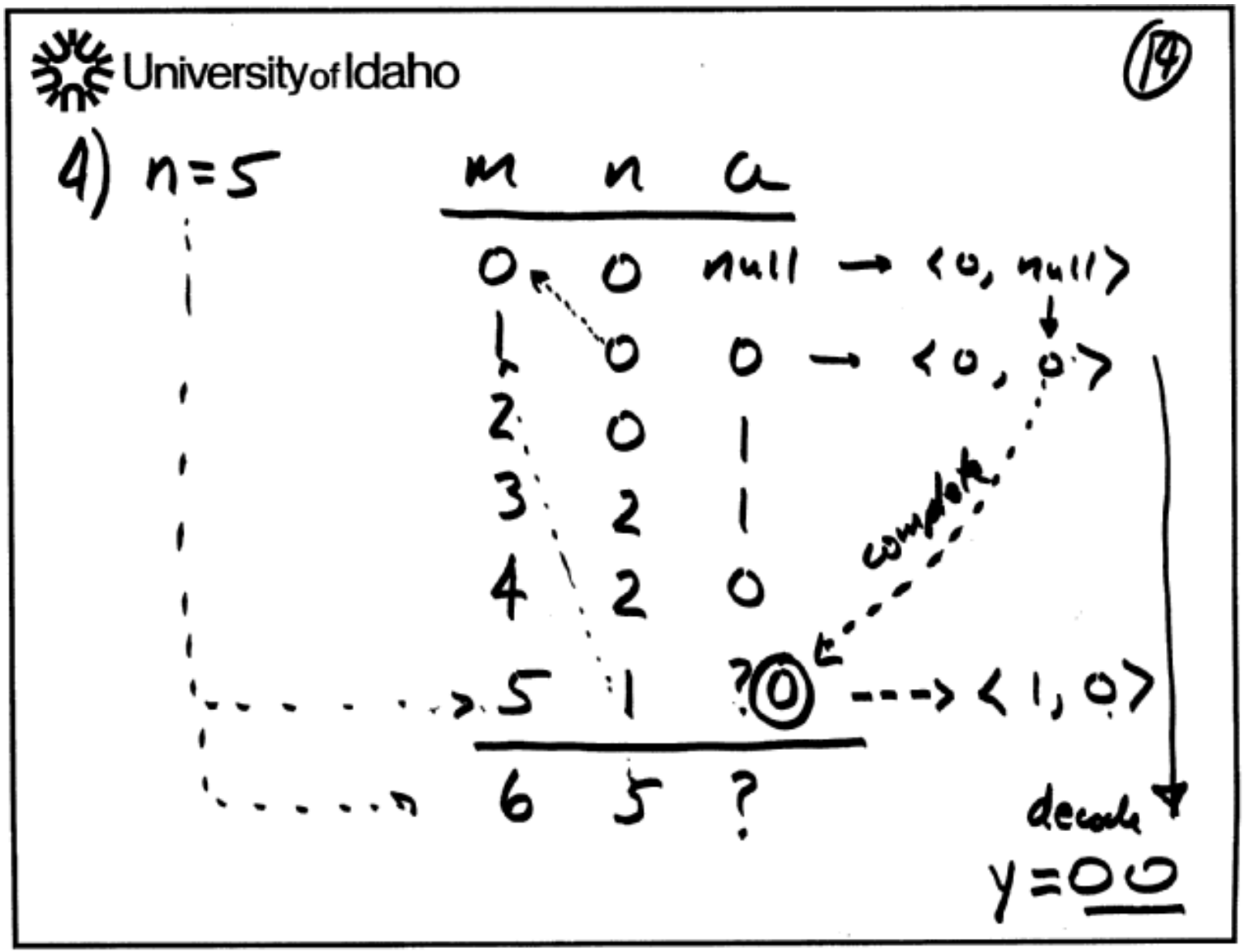
└ decode γ=1

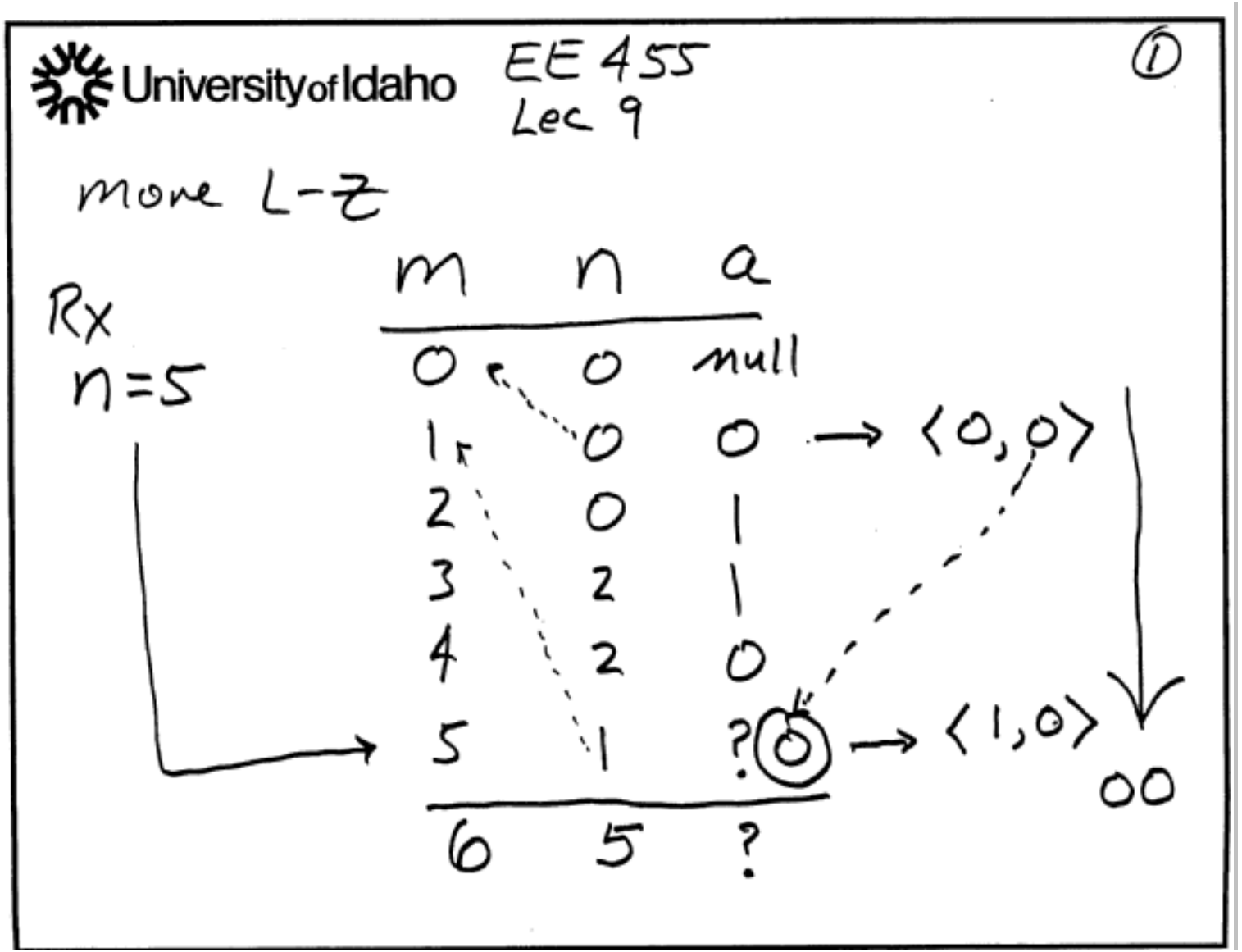

University of Idaho
12
2) n=2 (again)
m n a
0 0 null ... <0, null>
1 0 0
2 0 1 → <0, 1>
decode
y=
3 2 ?
complete
create
4 2 ?
complete old entry before you decode

University of Idaho

(13)

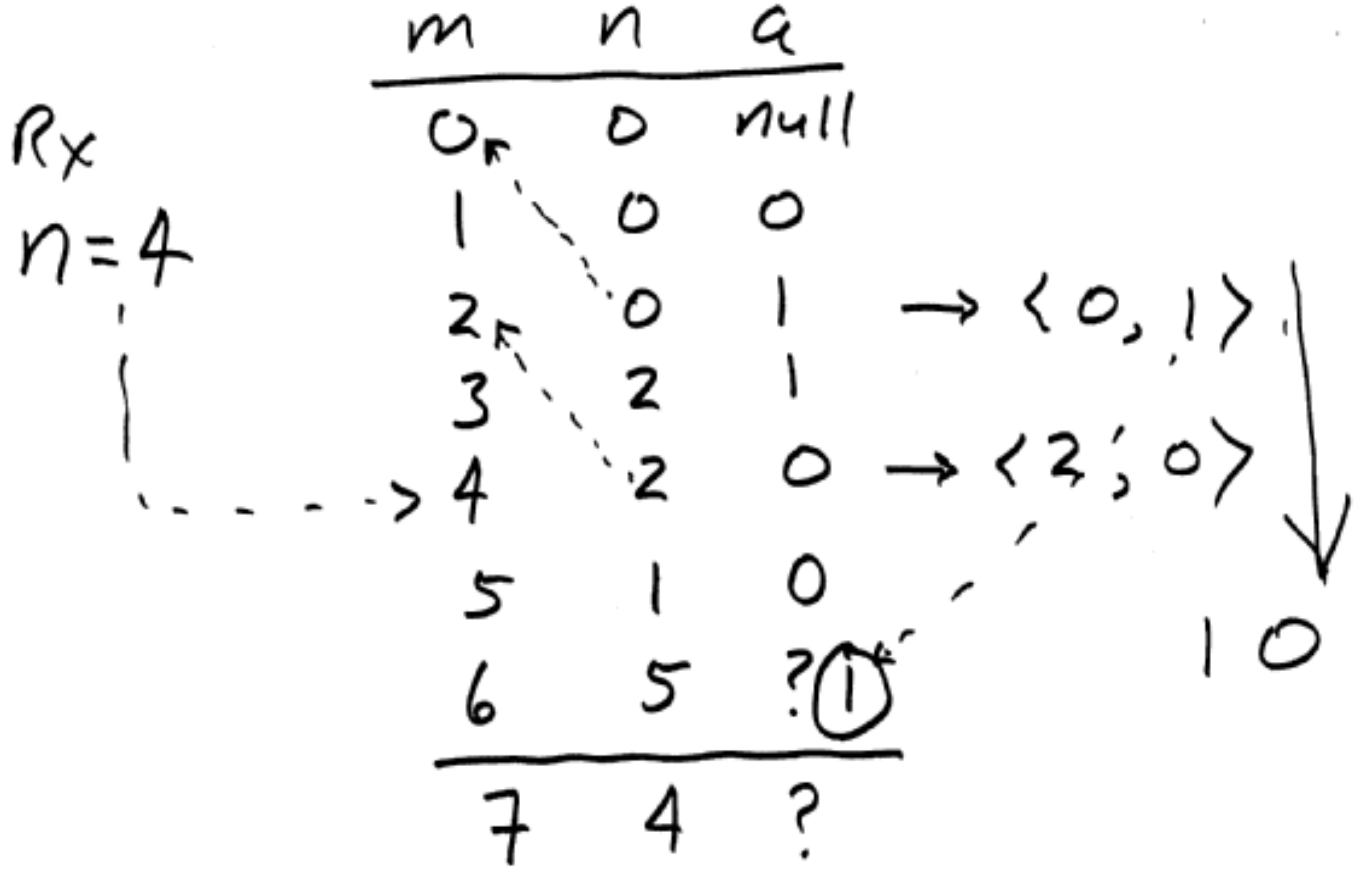



University of Idaho
14
4) n=5
m n a
0 0 null → <0, null>
1 0 0 → <0, 0>
2 0 1
3 2 1
4 2 0
5 1 ?0 ---> <1, 0>
6 5 ?
complete
decode
y=00


University of Idaho
EE 455
Lec 9
1
more L-Z
Rx
n=5
m n a
0 0 null
1 0 0 → <0,0>
2 0 1
3 2 1
4 2 0
5 1 ? → <1,0>
00
6 5 ?

University of Idaho

(2)



University of Idaho

$A = \{0, 1\}$

m	n	a
0	0	null
1	0	0
2	0	1

(3)

$n_i = 0$

0 0 | 1 1 | 0 0 0 | 1 0 | 1 1 x

n=1 n n n n

one comment regarding both adaptive source codes (e.g. adaptive Huffman, adaptive arithmetic) and dynamic dictionary codes (e.g. L-Z):

decoder must dynamically construct its decoding rules "on-the-fly"; ...

④

Therefore, the encoder must send enough information (over and above the message information) to enable the decoder to construct the decoding rules

Therefore, in the beginning these codes actually "decompress" rather than compress. Actual data compression sets in later.

These codes are effective for long messages.

End of Notes: