Huffman Coding

C.M. Liu

Perceptual Lab, College of Computer Science

National Chiao-Tung University

http://www.csie.nctu.edu.tw/~cmliu/Courses/Compression/

Office: EC538

(03)5731877

cmliu@cs.nctu.edu.tw

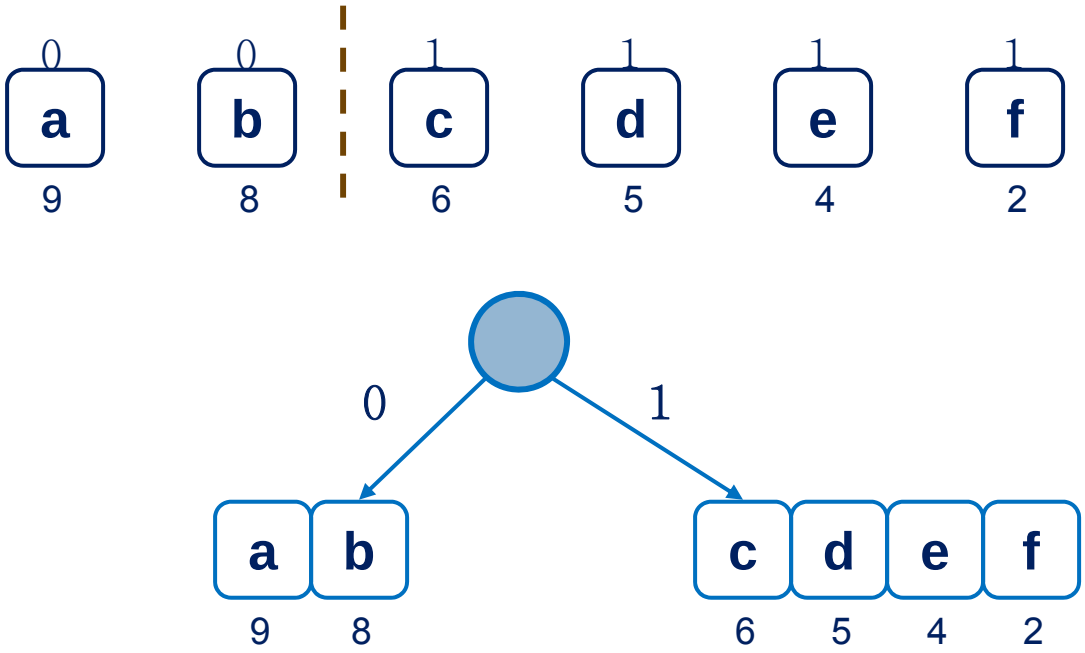
Overview

- Huffman Coding Algorithm
  - The procedure to build Huffman codes.
  - Extended Huffman Codes
- Adaptive Huffman Coding
  - Update Procedure
  - Decoding Procedure
- Golomb Codes

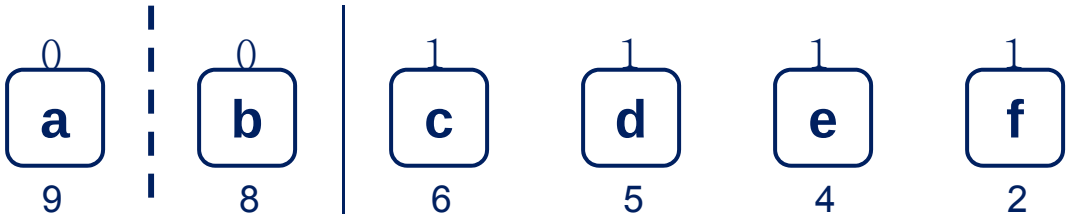
Shannon-Fano Coding

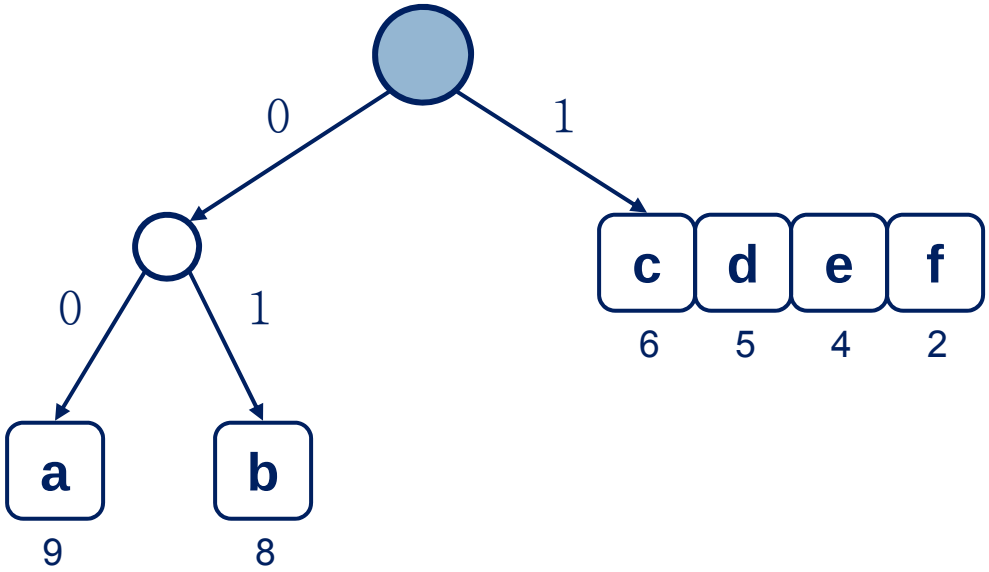
- The first code based on Shannon's theory
  - Suboptimal (it took a graduate student to fix it!)
- Algorithm
  - Start with empty codes
  - Compute frequency statistics for all symbols
  - Order the symbols in the set by frequency
  - Split the set to minimize* difference
  - Add '0' to the codes in the first set and '1' to the rest
  - Recursively assign the rest of the code bits for the two subsets, until sets cannot be split.

Shannon-Fano Coding (2)

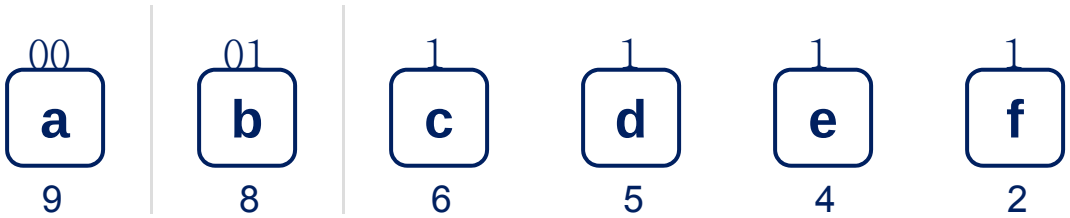


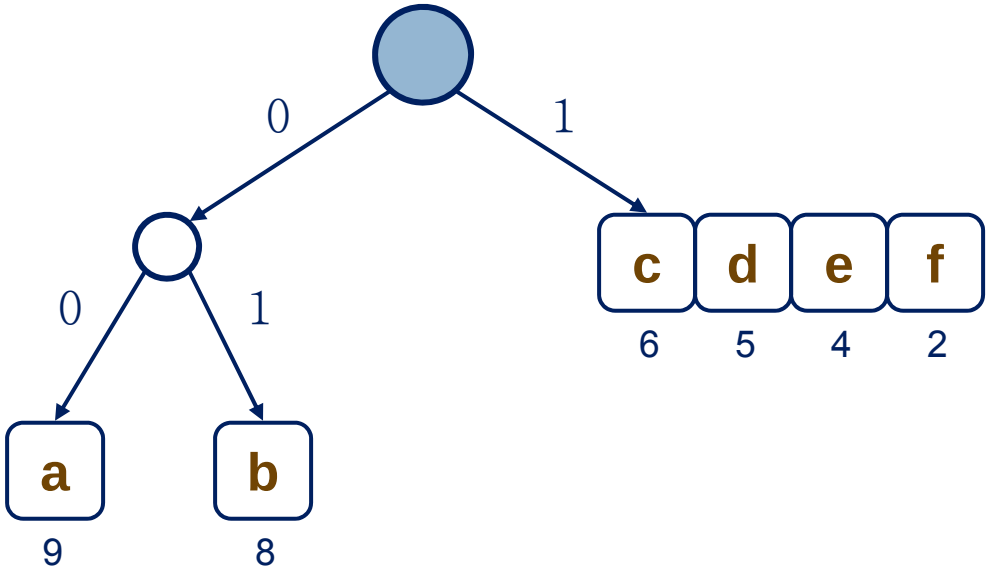
Shannon-Fano Coding (3)



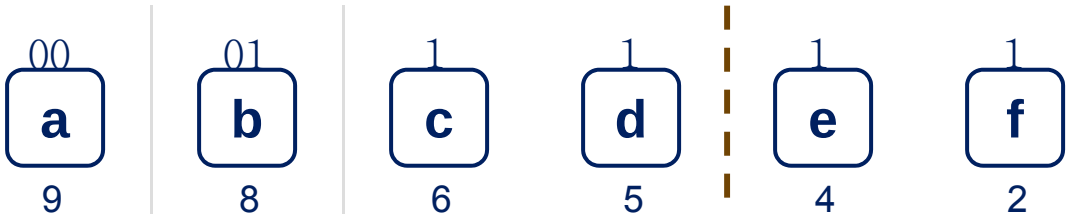


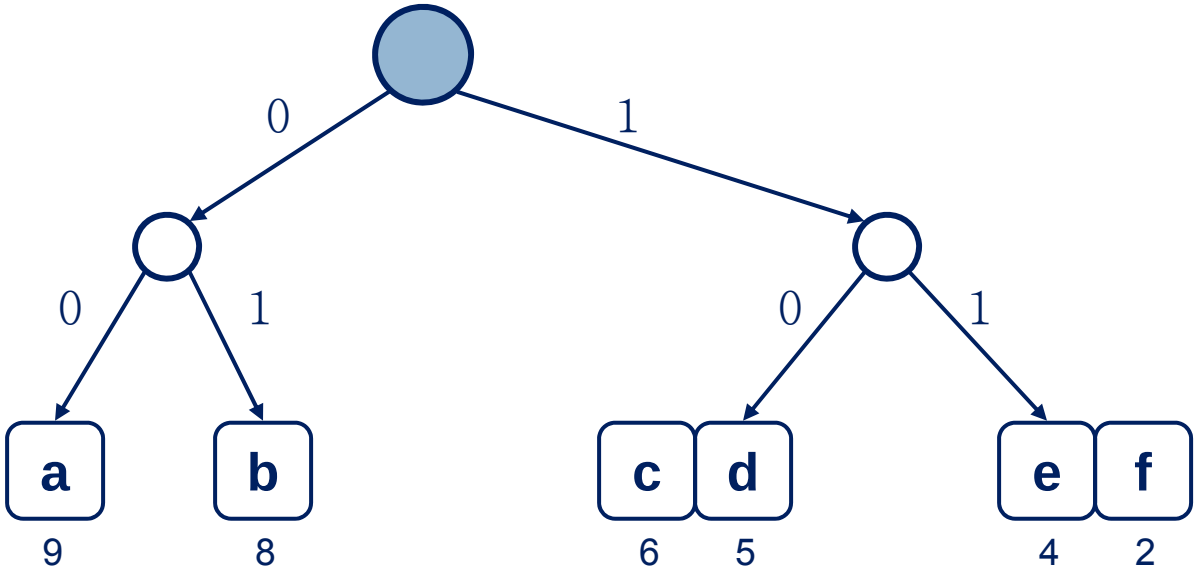
Shannon-Fano Coding (4)



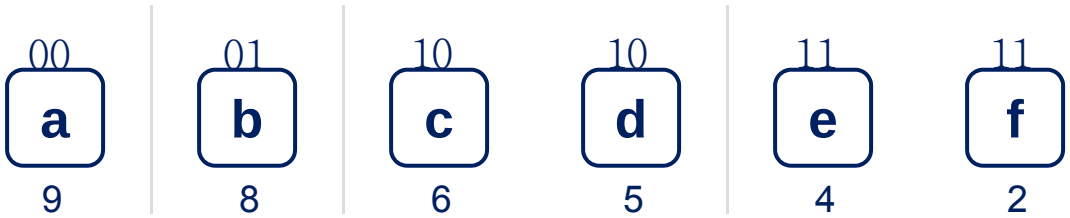


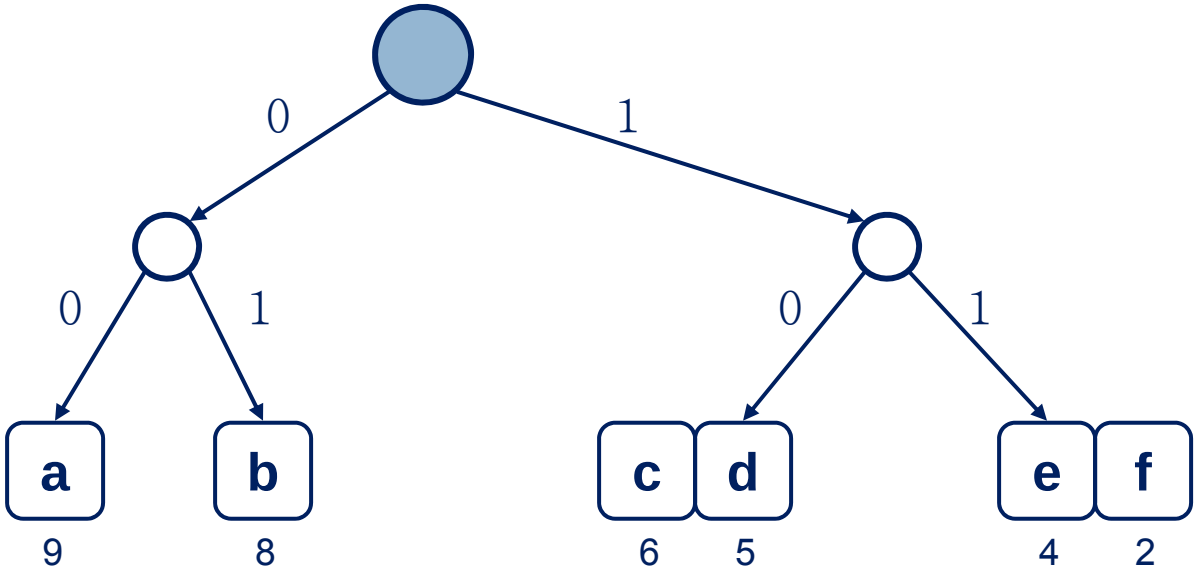
Shannon-Fano Coding (5)



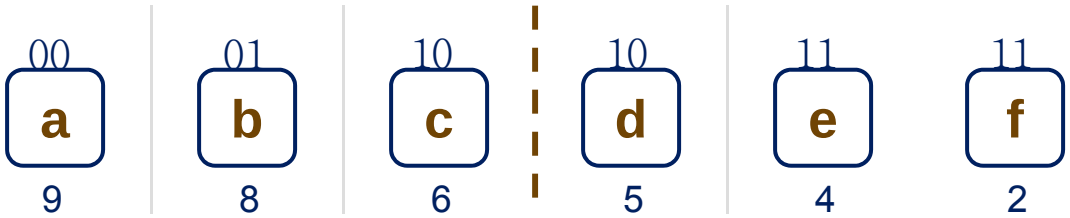


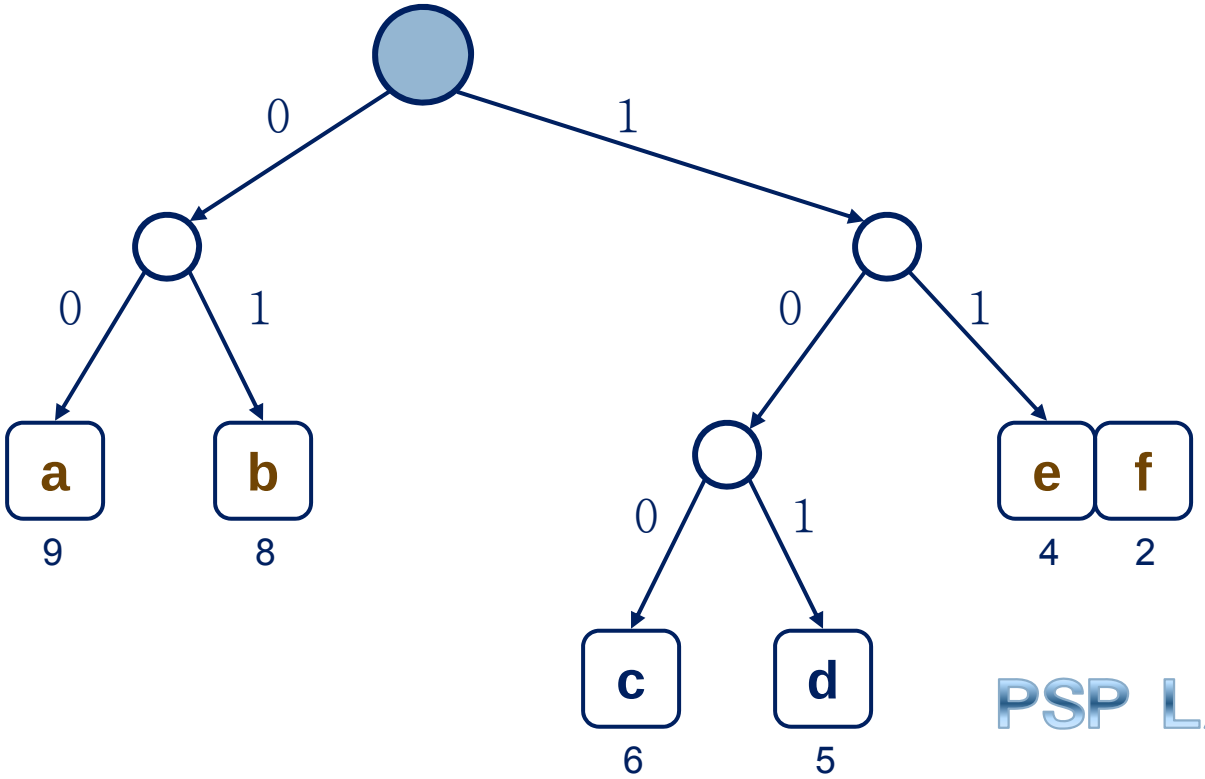
Shannon-Fano Coding (6)



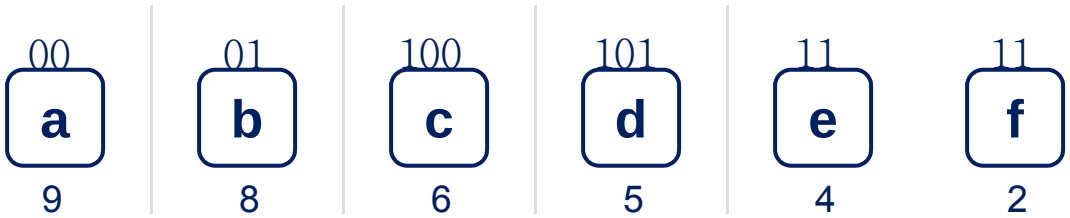


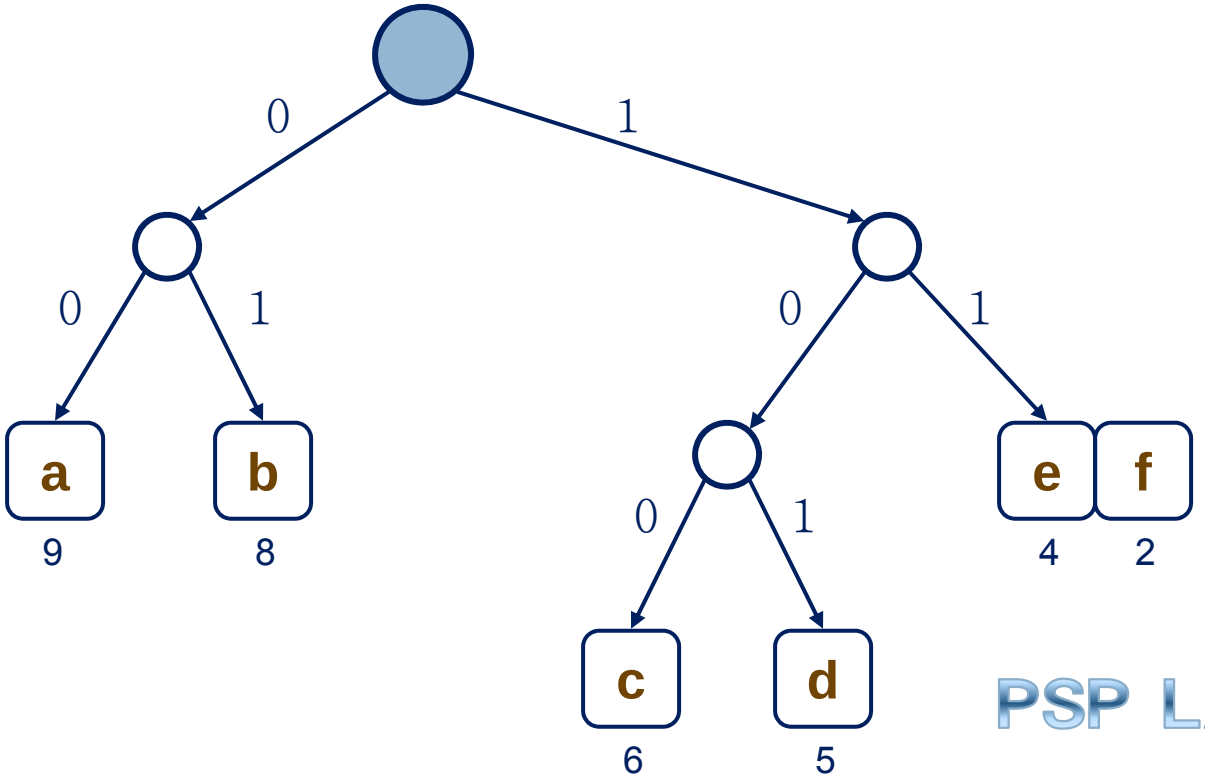
Shannon-Fano Coding (7)



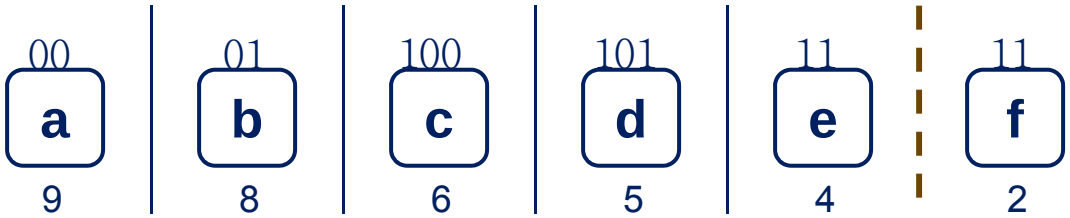


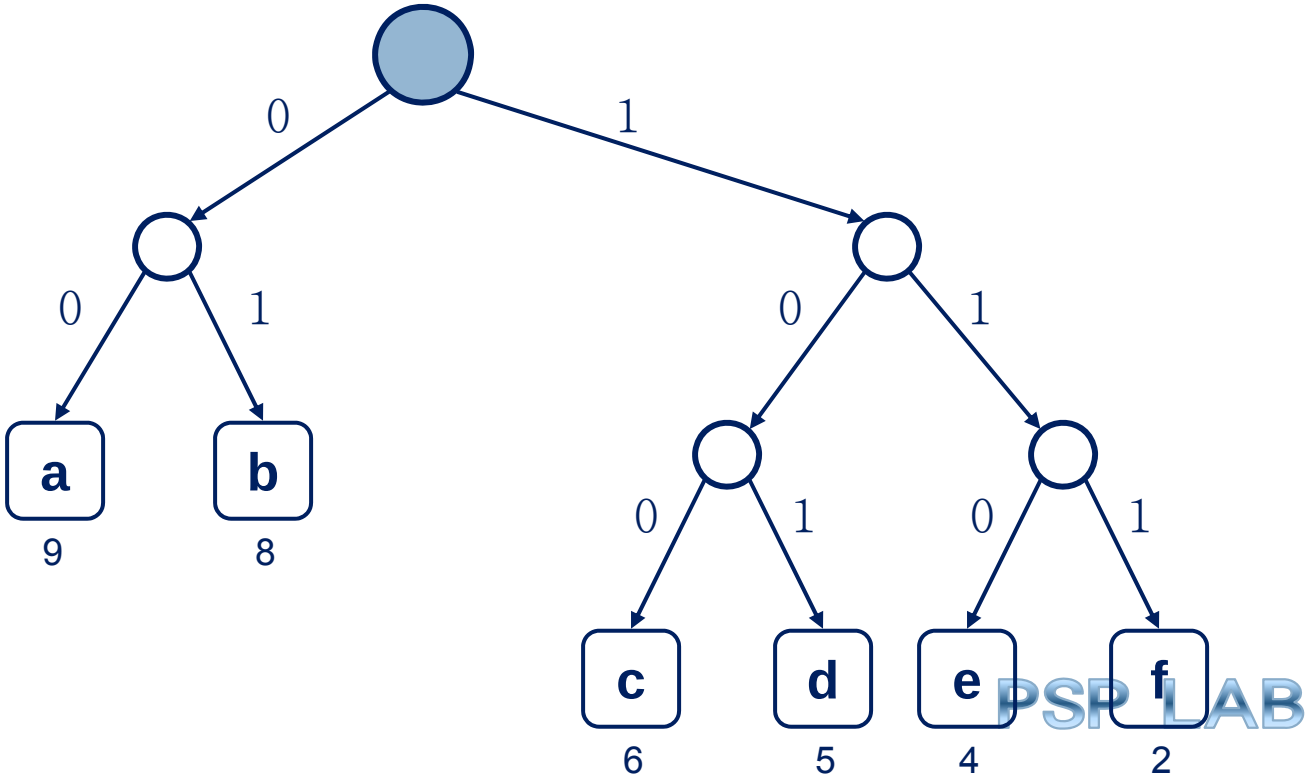
Shannon-Fano Coding (8)



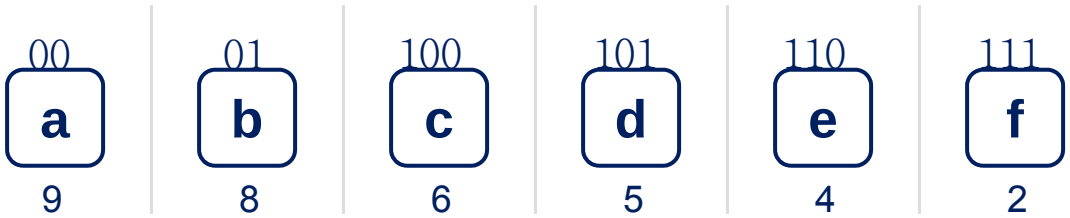


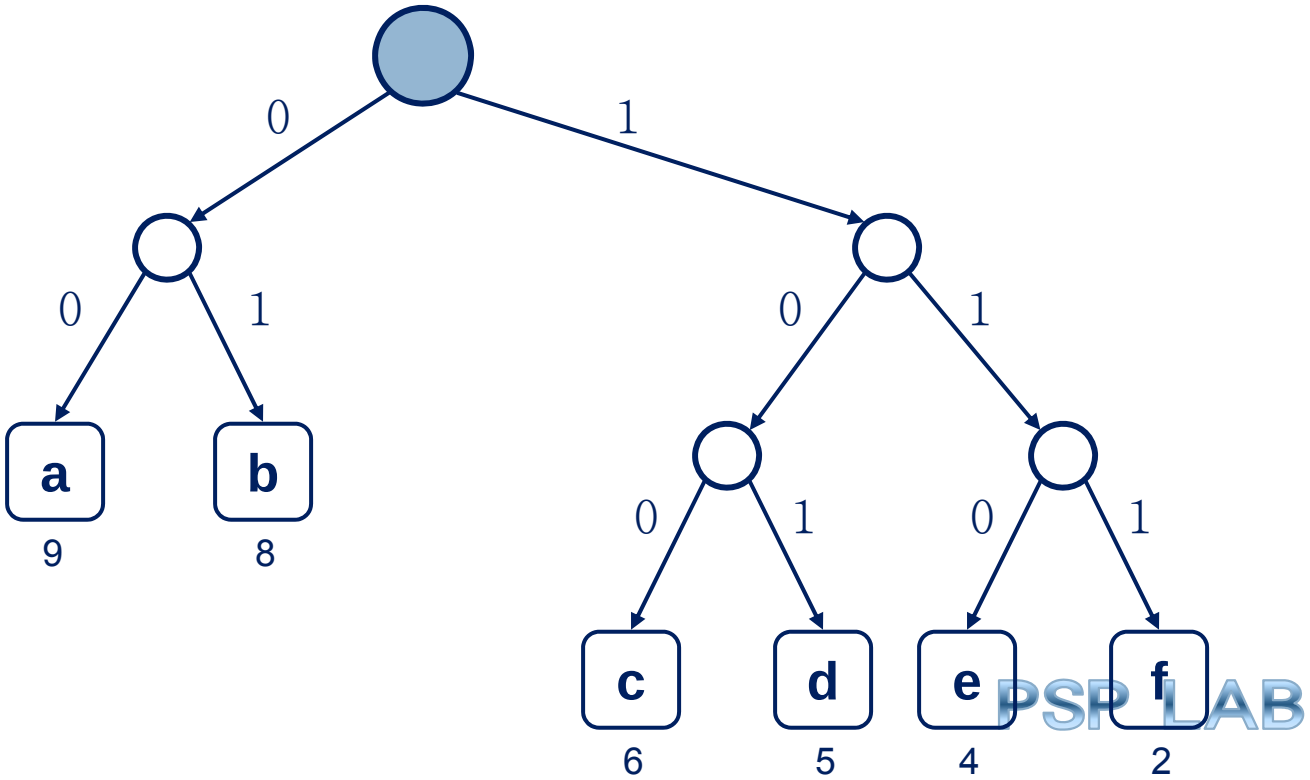
Shannon-Fano Coding (9)





Shannon-Fano Coding (10)





Shannon-Fano Coding: Remarks

- Shannon-Fano does not always produce optimal prefix codes;
  - the set of probabilities {0.35, 0.17, 0.17, 0.16, 0.15}
- Huffman coding is almost as computationally simple and produces prefix codes that always achieve the lowest expected code word length, under the constraints that each symbol is represented by a code formed of an integral number of bits.
- Symbol-by-symbol Huffman coding is only optimal if the probabilities of the symbols are independent and are some power of a half, i.e. $1/2^n$

Optimum Prefix Codes

- Key observations on optimal codes
  1. Symbols that occur more frequently will have shorter codewords
  2. The two least frequent symbols will have the same length
- Proofs
  1. Assume the opposite—code is clearly sub-optimal
  2. Assume the opposite
     - Let X, Y be the least frequent symbols &
     - $|code(X)| = k$, $|code(Y)| = k+1$

     Then
     - by unique decodability (UD), code(X) cannot be a prefix for code(Y)
     - also, all other codes are shorter

     ➔ Dropping the last bit of $|code(Y)|$ would generate a new, shorter, uniquely decodable code

     !!! This contradicts optimality assumption !!!

Huffman Coding

- David Huffman (1951)
  - Grad student of Robert M. Fano (MIT)
    - Term paper(!)
- Explained by example

Letter	Code	Probability	Set	Set Prob
a		0.2		
b		0.4		
c		0.2		
d		0.1		
e		0.1		

Huffman Coding by Example

- Init: Create a set out of each letter

Letter	Code	Probability	Set	Set Prob
a		0.2	a	0.2
b		0.4	b	0.4
c		0.2	c	0.2
d		0.1	d	0.1
e		0.1	e	0.1

Huffman Coding by Example

1. Sort sets according to probability (lowest first)

Letter	Code	Probability	Set	Set Prob
a		0.2	d	0.1
b		0.4	e	0.1
c		0.2	a	0.2
d		0.1	c	0.2
e		0.1	b	0.4

Huffman Coding by Example

2. Insert prefix ‘1’ into the codes of top set letters

Letter	Code	Probability	Set	Set Prob
a		0.2	d	0.1
b		0.4	e	0.1
c		0.2	a	0.2
d	**1**	0.1	c	0.2
e		0.1	b	0.4

Huffman Coding by Example

3. Insert prefix ‘0’ into the codes of the second set letters

Letter	Code	Probability	Set	Set Prob
a		0.2	d	0.1
b		0.4	e	0.1
c		0.2	a	0.2
d	1	0.1	c	0.2
e	**0**	0.1	b	0.4

Huffman Coding by Example

4. Merge the top two sets

Letter	Code	Probability	Set	Set Prob
a		0.2	de	0.2
b		0.4	a	0.2
c		0.2	c	0.2
d	1	0.1	d	0.4
e	0	0.1		

Huffman Coding by Example

1. Sort sets according to probability (lowest first)

Letter	Code	Probability	Set	Set Prob
a		0.2	de	0.2
b		0.4	a	0.2
c		0.2	c	0.2
d	1	0.1	b	0.4
e	0	0.1		

Huffman Coding by Example

2. Insert prefix ‘1’ into the codes of top set letters

Letter	Code	Probability	Set	Set Prob
a		0.2	de	0.2
b		0.4	a	0.2
c		0.2	c	0.2
d	**1**1	0.1	b	0.4
e	**1**0	0.1		

Huffman Coding by Example

3. Insert prefix ‘0’ into the codes of the second set letters

Letter	Code	Probability	Set	Set Prob
a	**0**	0.2	de	0.2
b		0.4	a	0.2
c		0.2	c	0.2
d	11	0.1	b	0.4
e	10	0.1		

Huffman Coding by Example

4. Merge the top two sets

Letter	Code	Probability	Set	Set Prob
a	0	0.2	dea	0.4
b		0.4	c	0.2
c		0.2	b	0.4
d	11	0.1		
e	10	0.1		

Huffman Coding by Example

1. Sort sets according to probability (lowest first)

Letter	Code	Probability	Set	Set Prob
a	0	0.2	c	0.2
b		0.4	dea	0.4
c		0.2	b	0.4
d	11	0.1		
e	10	0.1		

Huffman Coding by Example

2. Insert prefix ‘1’ into the codes of top set letters

Letter	Code	Probability	Set	Set Prob
a	0	0.2	c	0.2
b		0.4	dea	0.4
c	**1**	0.2	b	0.4
d	11	0.1		
e	10	0.1		

Huffman Coding by Example

3. Insert prefix ‘0’ into the codes of the second set letters

Letter	Code	Probability	Set	Set Prob
a	**0**0	0.2	c	0.2
b		0.4	dea	0.4
c	1	0.2	b	0.4
d	**0**11	0.1		
e	**0**01	0.1		

Huffman Coding by Example

4. Merge the top two sets

Letter	Code	Probability	Set	Set Prob	Set Prob
a	00	0.2	cdea	0.6	0.2
b		0.4	b	0.4	0.4
c	1	0.2			0.4
d	011	0.1			
e	010	0.1			

Huffman Coding by Example

1. Sort sets according to probability (lowest first)

Letter	Code	Probability	Set	Set Prob
a	00	0.2	b	0.4
b		0.4	cdea	0.6
c	1	0.2		
d	011	0.1		
e	010	0.1		

Huffman Coding by Example

2. Insert prefix ‘1’ into the codes of top set letters

Letter	Code	Probability	Set	Set Prob
a	00	0.2	b	0.4
b	1	0.4	cdea	0.6
c	1	0.2		
d	011	0.1		
e	010	0.1		

Huffman Coding by Example

3. Insert prefix ‘0’ into the codes of the second set letters

Letter	Code	Probability	Set	Set Prob
a	000	0.2	b	0.4
b	1	0.4	cdea	0.6
c	01	0.2		
d	0011	0.1		
e	0010	0.1		

Huffman Coding by Example

4. Merge the top two sets

Letter	Code	Probability	Set	Set Prob
a	000	0.2	abcde	1.0
b	1	0.4		
c	01	0.2		
d	0011	0.1		
e	0010	0.1		

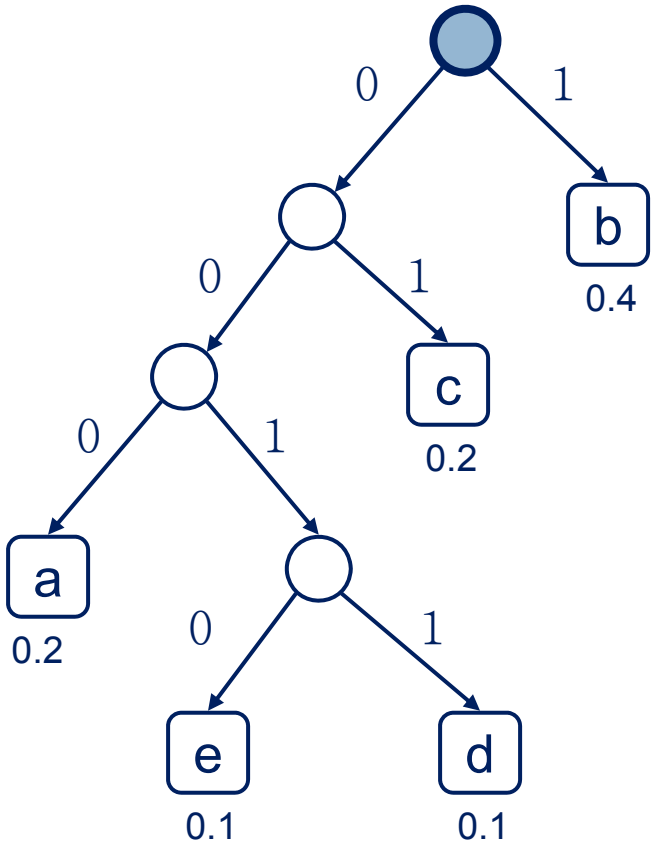
- The END

Example Summary

- Average code length
  - $l = 0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$ bits/symbol
- Entropy
  - $H = \sum_{s=a..e} P(s) \log_2 P(s) = 2.122$ bits/symbol
- Redundancy
  - $l - H = 0.078$ bits/symbol

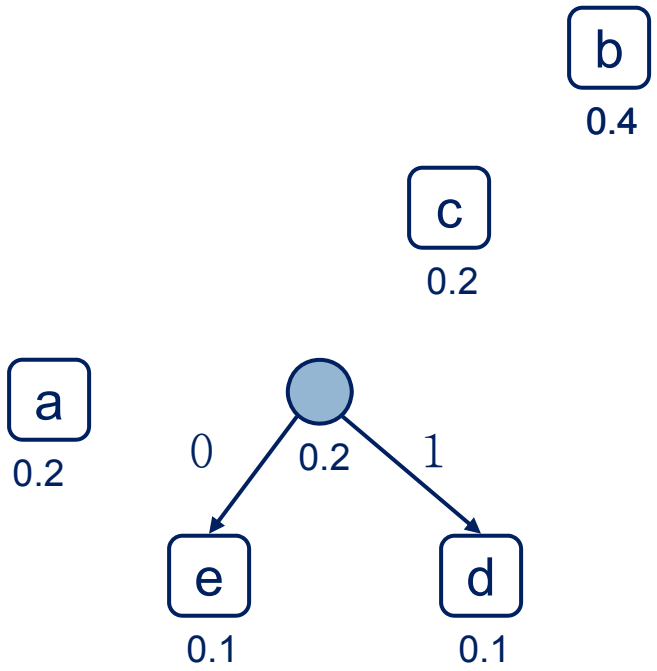
Huffman Tree

Letter	Code
a	000
b	1
c	01
d	0011
e	0010



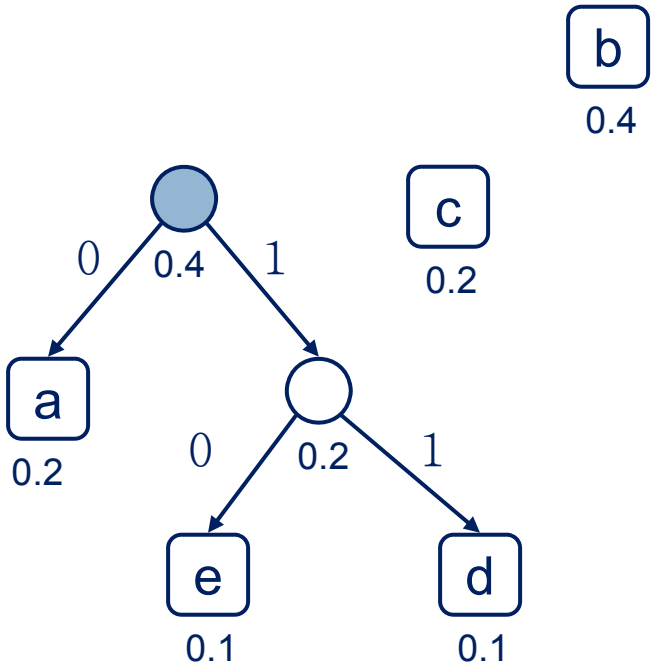
Building a Huffman Tree

Letter	Code
a	
b	
c	
d	1
e	0



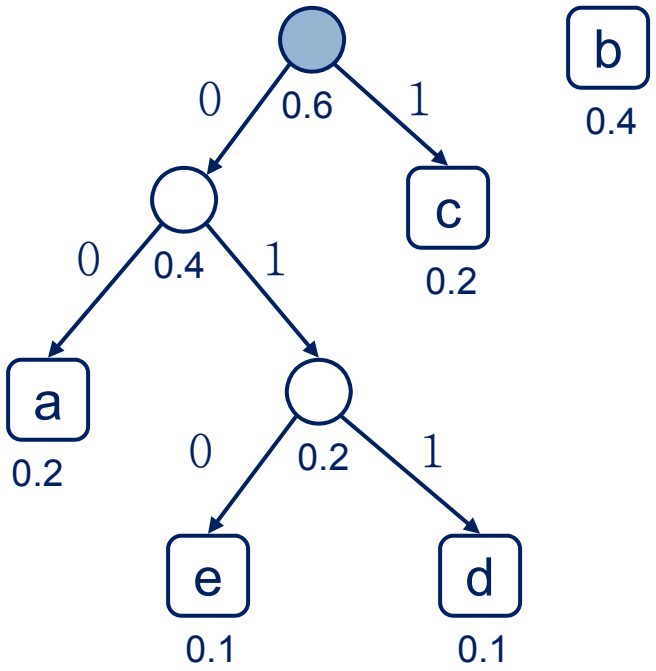
Building a Huffman Tree

Letter	Code
a	0
b	
c	
d	11
e	10



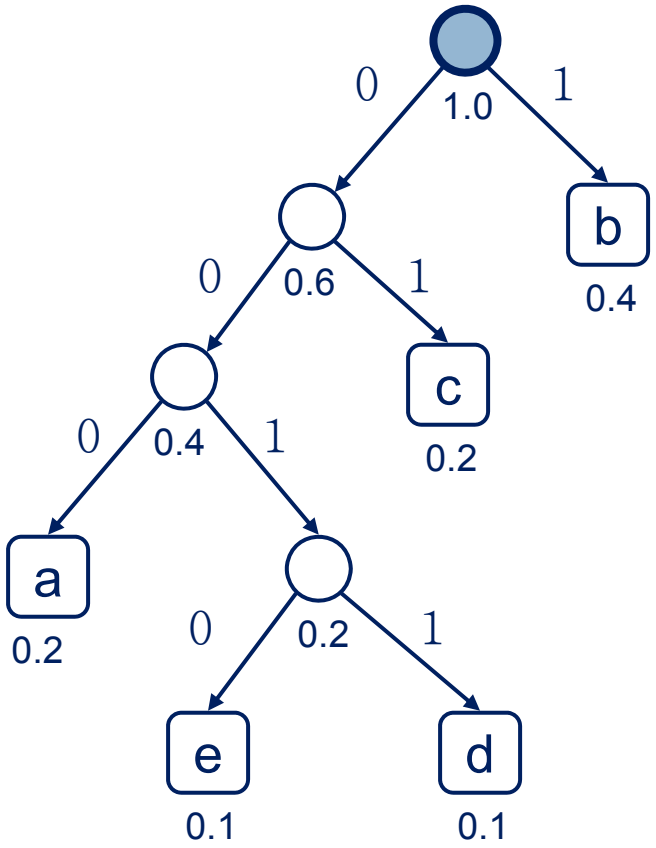
Building a Huffman Tree

Letter	Code
a	00
b	
c	1
d	011
e	010



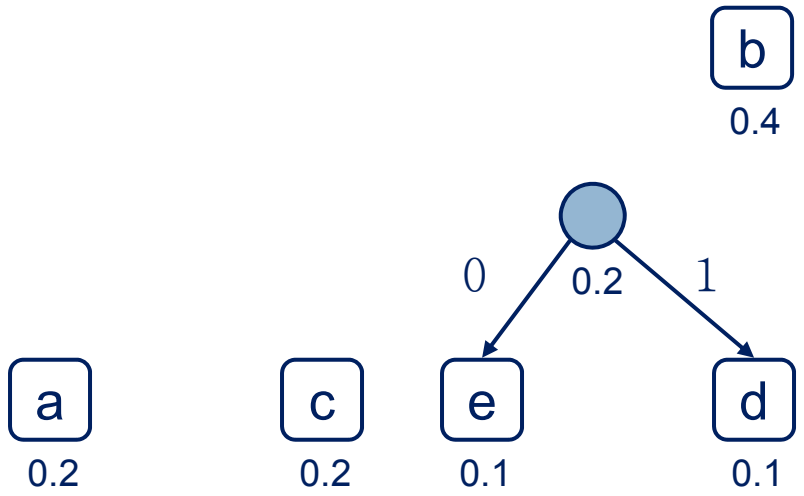
Building a Huffman Tree

Letter	Code
a	000
b	1
c	01
d	0011
e	0010



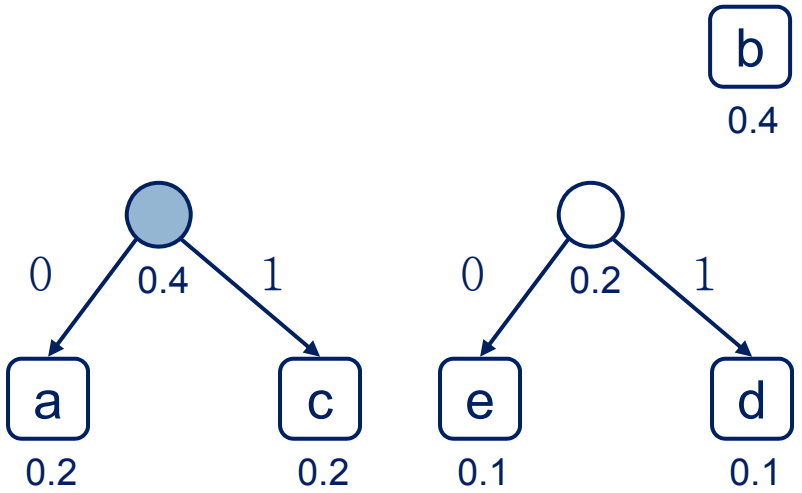
An Alternative Huffman Tree

Letter	Code
a	
b	
c	
d	1
e	0



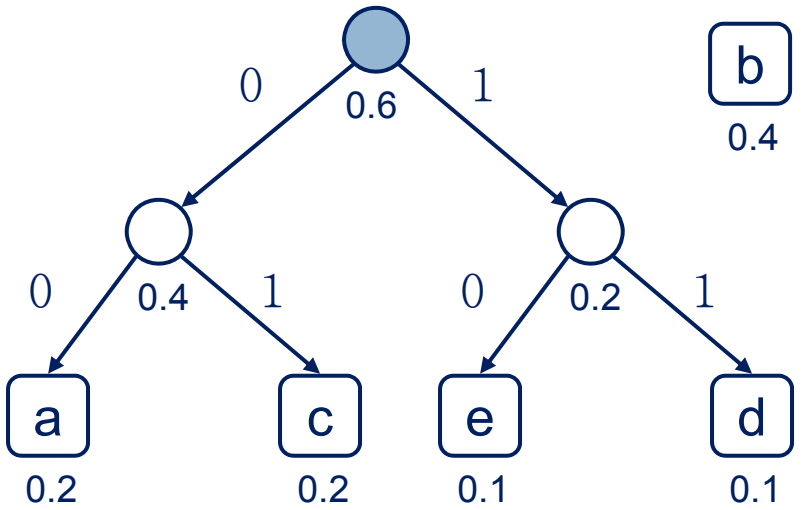
An Alternative Huffman Tree

Letter	Code
a	0
b	
c	1
d	1
e	0



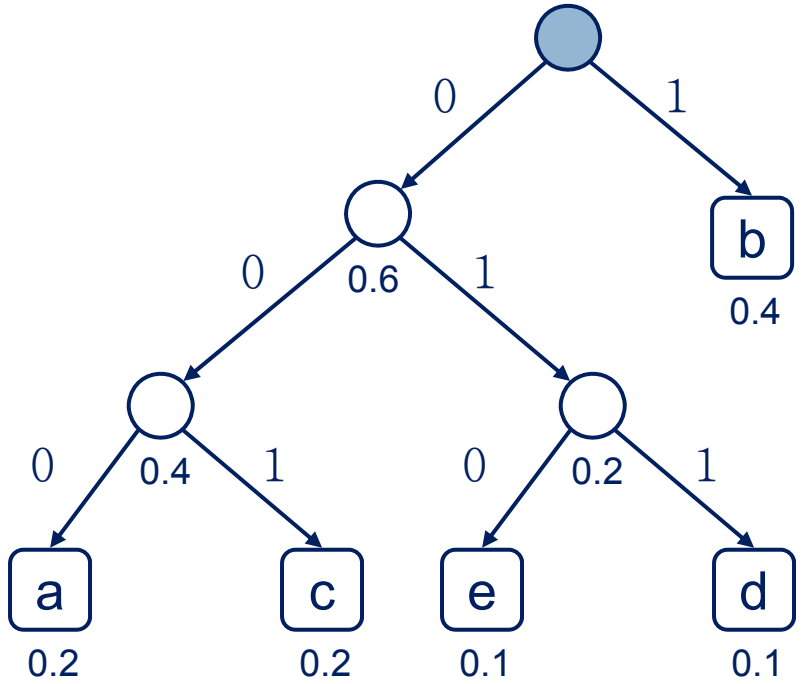
An Alternative Huffman Tree

Letter	Code
a	00
b	
c	01
d	11
e	10



An Alternative Huffman Tree

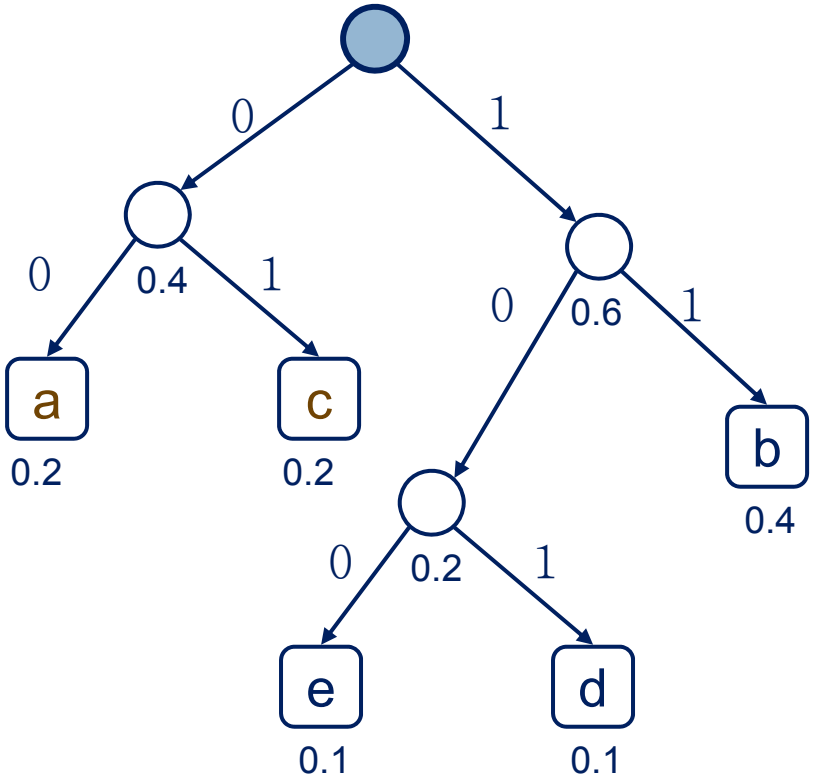
Letter	Code
a	000
b	1
c	001
d	011
e	010



- Average code length
  - l = 0.4x1 + (0.2 + 0.2 + 0.1 + 0.1)x3= 2.2 bits/symbol

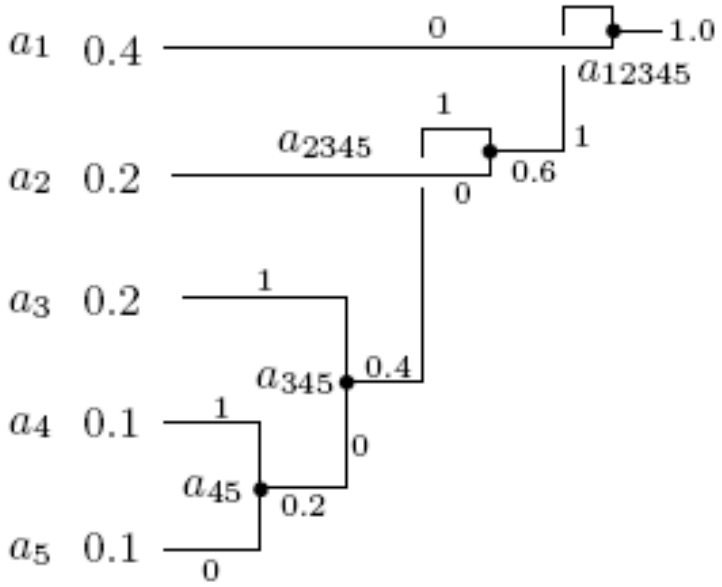
Yet Another Tree

Letter	Code
a	00
b	11
c	01
d	101
e	100

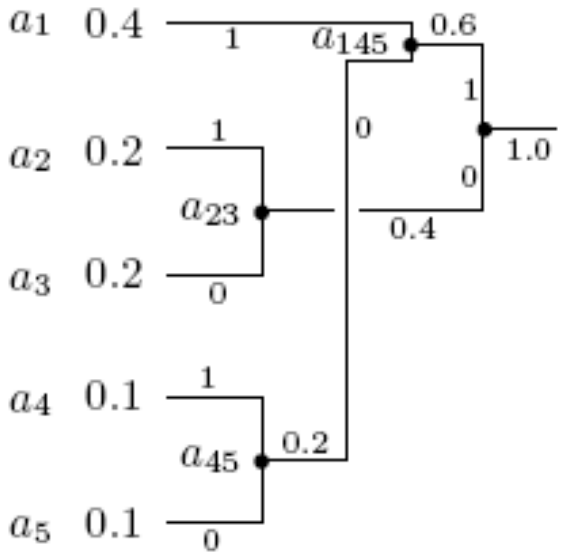


- Average code length
  - l = 0.4x2+ (0.2 + 0.2)x2 + (0.1 + 0.1)x3= 2.2 bits/symbol

Design Examples



(a)



(b)

PSP LAB

Min Variance Huffman Trees

- Huffman codes are not unique
  - All versions yield the same ***average*** length
- Which one should we choose?
  - The one with the minimum variance in codeword lengths
    - I.e. with the minimum height tree
- Why?
  - It will ensure the least amount of variability in the encoded stream
- How to achieve it?
  - During sorting, break ties by placing smaller sets higher
    - Alternatively, place newly merged sets as low as possible

Extended Huffman Codes

- Consider the source:
  - $\mathcal{A}$ = {a, b, c}, P(a) = 0.8, P(b) = 0.02, P(c) = 0.18
  - ***H*** = 0.816 bits/symbol
- Huffman code:

a	0
b	11
c	10

  - *l* = 1.2 bits/symbol
  - Redundancy = 0.384 b/sym (47%!)
- Q: Could we do better?

Extended Huffman Codes (2)

- Idea
  - Consider encoding ***sequences of two letters*** as opposed to single letters

Letter	Probability	Code
aa	0.6400	0
ab	0.0160	10101
ac	0.1440	11
ba	0.0160	101000
bb	0.0004	10100101
bc	0.0036	1010011
ca	0.1440	100
cb	0.0036	10100100
cc	0.0324	1011

$l = 1.7228/2 =$ **0.8614**

Red. = **0.0045** bits/symbol

Extended Huffman Codes (3)

- The idea can be extended further
  - Consider all possible n^m sequences (we did 3^2)
- In theory, by considering more sequences we can improve the coding
- In reality, the exponential growth of the alphabet makes this impractical
  - E.g., for length 3 ASCII seq.: $256^3 = 2^{24} = 16M$
- Most sequences would have zero frequency

➔ Other methods are needed

Adaptive Huffman Coding

- Problem
  - Huffman requires probability estimates
  - This could turn it into a two-pass procedure:
    1. Collect statistics, generate codewords
    2. Perform actual encoding
  - Not practical in many situations
    - E.g. compressing network transmissions
- Theoretical solution
  - Start with equal probabilities
  - Based on the first ***k*** symbol statistics ($k = 1, 2, \ldots$) regenerate codewords and encode $k+1^{st}$ symbol
- Too expensive in practice

Adaptive Huffman Coding (2)

- Basic idea
  - Alphabet $\mathcal{A} = \{a_1, \ldots, a_n\}$
  - Pick a fixed default binary codes for all symbols
  - Start with an empty Huffman tree
  - Read symbol **s** from source
    - If NYT(s) // **N**ot **Y**et **T**ransmitted
      - Send NYT, default(s)
      - Update tree (and keep it Huffman)
    - Else
      - Send codeword for **s**
      - Update tree
  - Until done
- Notes:
  - Codewords will change as a function of symbol frequencies
  - Encoder & decoder follow the same procedure so they stay in sync

Adaptive Huffman Tree

- Tree has at most $2n$ - 1 nodes
- Node attributes
  - ***symbol***, ***left***, ***right***, ***parent***, ***siblings***, ***leaf***
  - ***weight***
    - If x_k is leaf then ***weight***(x_k) = frequency of ***symbol***(x_k)
    - Else x_k = ***weight***(***left***(x_k)) + ***weight***(***right***(x_k))
  - ***id***, assigned as follows:
    - If ***weight***$(x_1) \leq$ ***weight***$(x_2) \leq \ldots \leq$ ***weight***(x_{2n-1}) then
    - ***id***$(x_1) \leq$ ***id***$(x_2) \leq \ldots \leq$ ***id***(x_{2n-1})
    - Also, ***parent***(x_{2k-1}) = ***parent***(x_{2k}), for $1 \leq k \leq n$
      - *Sibling property*

Updating the Tree

- Assign ***id***(root) = 2*n*-1, *weight*(NYT) = 0
- Start with an NYT node
- Whenever a new symbols is seen, a new node is formed by splitting the NYT
- Maintaining sibling property
  - Whenever node ***x*** is updated
    - Repeat
      - If ***weight***(*x*) < ***weight***(*y*), for all *y* ∈ *siblings*(*x*)
        - ***weight***(*x*)++
        - exit
      - Else

        swap(x, z), where z rightmost sibling: weight(x) == weight(z)

        weight(x)++

        x = ***parent***(*x*)
    - Until *x* == root

Adaptive Huffman *Encoding*

Input: aardvark

Output:

Symbol	Code
NYT	
a	
r	
d	
v	
k	

slightly more efficient default codes are possible
(4-/5-bit combination)

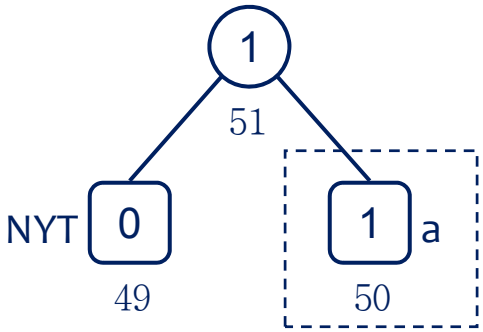
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: **a**ardvark

Output: **00000**

Symbol	Code
NYT	0
a	1
r	
d	
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

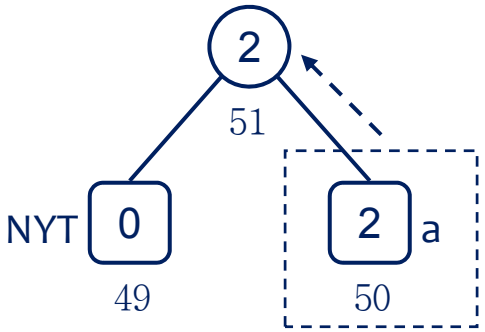
LAB

Adaptive Huffman *Encoding*

Input: a**a**rdvark

Output: 00000**1**

Symbol	Code
NYT	0
a	1
r	
d	
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

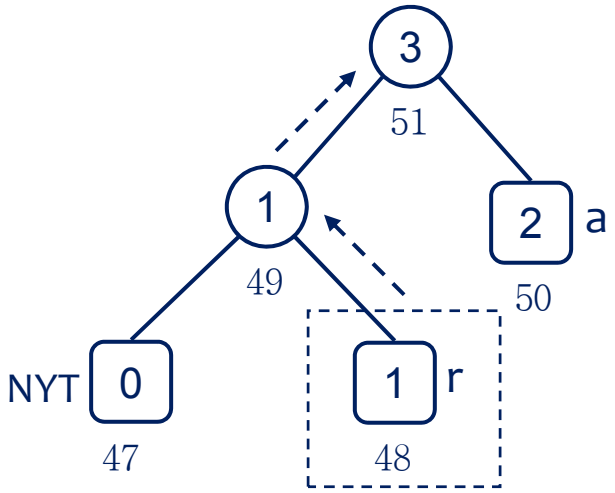
LAB

Adaptive Huffman *Encoding*

Input: aardvark

Output: 000001010001

Symbol	Code
NYT	00
a	1
r	01
d	
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

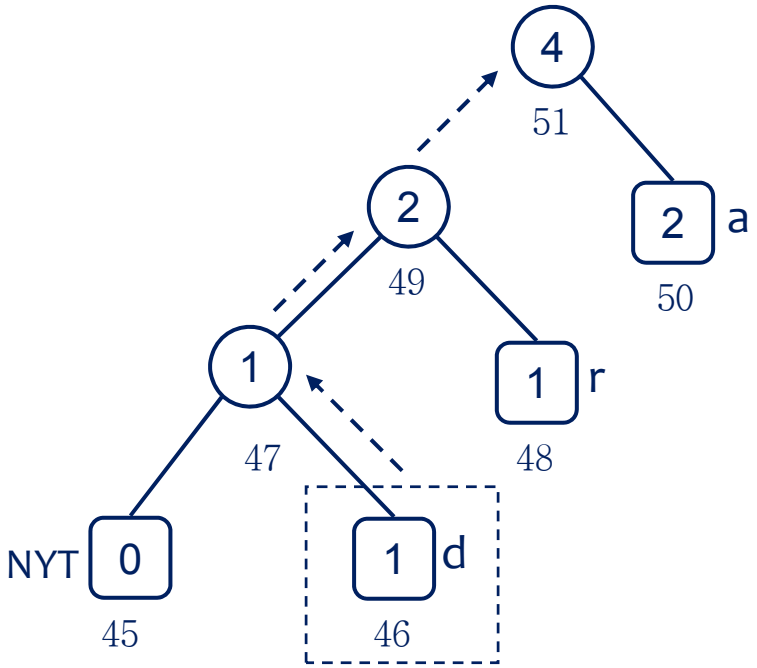
LAB

Adaptive Huffman *Encoding*

Input: aar**d**vark

Output: 000001010001**0000011**

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



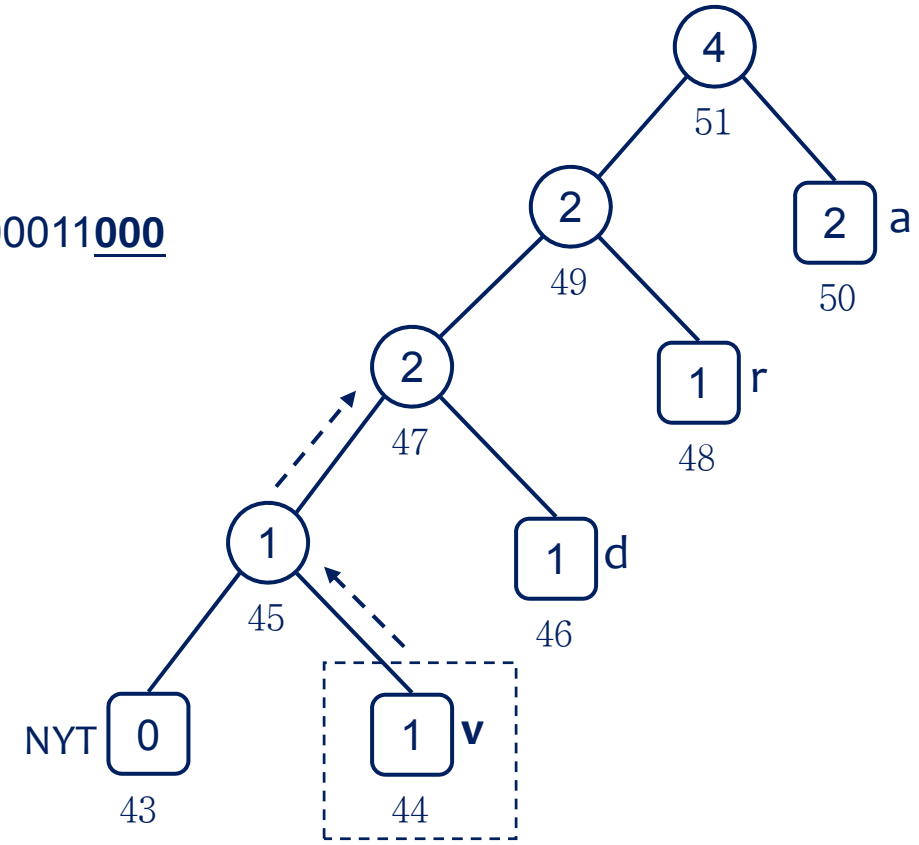
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: aard**v**ark

Output: 000001010001000011**000**

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

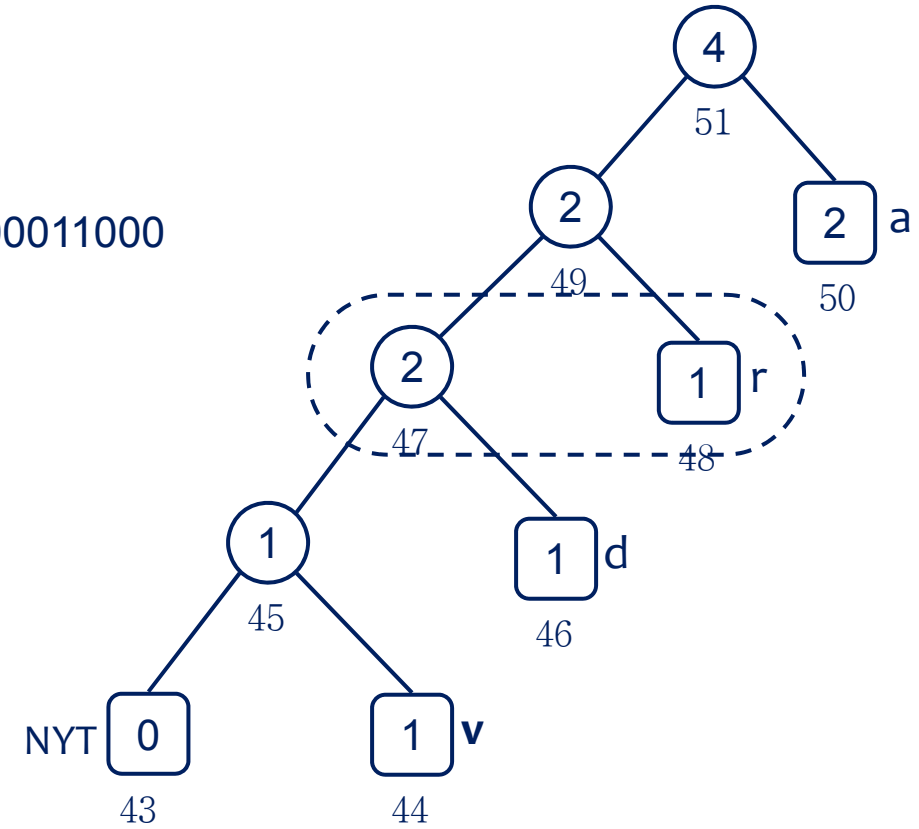
LAB

Adaptive Huffman *Encoding*

Input: aardvark

Output: 000001010001000011000

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



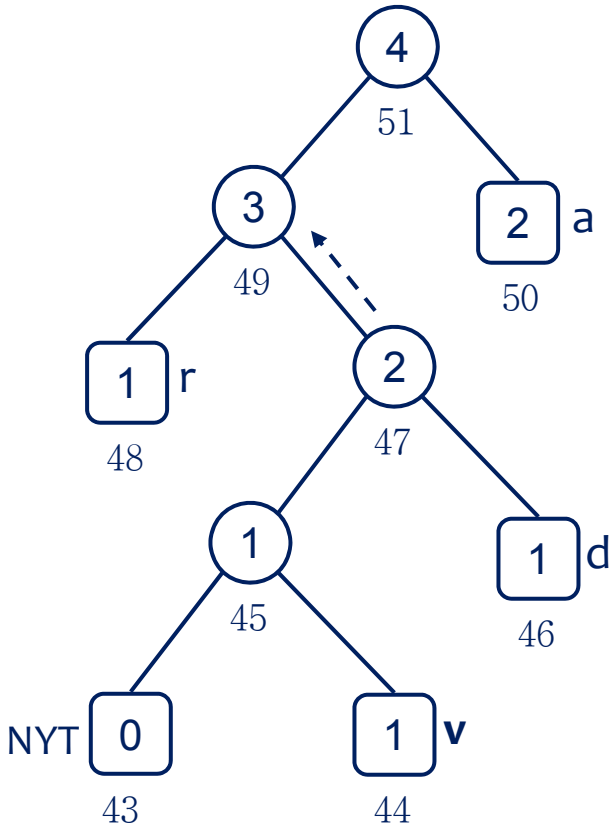
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: aardvark

Output: 000001010001000011000

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



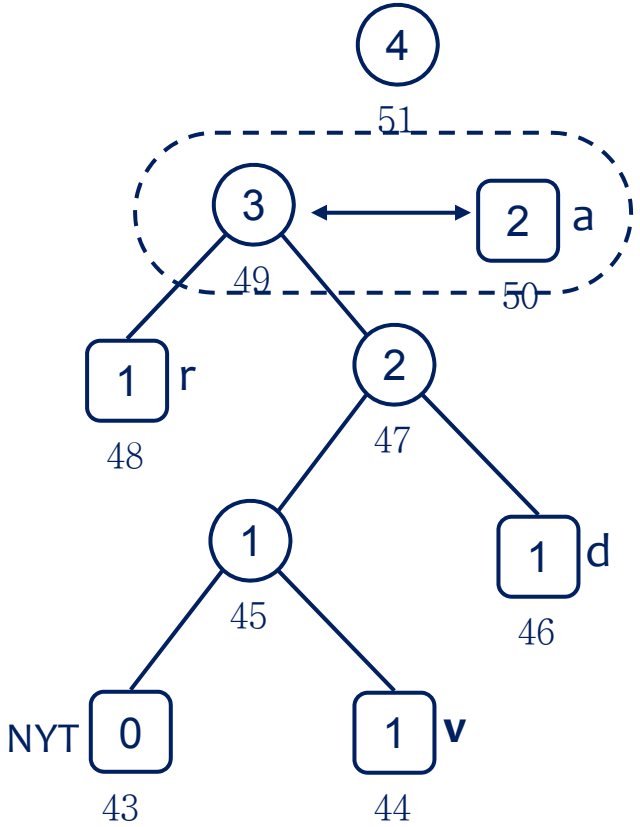
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: aardvark

Output: 000001010001000011000

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

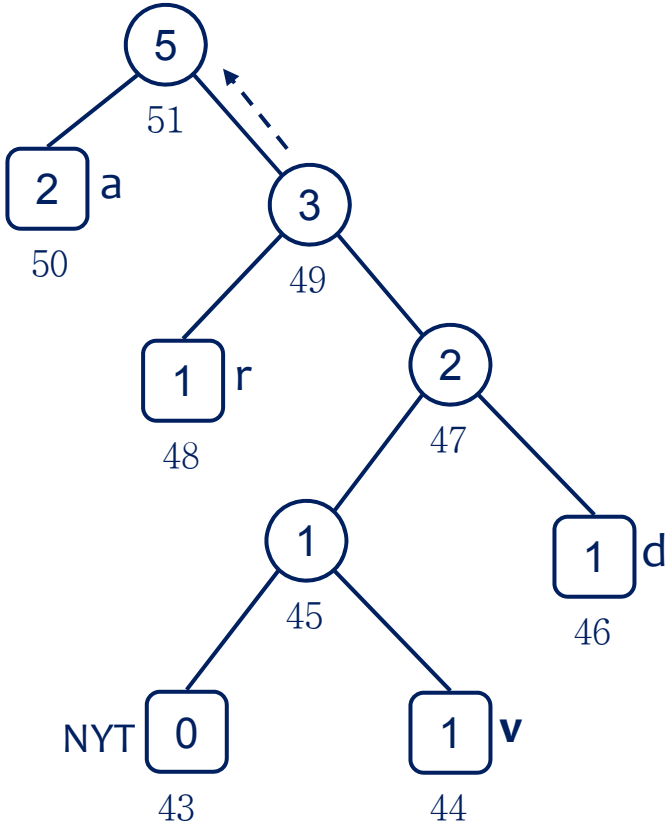
LAB

Adaptive Huffman *Encoding*

Input: aard**v**ark

Output: 00000101000100000110000**10101**

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



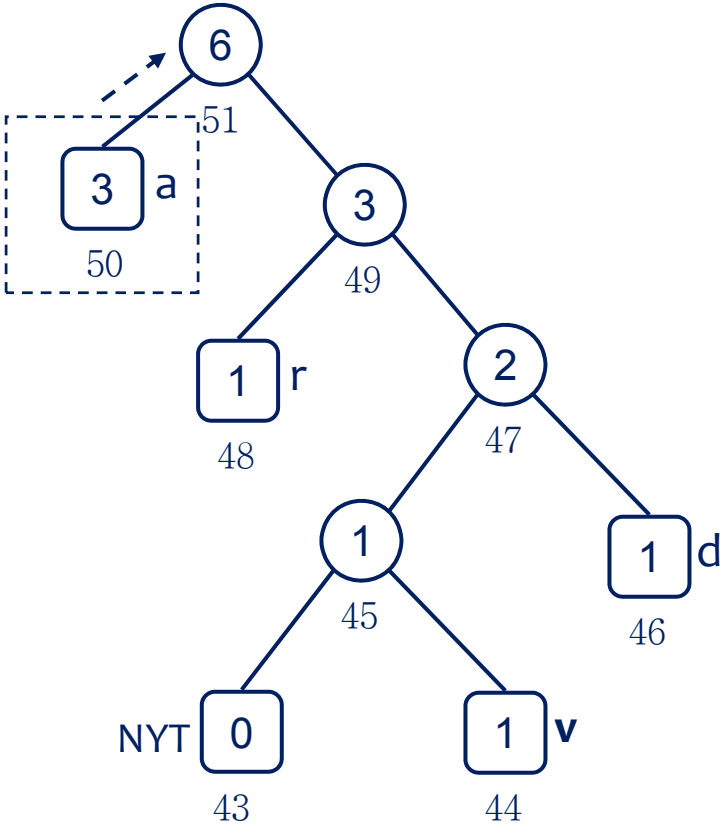
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: aardvark

Output: 000001010001000001100010101**0**

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

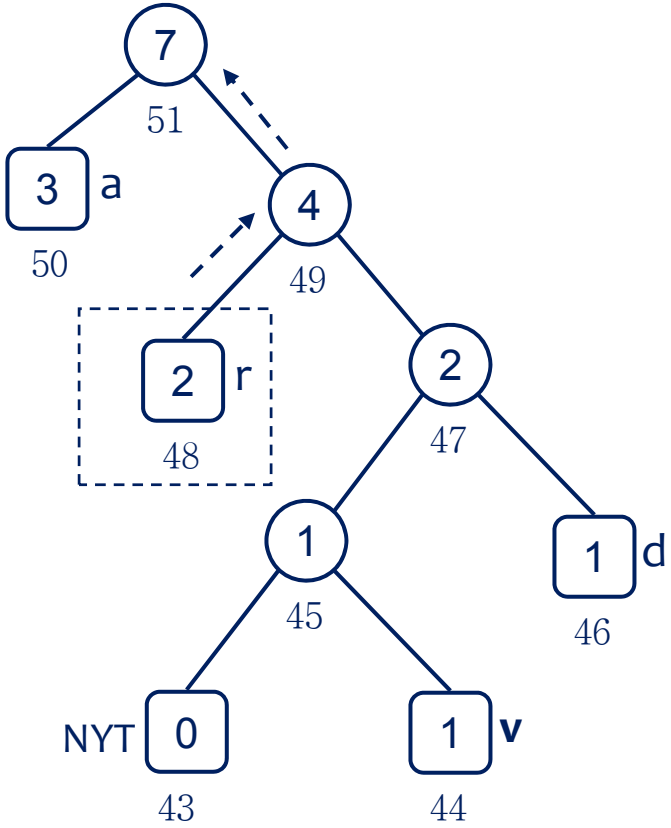
LAB

Adaptive Huffman *Encoding*

Input: aardvark

Output: 00000101000100000110001010101**0**

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



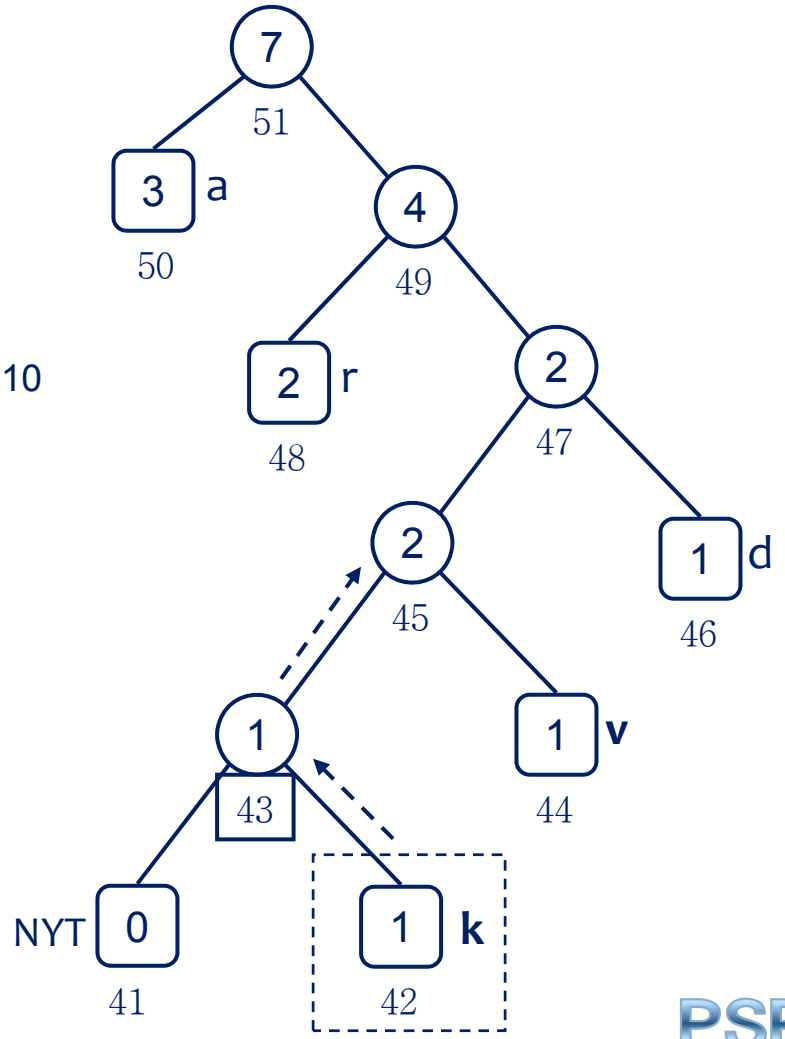
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Encoding*

Input: aardvar**k**

Output: 00000101000100000110001010101011100

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	

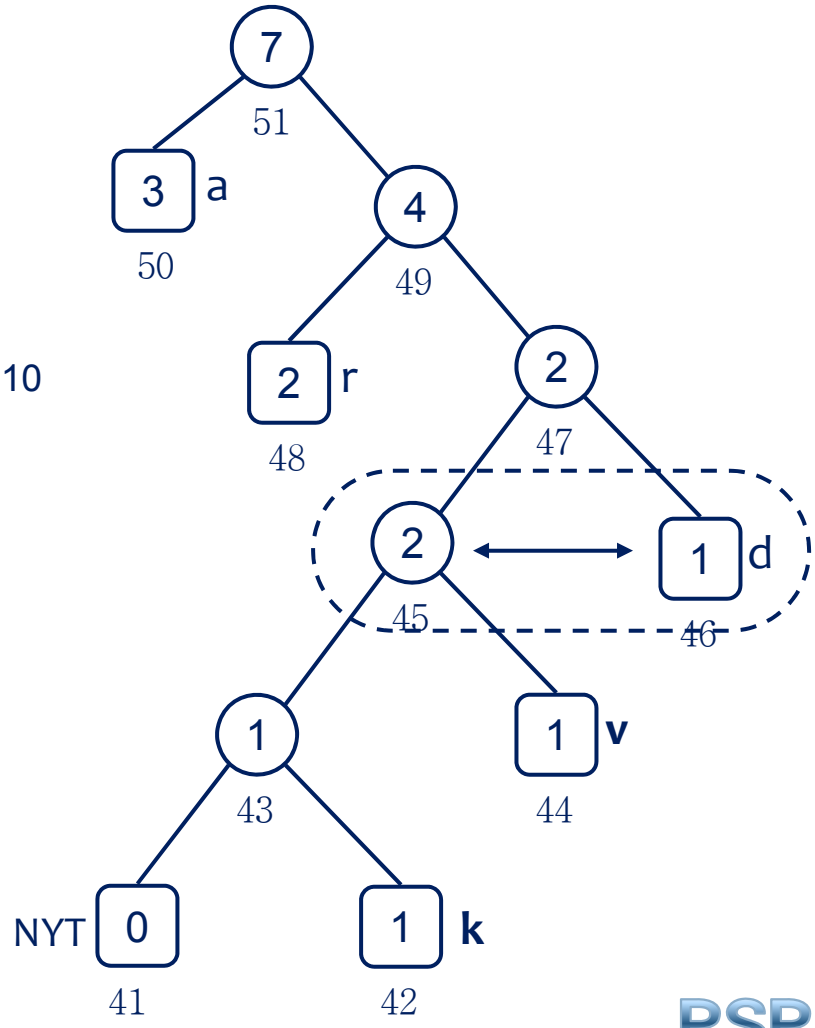


Adaptive Huffman *Encoding*

Input: aardvar**k**

Output: 00000101000100000110001010101011100

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



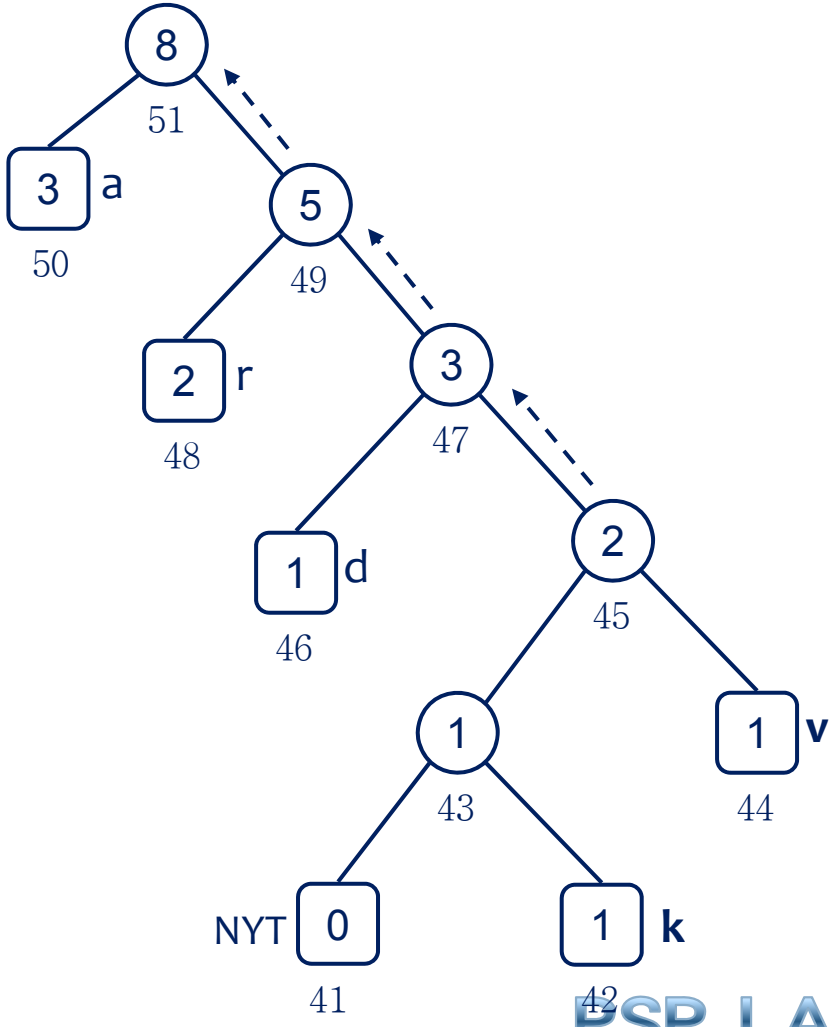
Adaptive Huffman *Encoding*

Input: aardvar**k**

Output: 00000101000100000110001010101011100**01010**

Symbol	Code
NYT	11100
a	0
r	10
d	110
v	1111
k	11101

k	01010



Adaptive Huffman *Decoding*

Output:

Input: 000001010001000001100010101010
110001010

Symbol	Code
NYT	
a	
r	
d	
v	
k	

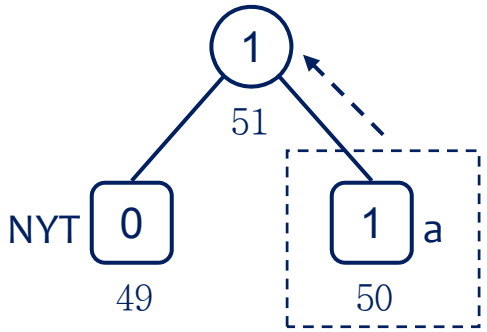
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: **a**

Input : 00000101000010000011000010101010
110001010

Symbol	Code
NYT	0
a	1
r	
d	
v	
k	



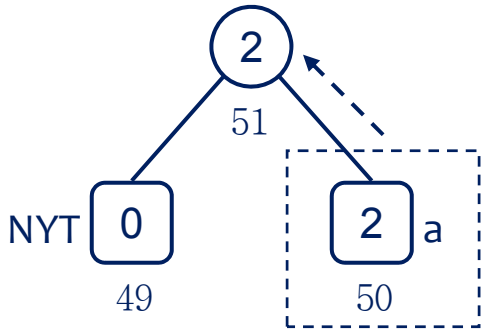
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: a**a**

Input : -----1010001000001100010101010
110001010

Symbol	Code
NYT	0
a	1
r	
d	
v	
k	



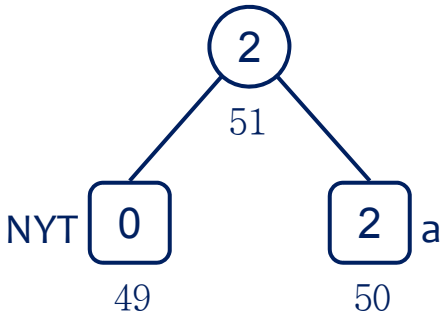
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aa

Input : ------0100010000011000010101010
110001010

Symbol	Code
NYT	0
a	1
r	
d	
v	
k	



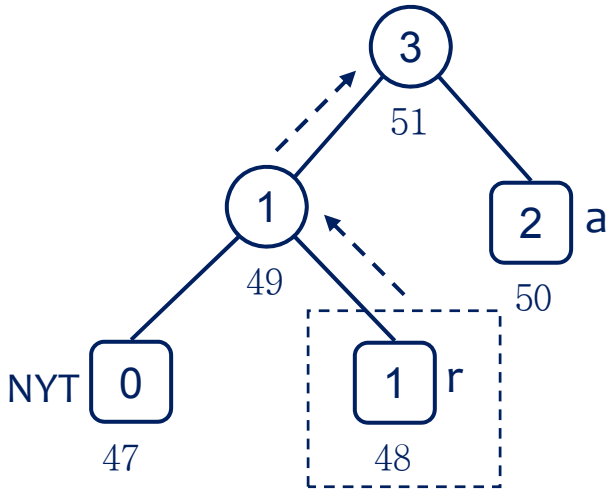
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aar

Input : --------10001000001100010101010
110001010

Symbol	Code
NYT	00
a	1
r	01
d	
v	
k	



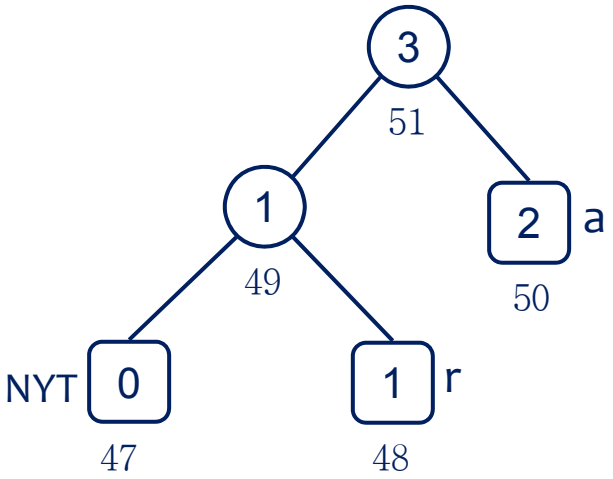
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aar

Input : ----------00000110001010101010
110001010

Symbol	Code
NYT	00
a	1
r	01
d	
v	
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

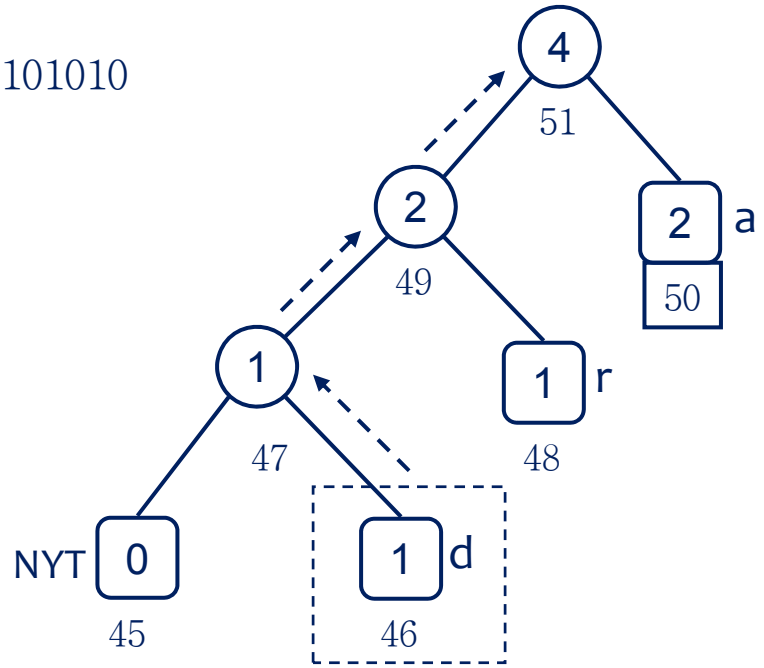
LAB

Adaptive Huffman *Decoding*

Output: aar**d**

Input : ----------00011**00010101010**
110001010

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



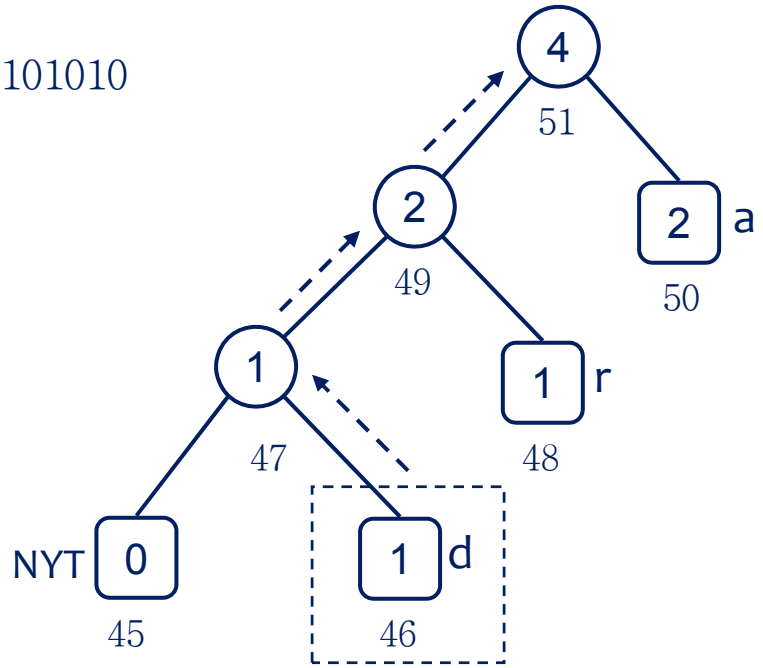
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aard

Input : ----------000**10101010**
110001010

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



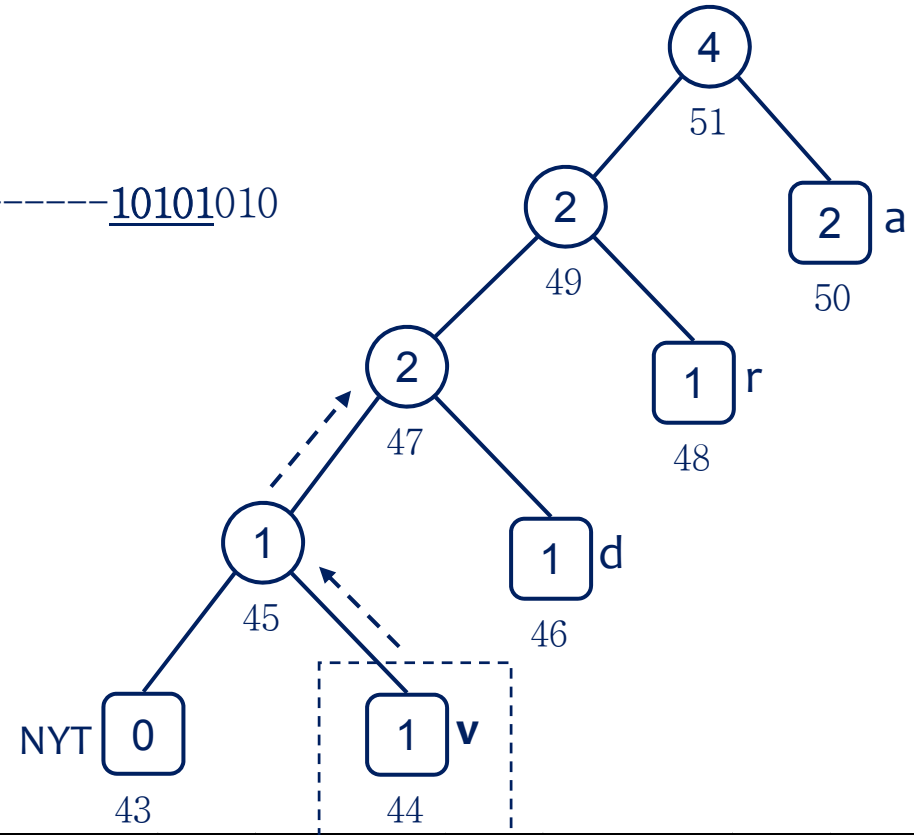
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aardv

Input : ----------10101010
110001010

Symbol	Code
NYT	000
a	1
r	01
d	001
v	
k	



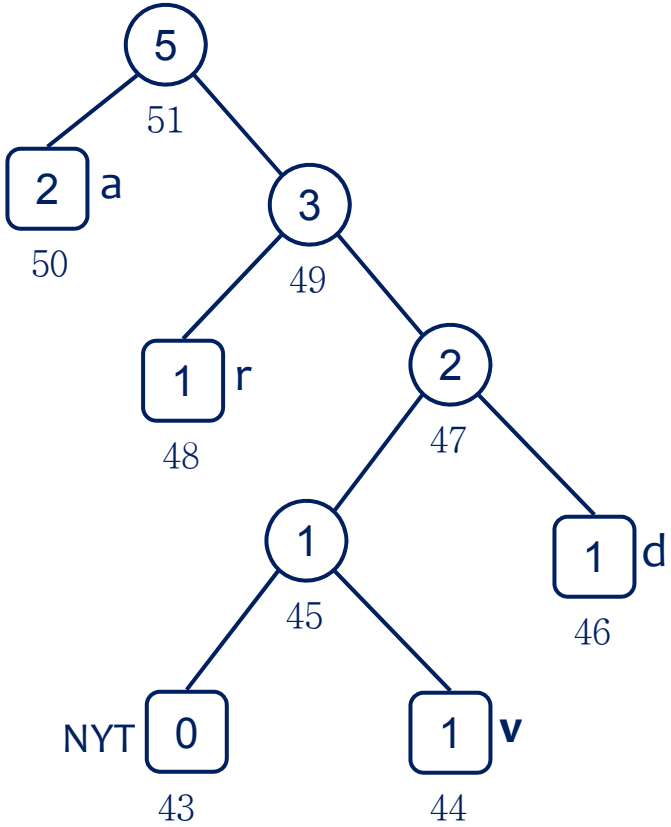
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aardv

Input : ----------010
110001010

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

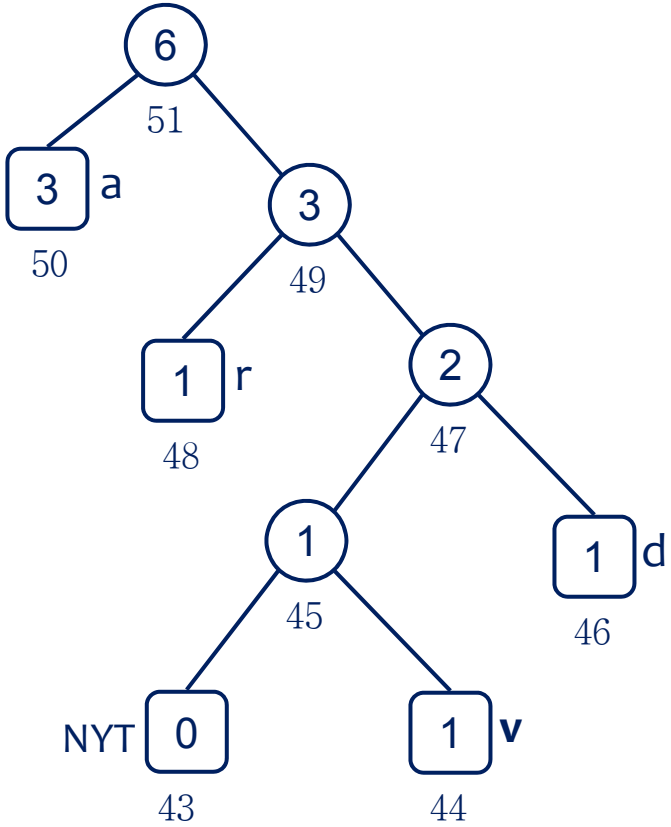
Adaptive Huffman *Decoding*

Output: aardv**a**

Input : ----------010

110001010

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



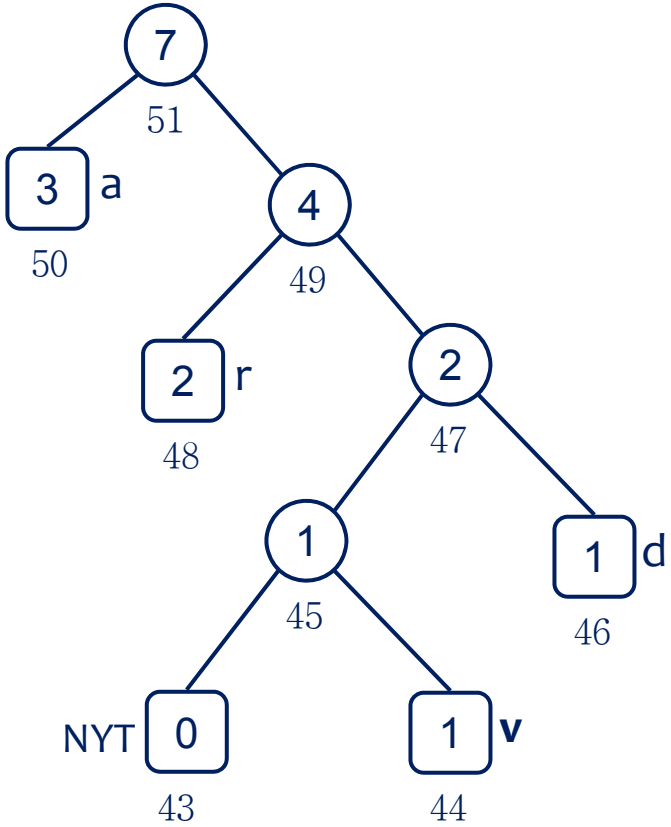
a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

Adaptive Huffman *Decoding*

Output: aardvar

Input : ----------10
110001010

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

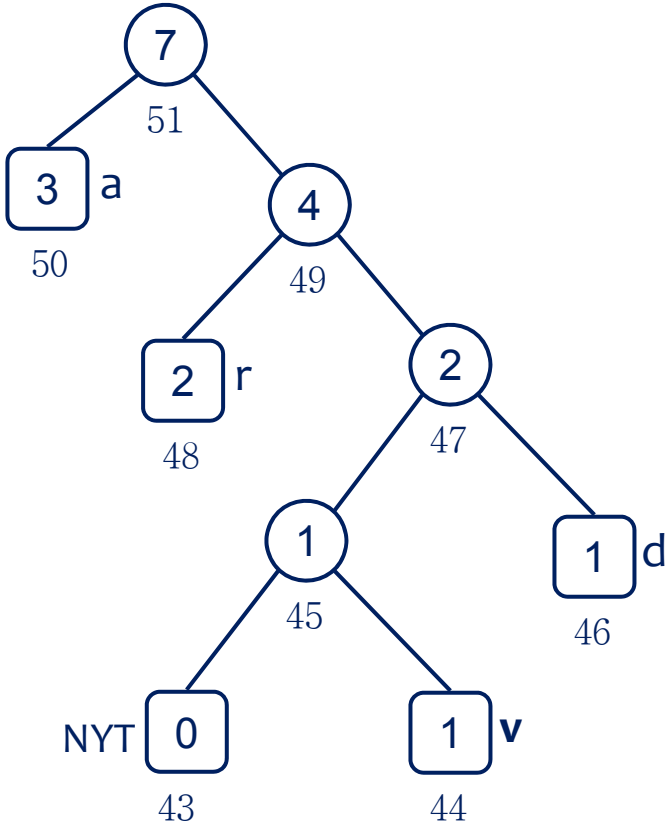
Adaptive Huffman *Decoding*

Output: aardvar

Input : ----------

1100 01010

Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	



a	00000	f	00101	k	01010	p	01111	u	10100
b	00001	g	00110	l	01011	q	10000	v	10101
c	00010	h	00111	m	01100	r	10001	w	10110
d	00011	i	01000	n	01101	s	10010	x	10111
e	00100	j	01001	o	01110	t	10011	y	11000

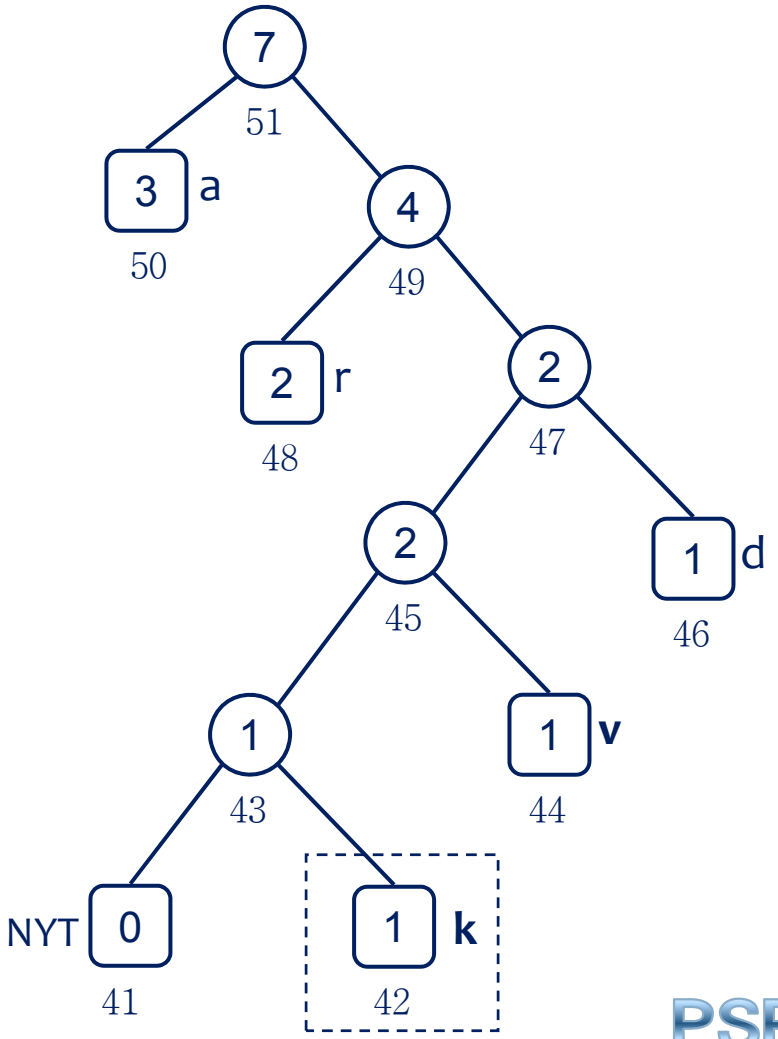
Adaptive Huffman *Decoding*

Output: aardvar**k**

Input : ----------
----01010

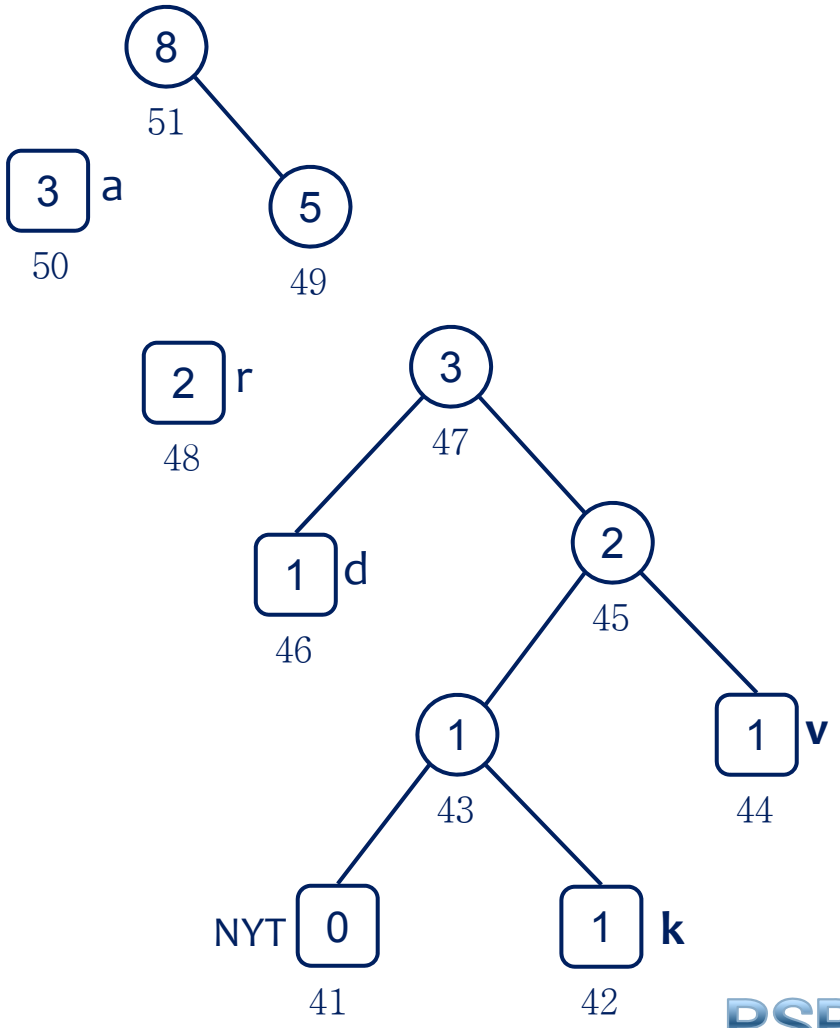
Symbol	Code
NYT	1100
a	0
r	10
d	111
v	1101
k	

k	01010



Adaptive Huffman *Decoding*

Symbol	Code
NYT	11100
a	0
r	10
d	110
v	1111
k	11101



Dealing with Counter Overflow

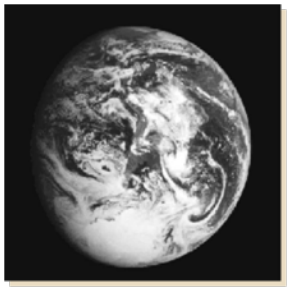
- Over time counters can overflow
  - E.g., 32-bit counter ~ 4 billion
    - BIG but still finite and can overflow on long network connections
- Solution?
  - Rescale all frequency counts (of leaf nodes) when limit is reached
    - E.g., divide by two all of them
  - Re-compute the rest of the tree (keep it Huffman!)
  - Note: After rescaling, 'new' symbols will count twice as much as old ones!
    - This is mostly a feature, not a bug:
      - Data tends to have strong local correlation
      - I.e., what happened a long time ago is not as important as what happened more recently

Huffman Image Compression

- Example images: 256x256 pixels, 8 bits/pixel, 65,536 bytes

Sena

Sensin



Earth

Omaha

- Huffman coding of ***pixel values***

Image	Bits/pixel	Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

Huffman Image Compression (2)

- Basic observations
  - The plain Huffman yields modest gains, except in the ***Earth*** case
    - Lots of black skews the pixel distribution 'nicely'
  - We are not taking into account 'obvious' correlations of pixel values
- Huffman coding of ***pixel differences***

Image	Bits/pixel	Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

Two-pass Huffman vs. Adaptive Huffman

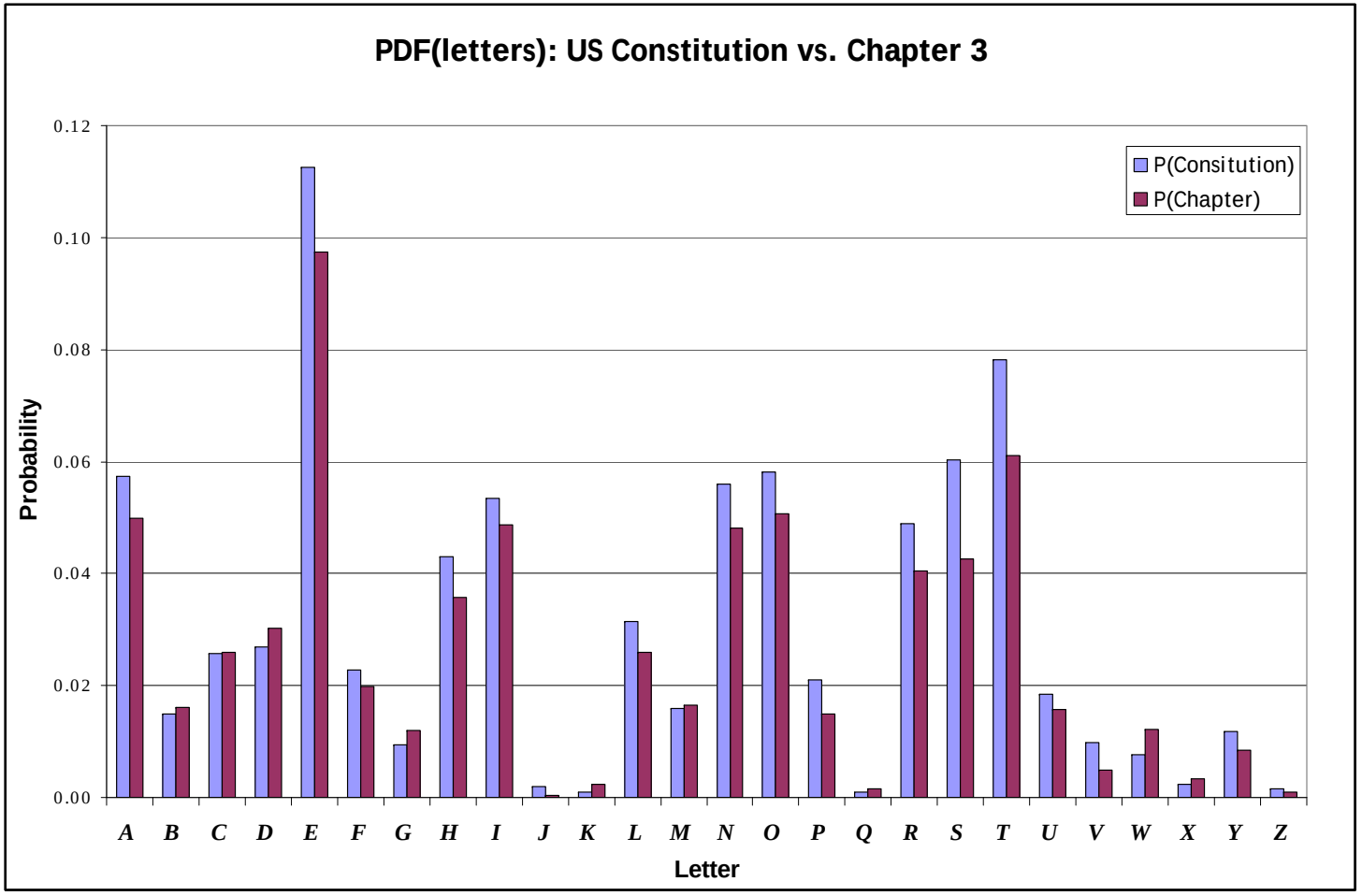
Two-pass

Image	Bits/pixel	Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

Adaptive

Image	Bits/pixel	Size (bytes)	Compression Ratio
Sena	3.93	32,261	2.03
Sensin	4.63	37,896	1.73
Earth	4.82	39,504	1.66
Omaha	6.39	52,321	1.25

Huffman Text Compression



Huffman Audio Compression

- Huffman coding: 16-bit CD audio (44,100 Hz) x 2 channels

File Name	Original File Size (bytes)	Entropy (bits)	*Est.* Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	12.8	725,420	1.16
Cohn	402,442	13.8	349,300	1.15
Mir	884,020	13.7	759,540	1.30

- Difference Huffman Coding

File Name	Original File Size (bytes)	Entropy of Diff. (bits)	*Est.* Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	9.7	569.792	1.65
Cohn	402,442	10.4	261,590	1.54
Mir	884,020	10.9	602,240	1.47

Golomb Codes

- **Invented** by Solomon W. Golomb in the 1960s.
- Golomb coding is optimal for the geometric distribution

$$\Pr(X = k) = (1-p)^{k-1}p$$

- Rice coding
  - Golomb code has a tunable parameter that can be any positive value, Rice codes are those in which the tunable parameter is a power of two
- ***Unary*** code
  - The unary representation of the number followed by 0

    $0 \rightarrow 0$

    $1 \rightarrow 10$

    $2 \rightarrow 110$

    $3 \rightarrow 1110$

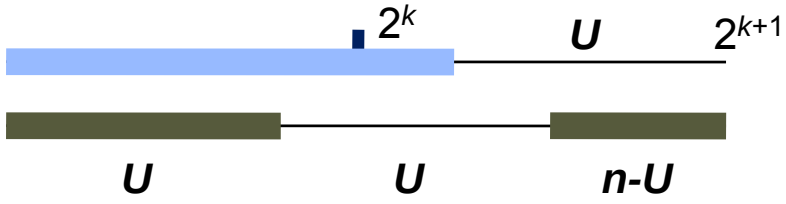
    …
  - Identical to Huffman code for {1, 2, 3, …} and P(k) = $1/2^k$
  - Optimal for the probability model

Golomb Codes (2)

- Uses a tunable parameter ***m*** to divide an input value into the quotient and the remainder.
- To represent ***n***, we compute
  - ***q*** = $\lfloor n/m \rfloor$ (quotient)
  - ***r*** = n - qm (remainder)
  - Represent ***q*** in ***unary code***, followed by
  - ***r*** in $\log_2 m$ bits
  - If ***m*** is not a power of 2 then we can use $\lceil \log_2 m \rceil$ bits
  - ***Truncated Binary Encoding***
    - $\lfloor \log_2 m \rfloor$-bit representation for $0 \leq r \leq 2^{\lceil \log_2 m \rceil} - m - 1$
    - $\lceil \log_2 m \rceil$-bit representation of $r + 2^{\lceil \log_2 m \rceil} - m$ for the rest

Golomb Codes– Truncated binary coding

- **Truncated binary coding**
  - An entropy encoding typically used for uniform probability distributions with a finite alphabet.
  - A more general form of binary encoding when *n* is not a power of two.
  - Coding (A Prefix Code)
    - For $2^k \le n \le 2^{k+1}$, there are $u = 2^{k+1} - n$ unused entries .
    - k-bit codes for $0 \le r \le u-1$.
    - (k+1)-bit codes for the rest by r+u.



Truncated binary	Encoding	Standard binary
0	~~0~~00	0
1	~~0~~01	1
2	~~0~~10	2
UNUSED	~~011~~	3
UNUSED	~~100~~	4
UNUSED	~~101~~	5/UNUSED
3	110	6/UNUSED
4	111	7/UNUSED

N=5

PSP LAB

Golomb Codes– Truncated binary coding

Input value	Offset value	Standard Binary	Truncated Binary
0	0	~~0~~00	00
1	2	001	010
2	3	010	011
3	4	011	100
4	5	100	101
5	6	101	110
6	7	110	111

N=7

Input value	Offset value	Standard Binary	Truncated Binary
0	0	~~0~~000	000
1	1	~~0~~001	001
2	2	~~0~~010	010
3	3	~~0~~011	011
4	4	~~0~~100	100
5	5	~~0~~101	101
6	12	0110	1100
7	13	0111	1101
8	14	1000	1110
9	15	1001	1111

N=10

PSP LAB

Golomb Code Example

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000

Golomb Code Example (2)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001

Golomb Code Example (3)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100

Golomb Code Example (4)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101

Golomb Code Example (5)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110

Golomb Code Example (6)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - **r+2**

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

Golomb Code Example (7)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000

Golomb Code Example (8)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000
7	1	1	1001

Golomb Code Example (9)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000
7	1	1	1001
8	1	2	10100

Golomb Code Example (10)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000
7	1	1	1001
8	1	2	10100
9	1	3	10101

Golomb Code Example (11)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000
7	1	1	1001
8	1	2	10100
9	1	3	10101
10	1	4	10110

Golomb Code Example (12)

- $m = 6$
  - $\lfloor \log_2 m \rfloor = 2$ $\quad\quad$ $\lceil \log_2 m \rceil = 3$
  - 2-bit codes for $0 \le r \le 2^{\lceil \log_2 6 \rceil} - 6 - 1$
    - $0 \le r \le 1$
  - 3-bit codes of $r + 2^{\lceil \log_2 6 \rceil} - 6$ for the rest
    - $r+2$

n	q	r	Codeword
0	0	0	000
1	0	1	001
2	0	2	0100
3	0	3	0101
4	0	4	0110
5	0	5	0111

n	q	r	Codeword
6	1	0	1000
7	1	1	1001
8	1	2	10100
9	1	3	10101
10	1	4	10110
11	1	5	10111

Golomb Codes: Choosing *m*

- Assume a binary string (zeroes & ones)
- It can be encoded counting the *runs* of identical bits
  - (either zeroes or ones)
  - A.k.a. run-length encoding (RLE)
- E.g.
  - 0000100110001001000000000100111000010000100010 0
  - ---4 -2 0--3 -2 --------9 -2 00---4 ---4 --3 -2
    - 4,2,0,3,2,9,2,0,0,4,4,3,2
  - 35 zeroes, 12 ones
  - P(0) = 35/(35+12) = 0.745

$$m = \left\lceil -\frac{\log_2(1+p)}{\log_2 p} \right\rceil \qquad m = \left\lceil -\frac{\log_2(1+0.745)}{\log_2 0.745} \right\rceil = 2$$

Summary

- Early Shannon—Fano code
- Huffman code
  - Original (two-pass) version
    - Collect symbol statistics, assign codes
    - Perform actual encoding of the source
  - Extended version
    - Group multiple symbols to reduce entropy estimate
  - Adaptive version
    - Most practical—build Huffman tree on the fly
    - Single pass
    - Escape codes for NYT symbols
    - Encoder & decoder are synchronized
    - More sensitive to local variation, tends to 'forget' older data
- Homeworks (pp. 78)
  - 4, 5, 6, 10.