Image Compression

Chapter 8 (B)

Basic compression methods

- Huffman coding
- Golomb coding*
- Arithmetic coding
- LZW coding
- Run length coding
- Symbol-Based coding *
- Bit-Plane coding *
- Block transform coding *
- Predictive coding (lossy and loss less)*
- Wavelet coding*

* will not be covered in this course

Huffman coding

- Named after Huffman, 1952
- The most popular technique for removing coding redundancy; yields the smallest possible number of code symbols per source symbol
- Variable length code
- Error-free compression technique
- Reduce only coding redundancy by minimizing the L_{avg} and assign shorter code words to the most probable gray levels

Huffman coding algorithm

- Arrange the symbol probabilities p_i in a decreasing order; consider them (p_i) as leaf nodes of a tree
- While there is more than one node:
  - *Merge the two nodes with smallest probability to form a new node whose probability is the sum of the two merged nodes*
  - *Arrange the combined node according to its probability in the tree*
  - *Repeat until only two nodes are left*

Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Huffman coding algorithm

- Starting from the top, arbitrarily assign 1 and 0 to each pair of branches merging into a node
- Continue sequentially from the root node to the leaf node where the symbol is located to complete the coding

Original source			Source reduction							
Symbol	**Probability**	**Code**	**1**		**2**		**3**		**4**	
a_2	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
a_6	0.3	00	0.3	00	0.3	00	0.3	00	0.4	1
a_1	0.1	011	0.1	011	0.2	010	0.3	01		
a_4	0.1	0100	0.1	0100	0.1	011				
a_3	0.06	01010	0.1	0101						
a_5	0.04	01011								

Lavg = (0.4)(1) + (0.3)(2) + (0.1)(3) + (0.1)(4) + (0.06)(5) + (0.04)(5) = 2.2 bits/pixel

Huffman (de)coding

- Consider the following encoded strings of code symbols

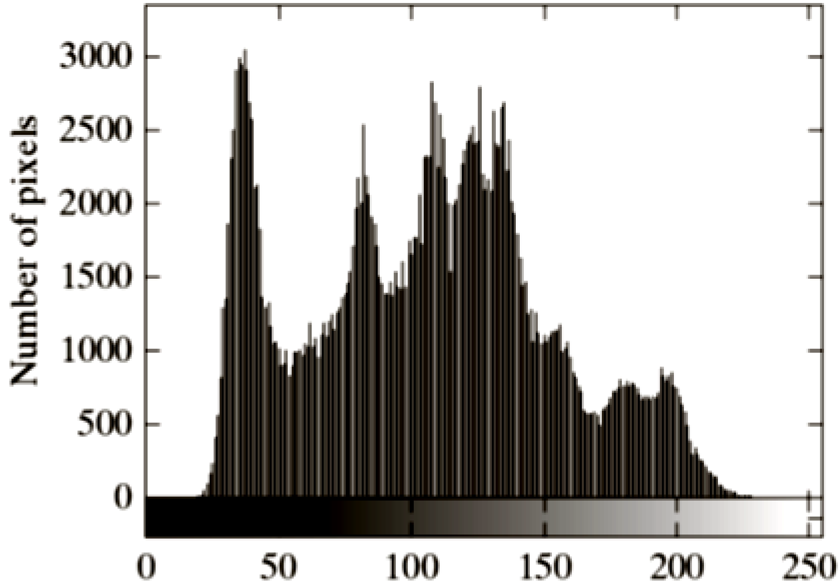
010100111100

- The sequence can be decoded by just examining the string from left to right

010100111100

$a_3 a_1 a_2 a_2 a_6$

Huffman Coding: Example



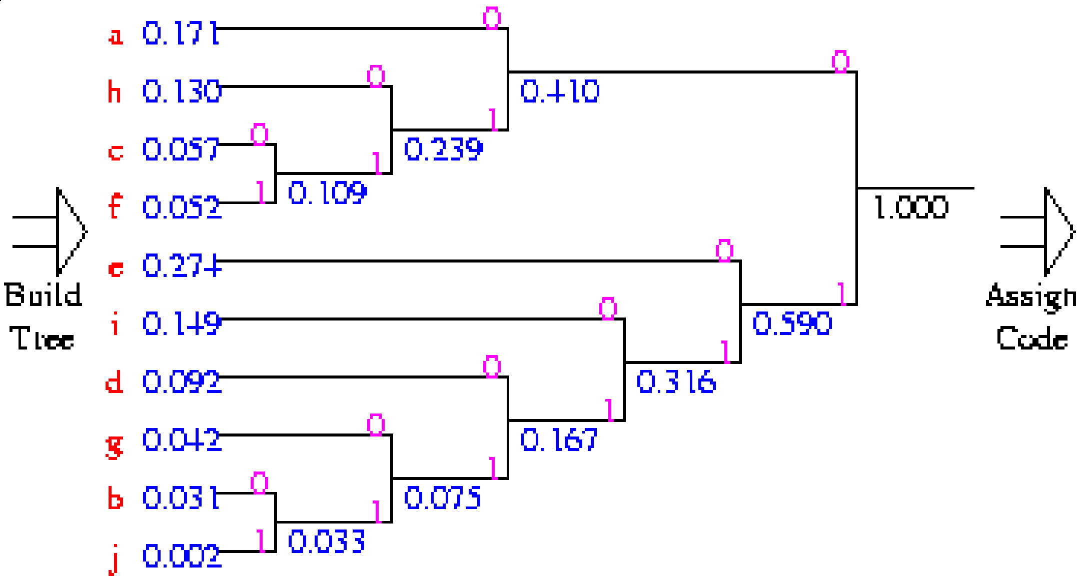
- Huffman code length = 7.428 bits/pixel
- Compression Ratio (C) = 7.428/8 = 1.077
- Relative Redundancy (R) = 1 – 1/1.077 = 0.0715 = 7.15%

Huffman Coding

X	p(X)
a	0.171
b	0.031
c	0.057
d	0.092
e	0.274
f	0.052
g	0.042
h	0.130
i	0.149
j	0.002

Huffman Coding

X	p(X)
a	0.171
b	0.031
c	0.057
d	0.092
e	0.274
f	0.052
g	0.042
h	0.130
i	0.149
j	0.002

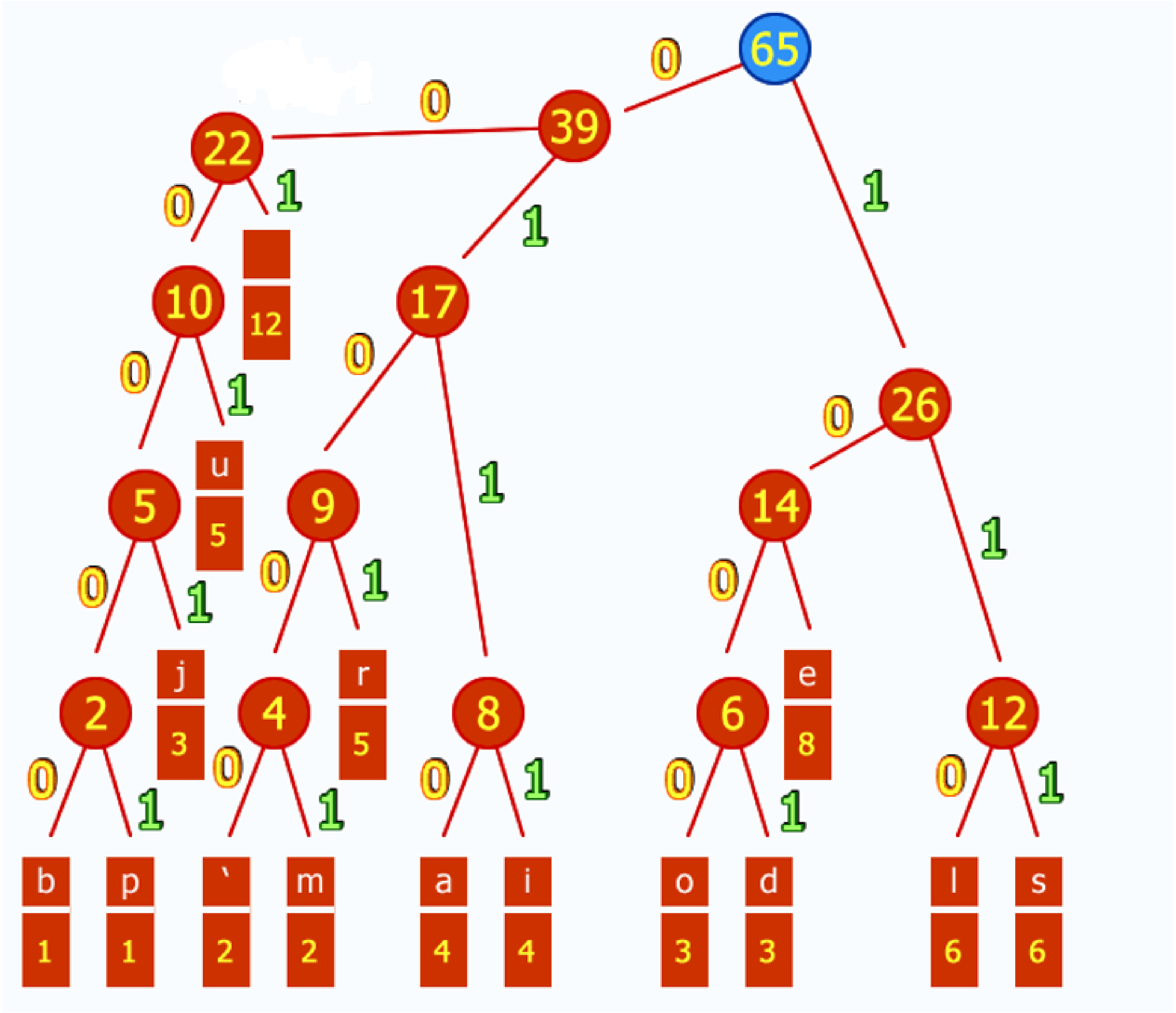


X	Code
a	0 0
b	1 1 1 1 1 0
c	0 1 1 0
d	1 1 1 0
e	1 0
f	0 1 1 1
g	1 1 1 1 0
h	0 1 0
i	1 1 0
j	1 1 1 1 1 1

Huffman Coding

File :

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12


65
39
26
22
17
14
12
10
9
8
6
5
4
2
12
u
5
j
3
r
5
e
8
b
1
p
1
`
2
m
2
a
4
i
4
o
3
d
3
l
6
s
6

Arithmetic coding

- Variable length code
- Error-free compression technique
- An entire sequence of source symbols is assigned a single arithmetic code word
  - one-to-one correspondence between source symbol and code word does not exist
- Due to above property, this coding can achieve theoretically higher compression rates than Huffman codes

Arithmetic coding

- The code word defines an interval of real numbers in the range 0 and 1
- Each symbol of the message reduces the size of the interval in accordance with its probability of occurrence

```
Set low to 0.0 Set high to 1.0
While there are still input symbols do
        Get an input symbol code_
        range = high - low.
        high = low + range*high_range(symbol)
        low = low + range*low_range(symbol)
End of While
output low
```

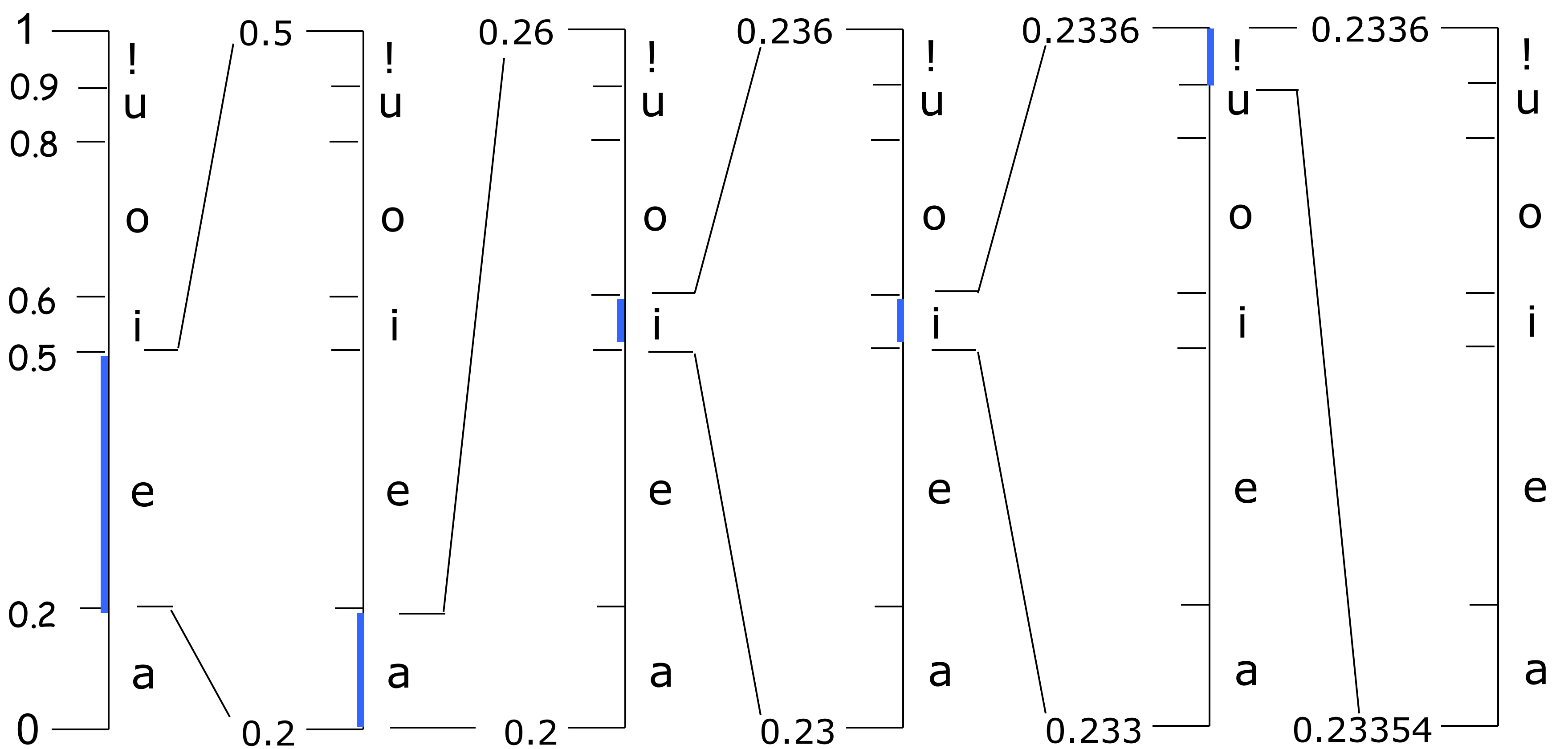
Arithmetic coding

Symbol	Probability	Range
a	.2	[0,0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1)

```
Set low to 0.0 Set high to 1.0
While there are still input symbols do
        Get an input symbol code_
        range = high - low.
        high = low + range*high_range(symbol)
        low = low + range*low_range(symbol)
End of While
output low
```

Symbol	Probability	Range
a	.2	[0,0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1)

After seeing **e** **a** **i** **i** **!**



Arithmetic Decoding

Symbol	Probability	Range
a	.2	[0,0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1)

Get encoded number
Do
find symbol whose range straddles the encoded number
output the symbol
range = symbol high value - symbol low value
subtract symbol low value from encoded number
divide encoded number by range
until no more symbols

- The decoder gets the final number :

  0.23354

- The number lies entirely within the space the model allocate for **e**

first character was **e**

Get encoded number
Do
find symbol whose range straddles the encoded number
output the symbol
range = symbol high value - symbol low value
subtract symbol low value from encoded number
divide encoded number by range
until no more symbols

Symbol	Probability	Range
a	.2	[0,0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1)

- The decoder gets the final number :
  0.23354
- The number lies entirely within the space the model allocate for **e**

first character was ***e***

- Apply decoding
  - Range = 0.5 -0.2 = 0.3
  - Encoded number = 0.23354 – 0.2 = 0.03354
  - New number = 0.03354 /0.3= 0.1118
- The range lies entirely within the space the model allocate for **a**

second character was ***a***

....

Get encoded number
Do
find symbol whose range straddles the encoded number
output the symbol
range = symbol high value - symbol low value
subtract symbol low value from encoded number
divide encoded number by range
until no more symbols

Symbol	Probability	Range
a	.2	[0,0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1)

Encoded Symbol	Output Symbol	Low	High	Range
0.23354	e	0.2	0.5	0.3
0.1118	a	0	0.2	0.2
…	…	…	…	…

Lempel-Ziv-Welch (LZW) coding

- An error-free compression technique
- Removes spatial redundnacy
- Assign fixed-length code words to variable length sequences of source symbols
- It does not require any knowledge of probability of occurrence
- LZW coding is used in the GIF, TIFF and PDF formats

LZW Encoding Algorithm

1. Initialize the dictionary to contain all blocks of length one
2. Search for the longest block **W** which has appeared in the dictionary.
3. Encode **W** by its index in the dictionary.
4. Add **W** followed by the first symbol of the next block to the dictionary.
5. Go to Step 2.

LZW Encoding Algorithm

1. Initialize the dictionary to contain all blocks of length one (D={a,b}).
2. Search for the longest block **W** which has appeared in the dictionary.
3. Encode **W** by its index in the dictionary.
4. Add **W** followed by the first symbol of the next block to the dictionary.
5. Go to Step 2.

Data: a b b a a b b a a b a b b a a a a b a a b b a

Dictionary			
Index	Entry	Index	Entry
0	a	7	
1	b	8	
2		9	
3		10	
4		11	
5		12	
6		13	

LZW Encoding – Images

- Images are scanned from left to right and from top to bottom
- A codebook or dictionary containing the source symbols to be coded is constructed on the fly
- For 8-bit monochrome images, first 256 words of the dictionary are assigned to intensities 0,1,2,3,…,255.

Dictionary Location	Entry
0	0
1	1
⋮	⋮
255	255
256	—
⋮	⋮
511	—

LZW Encoding – Images

- During the encoding process, intensity sequences that are not in the dictionary are placed in algorithmically determined locations e.g. a sequence of two-white pixels (255-255) may be assigned the location 256 in the dictionary

Dictionary Location	Entry
0	0
1	1
⋮	⋮
255	255
256	—
⋮	⋮
511	—

LZW Encoding – Images

- The next time that two consecutive white pixels are encountered, codeword 256 (the address of the location containing 255-255) is used to represent them

- If 9-bit, 512-word dictionary is employed then two pixels (16 bits) can be represented by 9 bits only

Dictionary Location	Entry
0	0
1	1
⋮	⋮
255	255
256	—
⋮	⋮
511	—

LZW Encoding – Images

39	39	126	126
39	39	126	126
39	39	126	126
39	39	126	126

- Image is scanned from left to right and from top to bottom OR consider it as a vector

Image: 39 39 126 126 39 39 126 126 39 39 126 126 39 39 126 126

Dictionary Location	Entry
0	0
1	1
…	…
255	255
256	-
257	-
…	-

LZW Encoding – Images

Image: 39 39 126 126 39 39 126 126 39 39 126 126 39 39 126 126

Encoded Output: 39 39 126 126 256 258 260 259 257 126

Dictionary Location	Entry
0	0
1	1
…	…
255	255
256	39 – 39
257	39 – 126
258	126 – 126
259	126 –39
260	39 – 39 – 126
261	126 – 126 – 39
262	39 – 39 – 126 – 126
263	126 – 39 – 39
264	39 – 126 – 126

LZW - Decoding

- Given the encoded (output) data, and the initial dictionary values
  - Do not need the entire dictionary.
- For each output symbol encountered
  - Replace with the input sequence from the dictionary
  - And, create new entry in the dictionary

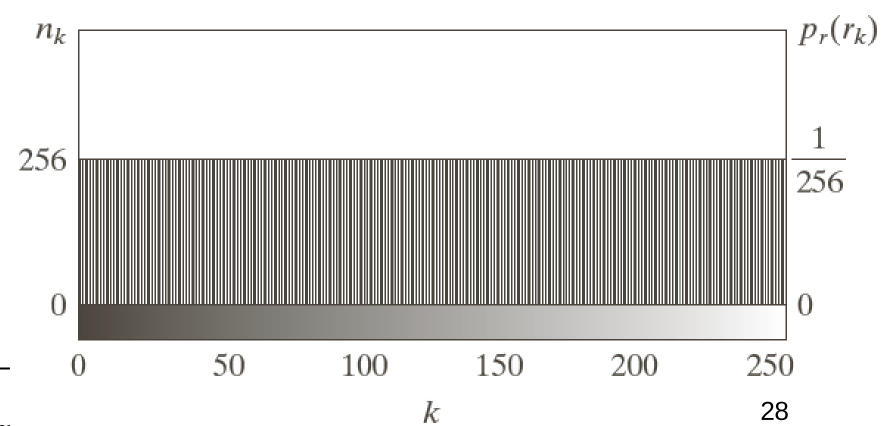
Run Length Encoding

- Image features
  - All 256 gray levels are equally probable → uniform histogram

    (variable length coding can not be applied)
  - The gray levels of each line are selected randomly so pixels are independent of one another in vertical direction
  - Pixels along each line are identical, they are completely dependent on one another in horizontal direction

Spatial redundancy

A computer generated (synthetic) 8-bit image M = N = 256



Recall

Run Length Encoding

- The spatial redundancy can be eliminated by using *run-length pairs (a mapping scheme)*
- **Run length pairs** has two parts
  - Start of new intensity
  - Number of consecutive pixels having that intensity
- Example (consider the image shown in previous slide)
  - Each 256 pixel line of the original image is replaced by a single 8-bit intensity value
  - Length of consecutive pixels having the same intensity = 256
  - Compression Ratio = $\boxed{\dfrac{256 \text{ x } 256 \text{ x } 8}{[256+256] \text{ x } 8} = 128}$

RLE – Binary Images

- Run-length Encoding is also effective in case of **binary images**
- Adjacent pixels in binary images are more likely to be identical
  - *Scan an image row from left to right and code each contiguous group (i.e. run) of 0s or 1s according to its length*
  - *Establish a convention to determine the value of the run*
- Common conventions are
  - *To specify the value of the first run of each row*
  - *To assume that each row begins with a white run, whose run length may in fact be zero*

RLE – Binary Images

- Example:

  000001100000100000000101100000 (30 bits in a row)

- Specify the value of the first run of each row:
  - *0525181125: gray level of the first run is ‘0’ (black)*

- Assume each row begins with a white run (bit ‘1’):
  - *0525181125: length of the first (white) run is 0*

References

- Chapter #8: Digital Image Processing by Rafael C. Gonzales & Richard E. Woods