# Multimedia Data Compression – Part I

Lecture 07

BIL464 Multimedia Systems
Mustafa Sert
Asst. Prof.
msert@baskent.edu.tr

Department of Computer Engineering, Başkent University
Ankara 06810 TURKEY

## Outline

- Motivation
- Types of Compression
- Run-Length-Encoding
- Entropy Encoding
- Huffman Encoding
- Arithmetic Encoding

## Motivation (1/2)

- Digital media files are usually very large, and they need to be made smaller (i.e., compressed)
- Without compression,
  - You probably won't have the storage capacity to save as many files as you'd like to, and you won't be able to communicate them across networks without overly taxing the patience of the recipients.
- However, you don't want to sacrifice the quality of your digital images, audio files, and videos in the compression process. Fortunately, the size of digital media files can be reduced significantly with little or no perceivable loss of quality.

## Motivation (2/2)

- Compression: the process of coding that will effectively reduce the total number of bits needed to represent certain information

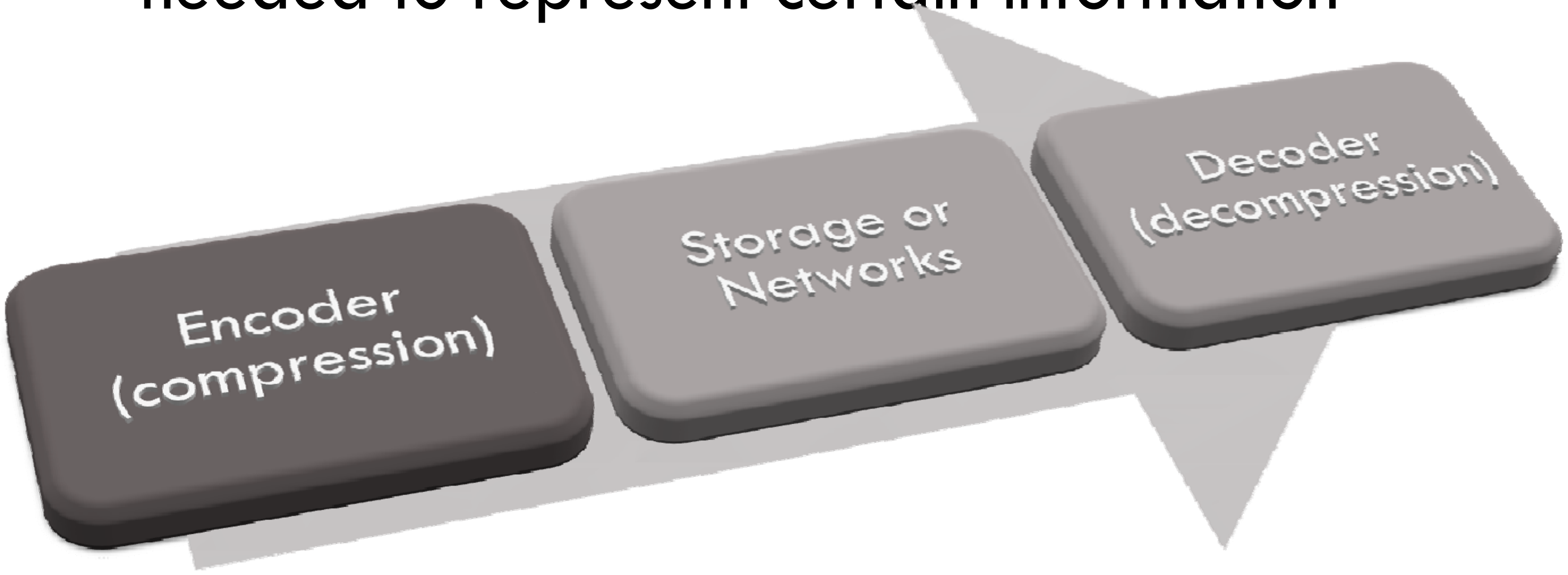


Fig. 7.1: A General Data Compression Scheme.

## Types of Compression (1/6)

- Compression algorithms can be divided into two basic types
  - **Lossless Compression:** No information is lost between the compression and decompression steps. Compression reduces the file size to fewer bits. Then decompression restores the data values to exactly what they were before the compression
  - **Lossy Compression:** Sacrifice some information. However, these algorithms are designed so that the information lost is generally not important to human perception. In image files, it could be subtle changes in color that the eye cannot detect. In sound files, it could be frequencies that are imperceptible to the human ear

## Types of Compression (2/6)

- There are other names given to types of compression algorithms:
  - *dictionary-based, entropy, arithmetic, adaptive, perceptual* and *differential compression methods*
- Dictionary-based methods (e.g. LZW compression) use a look-up table of fixed-length codes, where one code word may correspond to a string of symbols rather than to a single symbol in the file being compressed

## Types of Compression (3/6)

- Entropy based compression methods use a statistical analysis of the frequency of symbols and achieves compression by encoding more frequently-occurring symbols with shorter code words, with one code word assigned to each symbol. Shannon-Fano and Huffman encoding are examples of entropy compression
- Arithmetic encoding benefits from a similar statistical analysis, but encodes an entire file in a single code word rather than creating a separate code for each symbol

## Types of Compression (4/6)

- Adaptive methods gain information about the nature of the file in the process of compressing it, and adapt the encoding to reflect what has been learned at each step. LZW compression is by nature adaptive because the code table is created “on the fly” during compression and decompression. Huffman encoding can be made adaptive if frequency counts are updated as compression proceeds rather than being collected beforehand; the method adapts to the nature of the data as the data are read

## Types of Compression (5/6)

- Differential encoding is a form of lossless compression that reduces file size by recording the difference between neighboring values rather than recording the values themselves. Differential encoding can be applied to digital images, audio, or video
- In this lecture, we will look more closely at three algorithmic approaches to lossless compression. Other methods (i.e., lossy methods) will be covered in a separate lecture

## Types of Compression (6/6)

- The *compression rate* of a compression algorithm is the ratio of the original file size *a* to the size of the compressed file *b*, expressed as *a*:*b*. Alternatively, you can speak of the ratio of *b* to *a* as a percentage. For example, for a file that is reduced by compression to half its original size, you could say that 50% compression is achieved, or alternatively, that the compression rate is 2:1
- So the compression ratio is: $$compression\,ratio = \frac{B_0}{B_1}$$

  $B_0$ – number of bits before compression

  $B_1$ – number of bits after compression

## Run-Length-Encoding (RLE) Method (1/11)

- Is a kind of lossless compression
- Used in image compression
  - *.bmp* suffix—a Microsoft version of bitmap image files—optionally use run-length encoding.
- The method works as follows:
  - An image file is stored as a sequence of color values for consecutive pixel locations across rows and down columns. If the file is in RGB color mode, there are three bytes per pixel, one for each of the red, green, and blue color channels. If the file is grayscale, there is one byte per pixel.

## Run-Length-Encoding (RLE) Method (2/11)

- For simplicity, we'll use a grayscale file in this example. (Extending the algorithm to three bytes per pixel is straightforward.) Since each pixel position is encoded in one byte, it represents one of 256 grayscale values. (You recall that $2^8 = 256$, so eight bits can encode 256 different things.)
- Thus, a grayscale image file consists of a string of numbers, each of them between 0 and 255. Assume that the image has dimensions $100 \times 100$, for a total of 10,000 pixels. Assume also that the pixels are stored in row-major order, which means that the values from a whole row are stored from left to right, then the next row from left to right, and so forth

## Run-Length-Encoding (RLE) Method (3/11)

- The simple idea in run-length encoding is that instead of storing each of the 10,000 pixels as an individual value, it can be more concise to store number pairs (c, *n*), where c indicates the grayscale value and *n* indicates how many consecutive pixels have that value
- For example, say that in a 10,000-pixel grayscale image, the first 20 pixels are:
  - 255 255 255 255 255 255 242 242 242 242 238 238 238 238 238 238 255 255 255 255
  - The run-length encoding of this sequence would be (255, 6), (242, 4), (238, 6), (255, 4)

## Run-Length-Encoding (RLE) Method (4/11)

- The number of bytes needed to store the run-length encoded version of this line of pixels versus the number of bytes needed to store it originally is:
  - Without RLE, 20 pixels require 20 *pixels* * 1 *byte/pixel* = 20 *bytes*
  - For this example, we'll assume that everything has to be rounded to units of bytes. (We could compute how many bits are needed, but working with bytes suffices to make the point in this example.)

## Run-Length-Encoding (RLE) Method (5/11)

- The number of bytes required with RLE, we first need to figure out how many bytes are needed to store *n* in each (c, *n*) pair.
  - Clearly, c can be stored in one byte since its values range from 0 to 255.
  - Now we have to consider how large *n* could possibly be. In our example, the image has 10,000 pixels in it. It is possible that all 10,000 pixels have the same color, in which case *n* = 10,000. To store 10,000, we'd need 14 bits: ($10{,}000_{10}$ = $10011100010000_2$). Since we're assuming we allocate memory only in byte increments, this means we need two bytes.
  - If we need two bytes to store *n* in the (c, *n*) pair, then in our example above, the RLE encoded string of values would require 12 bytes rather than 20.

## Run-Length-Encoding (RLE) Method (6/11)

- **Discussion:** Say that the run-length encoding algorithm scans the image file in a preprocessing step to determine the size of the largest run of colors, where a run is a contiguous sequence of the same color. Let the size of this largest run be *r*. Try some examples values for *r*. What if the largest run is 300—that is, $r = 300$?
- Then how many bits would you need to store the second value in each (*c*, *n*) pair? You can represent the numbers 0 through 255 with eight bits, right? So eight bits is not enough, because you can't represent a number as big as 300. You can represent the values 0 through 511 with nine bits, so nine bits is enough. If you round this up to the nearest byte, that's two bytes. If you think this through intuitively, you should be able to see that the formula for figuring out how many <u>bytes</u> you need to represent a number that can be anywhere between 0 and *r* is

$$b = \left\lceil \frac{\log_2(r+1)}{8} \right\rceil$$

## Run-Length-Encoding (RLE) Method (7/11)

- Example 2: what if we have a sequence like the following?

  255 255 255 255 243 240 242 242 242 241 238 238 237 237 237 237 255 255 255 255

  - The run-length encoding of this sequence would be

    (255, 4), (243, 1), (240, 1), (242, 3), (241, 1), (238, 2), (237, 4), (255, 4)

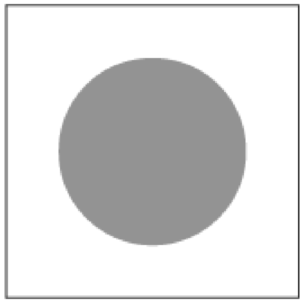
  - RLE actually requires more rather than fewer bytes than the original uncompressed image—24 rather than 20

## Run-Length-Encoding (RLE) Method (8/11)

- **Class Discussion:** The actual implementation of the method in slide #16 might be different.
  - Rather than determining the largest *n* in advance and then setting the bit depth of *n* accordingly, it makes sense to choose a bit depth of *n* in advance. Then if more than *n* consecutive pixels of the same color are encountered, the runs are divided into blocks.
  - For example, if one byte is used to represent each *n*, then the largest value for *n* is 255. If 1000 consecutive whites exist in the file (with color value 255), they would be represented as (255, 255), (255, 255), (255, 255), (255, 235)

## Run-Length-Encoding (RLE) Method (9/11)

- **Example 3:** The image in the figure below is a good example of an image where run-length encoding is beneficial
- Disregarding heading information on the image file, the image contains 100 × 100 = 10,000 pixels. With run-length encoding, again disregarding the size of the header on the file, the file requires 1084 bytes. The compression rate is about 9:1. This is an excellent compression rate, but obviously most images will not have such clearly defined color areas, and so few of these areas. (The encoded file size was obtained by saving the file as a BMP image in an application program and then looking at the file properties.)



## Run-Length-Encoding (RLE) Method (10/11)

- Concluding Remarks
  - Run-length encoding is a simple algorithm that gives acceptable results on some types of images, with no risk of loss of quality
  - It should be clear that no information is lost in the encoding process
  - In practice, RLE algorithms are fine-tuned and do not operate in precisely the way shown here, but the main principle is always the same
  - Lossless compression algorithms are applied in situations where loss of data cannot be tolerated (e.g., compression of both text and of binary-encoded computer programs)

## Run-Length-Encoding (RLE) Method (11/11)

- Concluding Remarks
  - Sound files do not lend themselves well to lossless compression. Consecutive audio samples with the same value are unusual, so RLE is not very effective for sound
  - Image files are better candidates for lossless compression
  - a number of image file formats—PNG and TIFF, for example—offer forms of lossless compression such as LZW
  - Lossless compression can also be used as one step in a more complex algorithm that does include lossy steps
    - This is the case with Huffman encoding, one step in the JPEG compression algorithm

## Entropy Encoding (1/11)

- Entropy encoding works by means of variable-length codes, using fewer bits to encode symbols that occur more frequently, while using more bits for symbols that occur infrequently
- Claude Shannon's equation, below, gives us a way a judging whether our choice of number of bits for different symbols is close to optimal.
- Let *S* be a string of symbols and $p_i$ be the frequency of the $i^{th}$ symbol in the string. ($p_i$ can equivalently be defined as the probability that the $i^{th}$ symbol will appear at any given position in the string.) Then

$$\boxed{H(S) = \eta = \sum_i p_i \log_2\left(\frac{1}{p_i}\right)}$$

# Entropy Encoding (2/11)

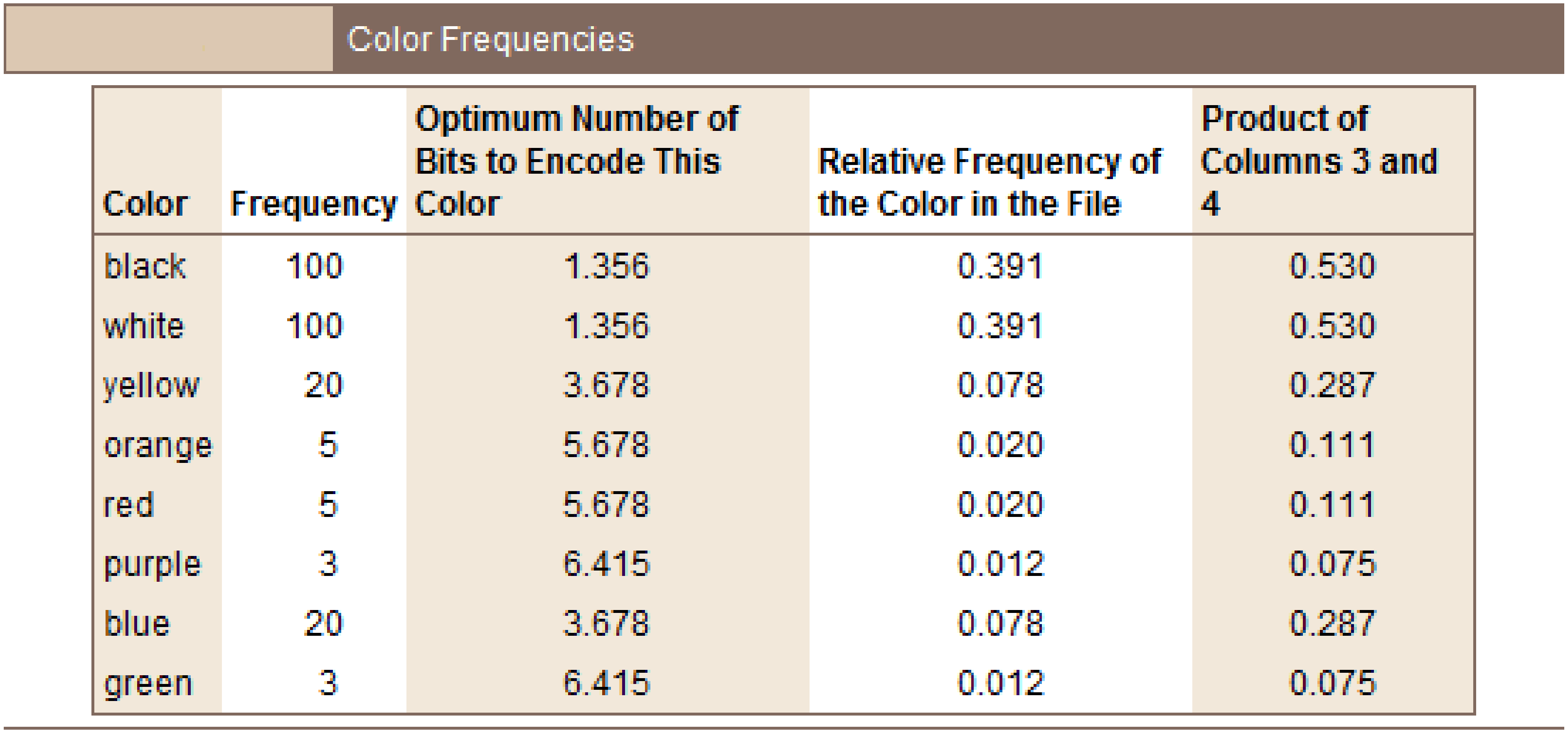
- Applying *Shannon's entropy equation*, you can determine an optimum value for the average number of bits needed to represent each symbol-instance in a string of symbols, based on how frequently each symbol appears
- Example 1: Think about an image file that has exactly 256 pixels in it, each pixel of a different color. Then the frequency of each color is 1/256. Thus, Shannon's equation reduces to

$$\sum_{0}^{255} \frac{1}{256}\left(\log_2\left(\frac{1}{\frac{1}{256}}\right)\right) = \sum_{0}^{255} \frac{1}{256}(\log_2(256)) = \sum_{0}^{255} \frac{1}{256}(8) = 8$$

- This means that the average number of bits needed to encode each color is eight, which makes sense in light of the fact that $\log_2 256 = 8$

## Entropy Encoding (3/11)

- Example 2: what if you had an image file of 256 pixels and only eight colors in the image with the following frequencies:



Color Frequencies

| Color | Frequency | Optimum Number of Bits to Encode This Color | Relative Frequency of the Color in the File | Product of Columns 3 and 4 |
|---|---|---|---|---|
| black | 100 | 1.356 | 0.391 | 0.530 |
| white | 100 | 1.356 | 0.391 | 0.530 |
| yellow | 20 | 3.678 | 0.078 | 0.287 |
| orange | 5 | 5.678 | 0.020 | 0.111 |
| red | 5 | 5.678 | 0.020 | 0.111 |
| purple | 3 | 6.415 | 0.012 | 0.075 |
| blue | 20 | 3.678 | 0.078 | 0.287 |
| green | 3 | 6.415 | 0.012 | 0.075 |

## Entropy Encoding (4/11)

- Then Shannon's equation becomes:

$$\frac{100}{256}\log_2\left(\frac{256}{100}\right) + \frac{100}{256}\log_2\left(\frac{256}{100}\right) + \frac{20}{256}\log_2\left(\frac{256}{20}\right) + \frac{5}{256}\log_2\left(\frac{256}{5}\right)$$
$$+ \frac{5}{256}\log_2\left(\frac{256}{5}\right) + \frac{3}{256}\log_2\left(\frac{256}{3}\right) + \frac{20}{256}\log_2\left(\frac{256}{20}\right) + \frac{3}{256}\log_2\left(\frac{256}{3}\right)$$
$$\approx 0.530 + 0.530 + 0.287 + 0.111 + 0.111 + 0.075 + 0.287 + 0.075 \approx 2.006$$

- So what's the significance of this? Let's see how Shannon's equation can be applied to compression. Consider each term individually. The first term corresponds to the color black. Relative to the size of the entire file, the symbol for black carries $\log_2\left(\frac{256}{100}\right) = 1.356$ bits of information every time it appears in the file, a measure of how many times black appears relative to the size of the file.

# Entropy Encoding (5/11)

- The third term corresponds to the color yellow. Yellow conveys $\log_2\left(\frac{256}{20}\right) = 3.678$ bits of information each time it appears
- The implication is that if we were to encode each color with a number of bits equal to its information content, this would be an optimum encoding. That is, we couldn't encode the file in any fewer bits. Overall, the minimum value for the average number of bits required to represent each symbol-instance in this file is 2.006. This implies that in an optimally compressed file, the average number of bits used to encode each symbol-instance cannot be less than 2.006

## Entropy Encoding (6/11)

- Keep in mind that the implication in Shannon's equation is that we don't necessarily need to use the same number of bits to represent each symbol.
  - A better compression ratio is achieved if we use fewer bits to represent symbols that appear more frequently in the file.
  - The Algorithm on the next slide, the *Shannon-Fano algorithm*, describes one way that Shannon's equation can be applied for compression. It attempts to approach an optimum compression ratio by assigning relatively shorter code words to symbols that are used infrequently, and vice versa

# Entropy Encoding (7/11)

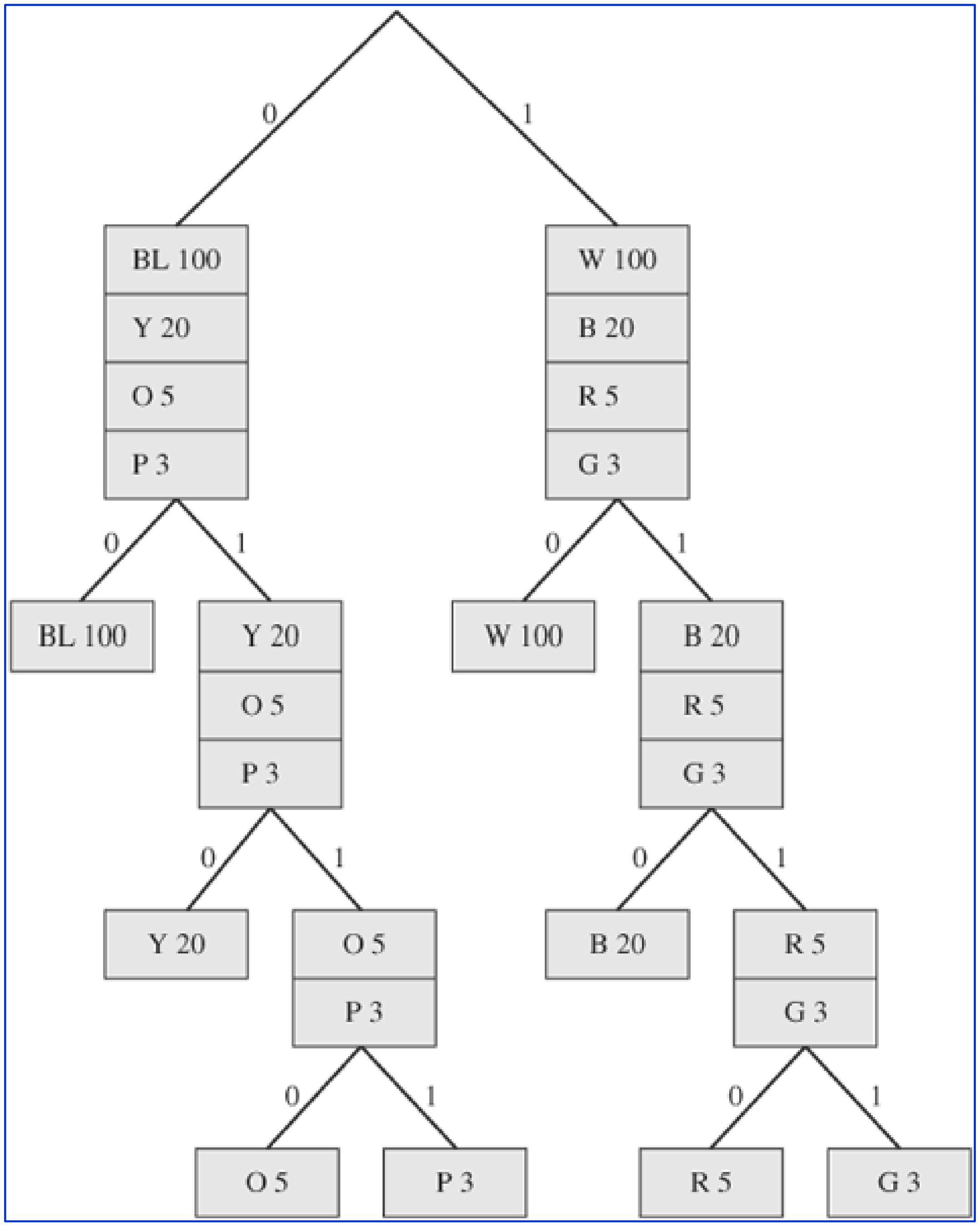
ALGORITHM

```
algorithm Shannon-Fano_compress
/*Input: A file containing symbols. The symbols could represent characters of text,
colors in an image file, etc.
Output: A tree representing codes for the symbols. Interior nodes contain no data.
Each leaf node contains a unique symbol as data.*/
{
  list = a list of the symbols in the input file, sorted by their frequency of appearance
  code_tree = split_evenly(list)
  }
algorithm split_evenly(list)
/*Input: A sorted list of symbols, list.
Output: A tree representing the encoding of the symbols in list.*/
{
  if the size of list = =1, then {
    create a node for the one symbol in list, giving it NULL children
    put the symbol as the data of the node
    return the node
  }
  else {
    Divide list into two lists list1 and list2 such that the sum of the frequencies in list1
is close as possible to the sum of the frequencies in list.
    t = a tree with one node
    lchild = split_evenly(list1)
    rchild = split_evenly(list2)
    attach lchild as the left child of t
    attach rchild as the right child of t
    return t
    }
  }
```

## Entropy Encoding (8/11)

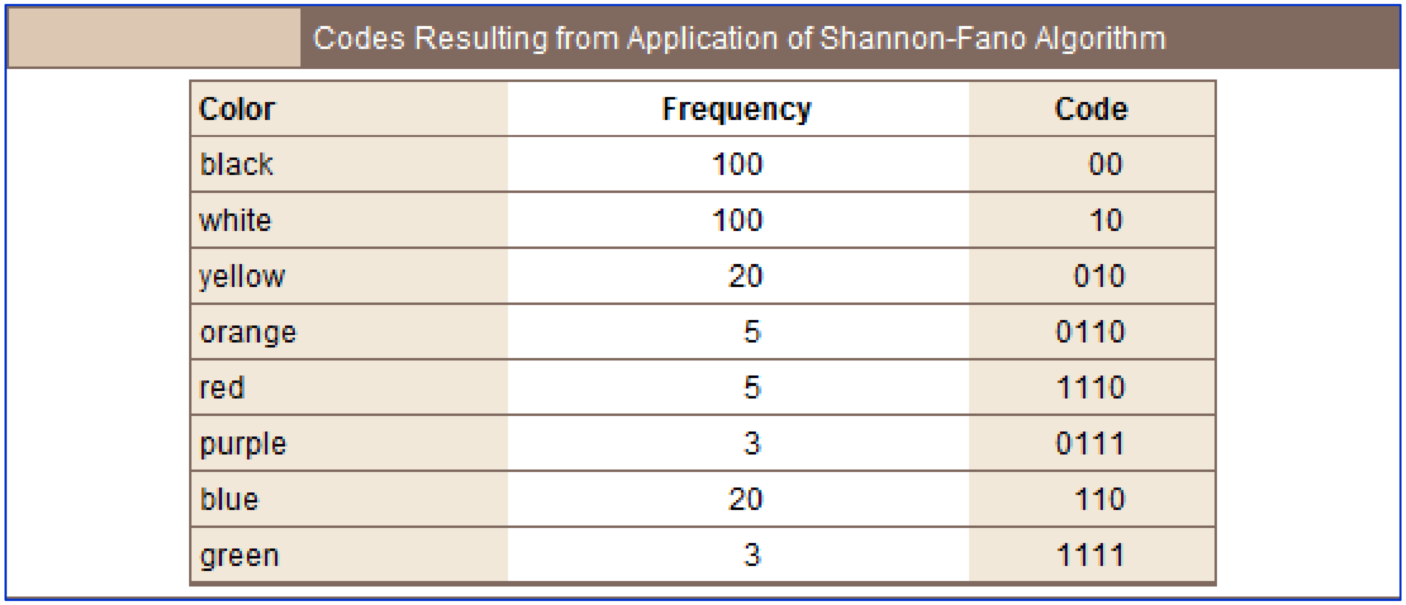
- The algorithm takes a recursive top-down approach, each recursive step dividing the symbols in half such that the symbols in the two halves have approximately the same number of instances in the file being compressed.
- The algorithm returns a code-tree where the branches are implicitly labeled with 0s and 1s. Gathering the 0s and 1s down each path from root to leaf node gives you the code for the symbol related to the leaf node.
- In our case, the tree could look like the one shown in the Figure (see next slide). Colors are abbreviated, and frequencies are shown beside the colors.

## Example of the Shannon-Fano algorithm applied to compression



## Entropy Encoding (10/11)

- In this example, the colors would be encoded as shown in theTable below. The codes vary in length from two to four bits. How close is the number of bits per symbol used in this encoding to the minimum of 2.006 determined by Shannon's equation? All the symbols for black and white are encoded with two bits, all the symbols for yellow and blue are encoded with three bits, and so forth. This give us :
(100 * 2) + (100 * 2) + (20 * 3) + (5 * 4) + (5 * 4) + (3 * 4) + (20 * 3) + (3 * 4) = 584 bits



Codes Resulting from Application of Shannon-Fano Algorithm

| Color | Frequency | Code |
|---|---|---|
| black | 100 | 00 |
| white | 100 | 10 |
| yellow | 20 | 010 |
| orange | 5 | 0110 |
| red | 5 | 1110 |
| purple | 3 | 0111 |
| blue | 20 | 110 |
| green | 3 | 1111 |

## Entropy Encoding (11/11)

- The colors require 584 bits to encode. For 256 symbols in the file, this is an average of $\frac{584}{256} = 2.28$ bits per symbol-instance—close to the minimum average number of bits per symbol given by Shannon's equation. If we assume that before compression, eight bits were used for each symbol, then the compression rate is $\frac{8}{2.28}$, which is about 3.5:1

- Huffman encoding is another type of entropy encoding that is useful in image compression.

## Huffman Encoding (1/8)

- Huffman encoding is another lossless compression algorithm that is used on bitmap image files
- it is a *variable-length encoding* scheme; that is, not all color codes use the same number of bits
- The algorithm is devised such that colors that appear more frequently in the image are encoded with fewer bits. Thus, it is a form of entropy encoding, like the Shannon-Fano algorithm
- The Huffman encoding algorithm requires two passes:
  - (1) determining the codes for the colors
  - (2) compressing the image file by replacing each color with its code

## Huffman Encoding (2/8)

- In the first pass through the image file, the number of instances of each color is determined
  - This information is stored in a frequency table
- A tree data structure is built from the frequency table in the following manner:
  - A node is created for each of the colors in the image, with the frequency of that color's appearance stored in the node.
  - These nodes will be the leaves of the code tree
  - Let's use the variable *freq* to hold the frequency in each node. Now the two nodes with the smallest value for *freq* are joined such that they are the children of a common parent node, and the parent node's *freq* value is set to the sum of the *freq* values in the children nodes. This node-combining process repeats until you arrive at the creation of a root node

# Huffman Encoding (3/8)

ALGORITHM

```
algorithm Huffman_encodingb
/*Input: Bitmap image.
Output: Compressed image and code table.*/
{
  /*Let color_freq[] be the frequency table listing each color that appears in the
image and how many times it appears. Without loss of generality, assume that all
colors from 0 to n – 1 appear in the image.*/
  initialize color_freq
/*Assume each node in the Huffman tree contains variable c for color and variable
freq for the number of times color c appears in the image.*/
  for i = 0 to n – 1 {
/*Let nd.c denote the c field of node nd*/
   create a node nd such that nd.c = i and nd.freq = color_freq[i]
  }
  while at least two nodes without a parent node remain {
   node1 = the node that has the smallest freq among nodes remaining that
have no parent node
   node2 = the node that has the second smallest freq among nodes remaining that have no parent node
/*Assume some protocol for handling cases where two of sums are the same*/
   nd_new = a new node
   nd_new.freq = node1.freq + node2.freq
   make nd_new the parent of node1 and node2
  }
/*Assign codes to each color by labeling branches of the Huffman tree*/
   label each left branch of the tree with a 0 and each right branch with a 1
for each leaf node {
   travel down the tree from the root to each leaf node, gathering the code for the
color associated with the leaf nodes
   put the color and the code in a code table
  }
  using the code table, compress the image file
}
```

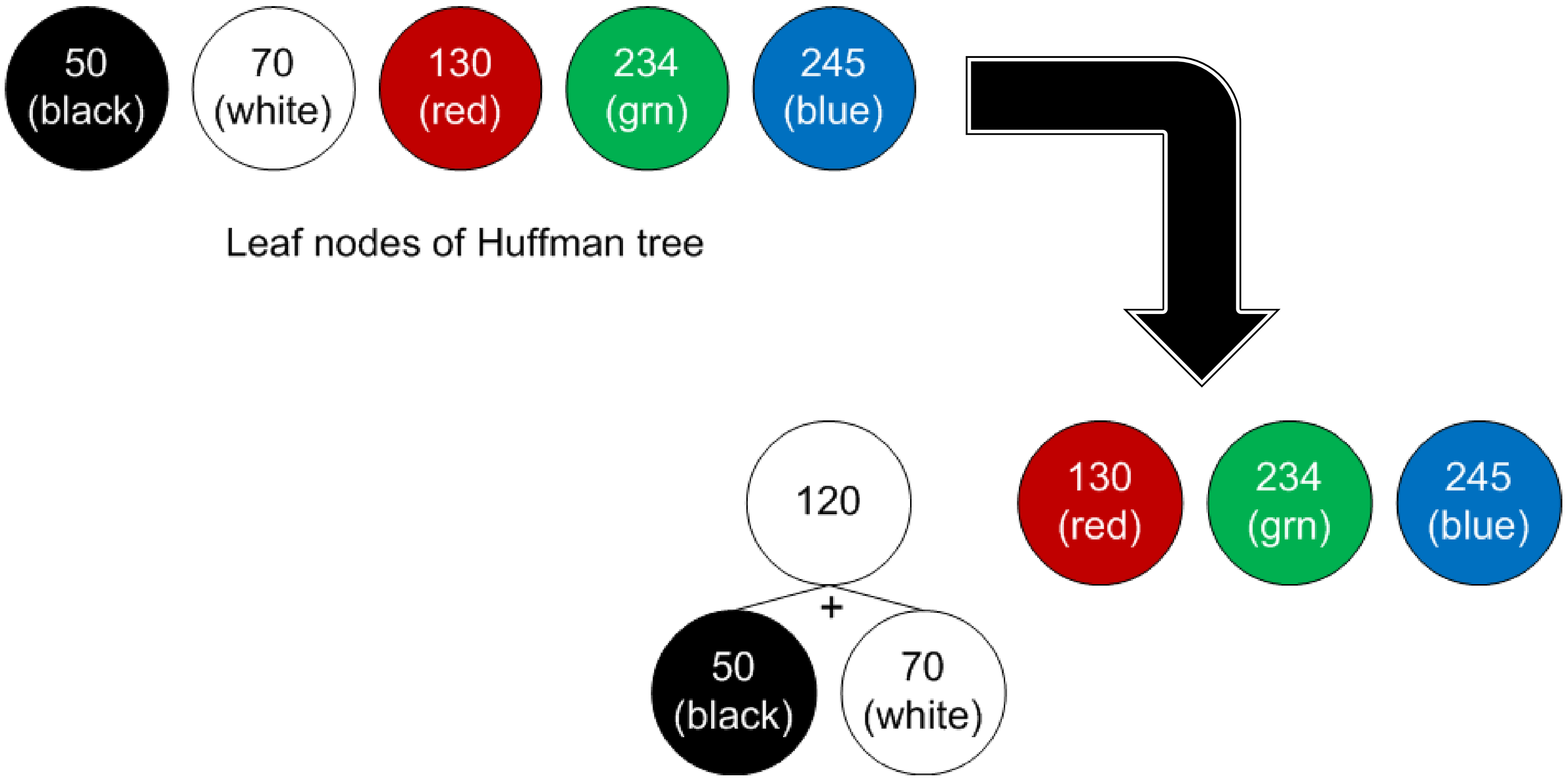
## Huffman Encoding (4/8)

- Example: An image file has only 729 pixels in it, with the following colors and the corresponding frequencies:

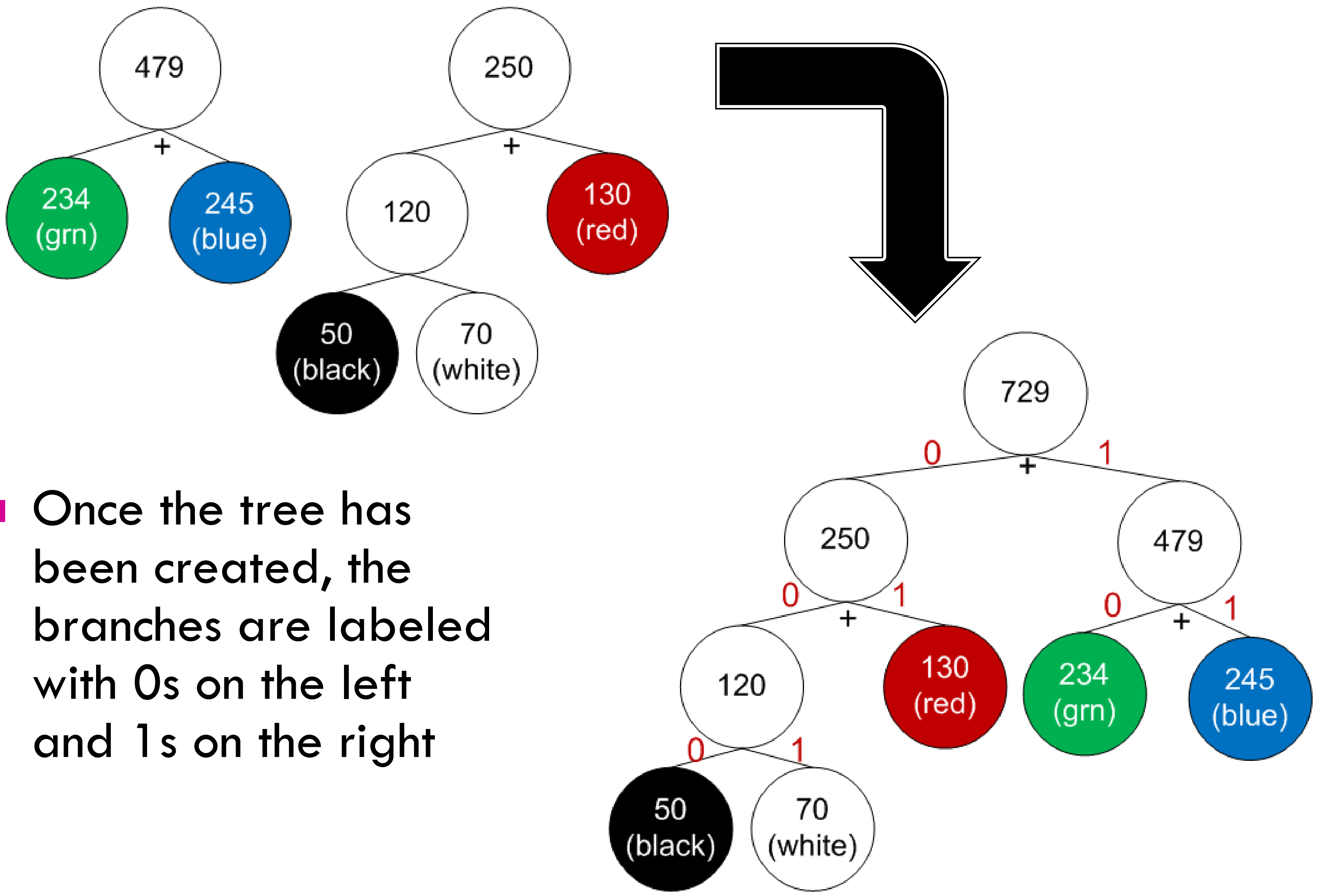
| Color | Frequency |
|---|---|
| white | 70 |
| black | 50 |
| red | 130 |
| green | 234 |
| blue | 245 |

## Huffman Encoding (5/8)

- The arrangement of the nodes are in ascending order (equal frequencies are randomly chosen)



## Huffman Encoding (6/8)



- Once the tree has been created, the branches are labeled with 0s on the left and 1s on the right

## Huffman Encoding (7/8)

- Then for each leaf node, traverse the tree from the root to the leaf, gathering the code on the way down the tree

| Color | Code |
|---|---|
| white | 001 |
| black | 000 |
| red | 01 |
| green | 10 |
| blue | 11 |

- Note that, not all codes are the same number of bits, and no code is a prefix of any other code

## Huffman Encoding (8/8)

- After the codes have been created, the image file can be compressed using these codes. Say that the first ten pixels in the image are wwwkkwwbgr (with black abbreviated as k, white as w, red as r, green as g, and blue as b). The compressed version of this string of pixels, in binary, is

  0010010010000000001001111001

- To understand decoding, picture what you would have in the compressed file:
  - Either a code table, the Huffman tree (in which the codes are implicit), or
  - The frequency table must be saved so that the Huffman tree can be used for the decoding.

## Arithmetic Encoding (1/13)

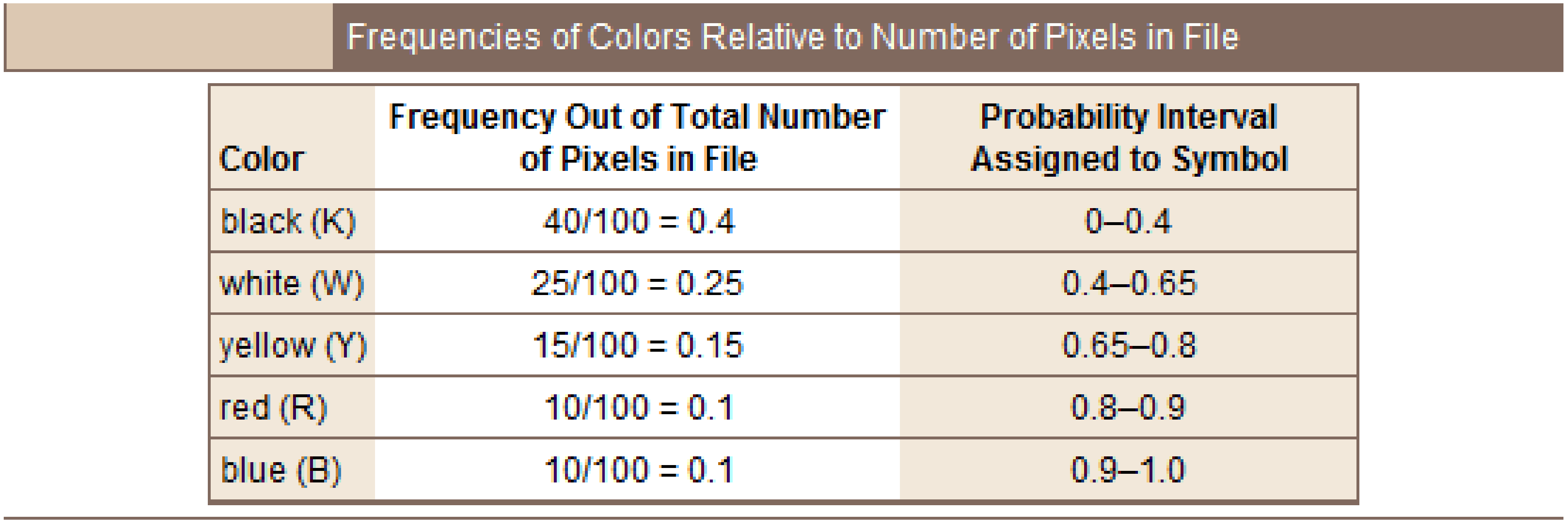
- One drawback to the Shannon-Fano algorithm is that each symbol must be treated individually; each symbol has its own code, and that code must be represented in an integral number of bits.
  - Huffman encoding has the same disadvantage.
  - By Shannon's equation, we may be able to determine that an optimum encoding is achievable by using a non-integer number of bits for each code. In our previous example, if it were possible to use exactly 1.3561 bits to encode black and white, 3.6781 bits to encode yellow and blue, 5.6781 bits to encode orange and red, and 6.415 bits to encode purple and green, then we could achieve an optimum compression rate. Each of these values is given by: $\log_2\left(\frac{1}{p_i}\right).)$
  - The problem is that using the Shannon-Fano algorithm, the optimum encoding isn't possible because we have to use an integer number of bits for each code!

## Arithmetic Encoding (2/13)

- *Arithmetic encoding* overcomes some of the disadvantage of the Shannon-Fano algorithm.
  - Like the Shannon-Fano algorithm, arithmetic encoding is based on a statistical analysis of the frequency of symbols in a file.
  - It differs, however, in that it derives additional advantage from encoding an entire file (or string of symbols) as one entity rather than creating a code symbol by symbol.
- The idea is that a string of symbols will be encoded in a single floating point number. In theory, this floating point number can be represented in whatever number of bits is needed for the compression at hand.
  - To implement this with no limit on the number of bits used, we would need infinite precision in the representation of floating point numbers, which is not possible. Let's put this issue aside for now.

## Arithmetic Encoding (3/13)

- Arithmetic encoding uses the same strategy as entropy encoding, beginning with a list of the symbols in the input file and their frequency of occurrence.
- For example, you have a file that contains 100 pixels in five colors: black (K), white (W), yellow (Y), red (R), and blue (B). The frequency of appearance of each color in the file is given in the Table:



Frequencies of Colors Relative to Number of Pixels in File

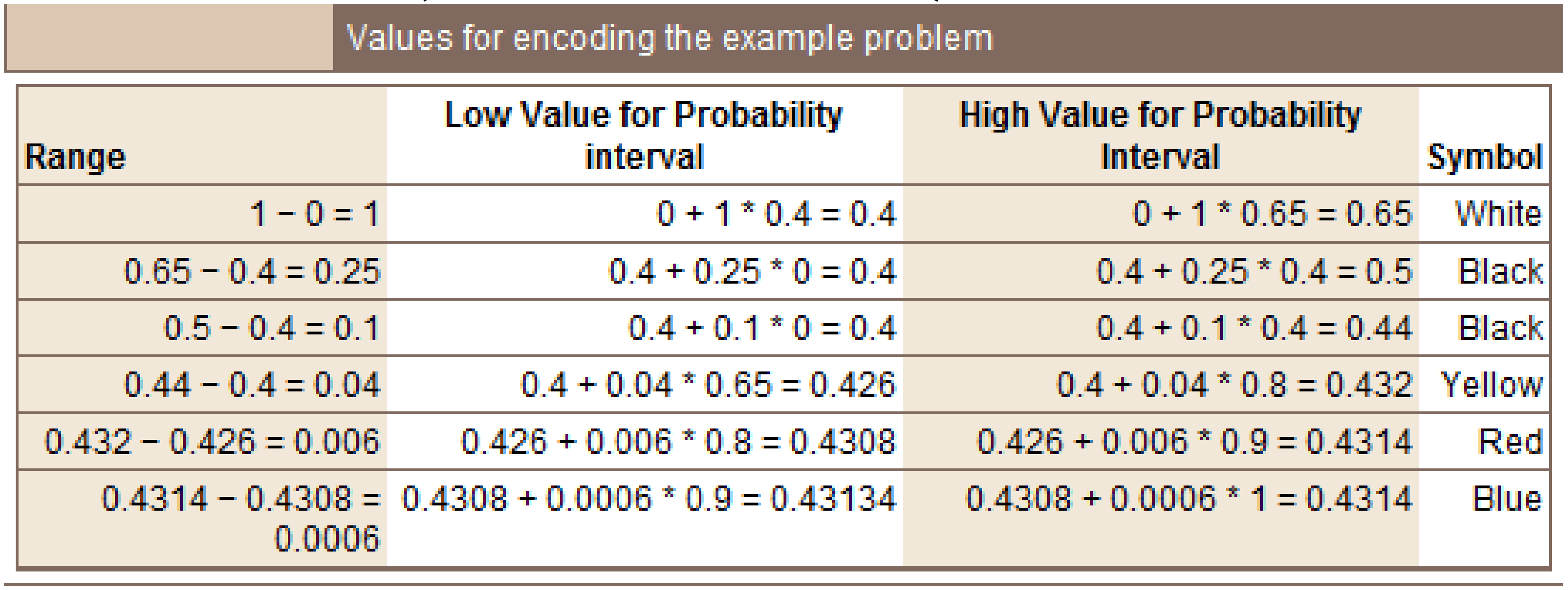
| Color | Frequency Out of Total Number of Pixels in File | Probability Interval Assigned to Symbol |
|---|---|---|
| black (K) | 40/100 = 0.4 | 0–0.4 |
| white (W) | 25/100 = 0.25 | 0.4–0.65 |
| yellow (Y) | 15/100 = 0.15 | 0.65–0.8 |
| red (R) | 10/100 = 0.1 | 0.8–0.9 |
| blue (B) | 10/100 = 0.1 | 0.9–1.0 |

# Arithmetic Encoding (4/13)

- The frequencies are expressed as numbers between 0 and 1, shown in column two.
- Each color symbol is assigned a *probability interval* whose size corresponds to its frequency of occurrence, as shown in column three.
- The entire range between 0 and 1 is called the *probability range*, and the section assigned to one symbol is a probability interval.
- For example, black's probability interval is 0–0.4. (Assume that these are half open intervals, for example [0, 0.4)). The order of color symbols in the probability range is not important, as long as the symbols are given the same order by the encoder and decoder.

# Arithmetic Encoding (5/13)

- Now let’s consider only the first six pixels in the file to make the example manageable.
- These pixels have the colors white, black, black, yellow, red and blue—represented by the symbols W, K, K, Y, R, and B.
- The arithmetic encoding algorithm assigns a floating point number to a sequence of symbols by successively narrowing the range between 0 and 1 (See the Table below)

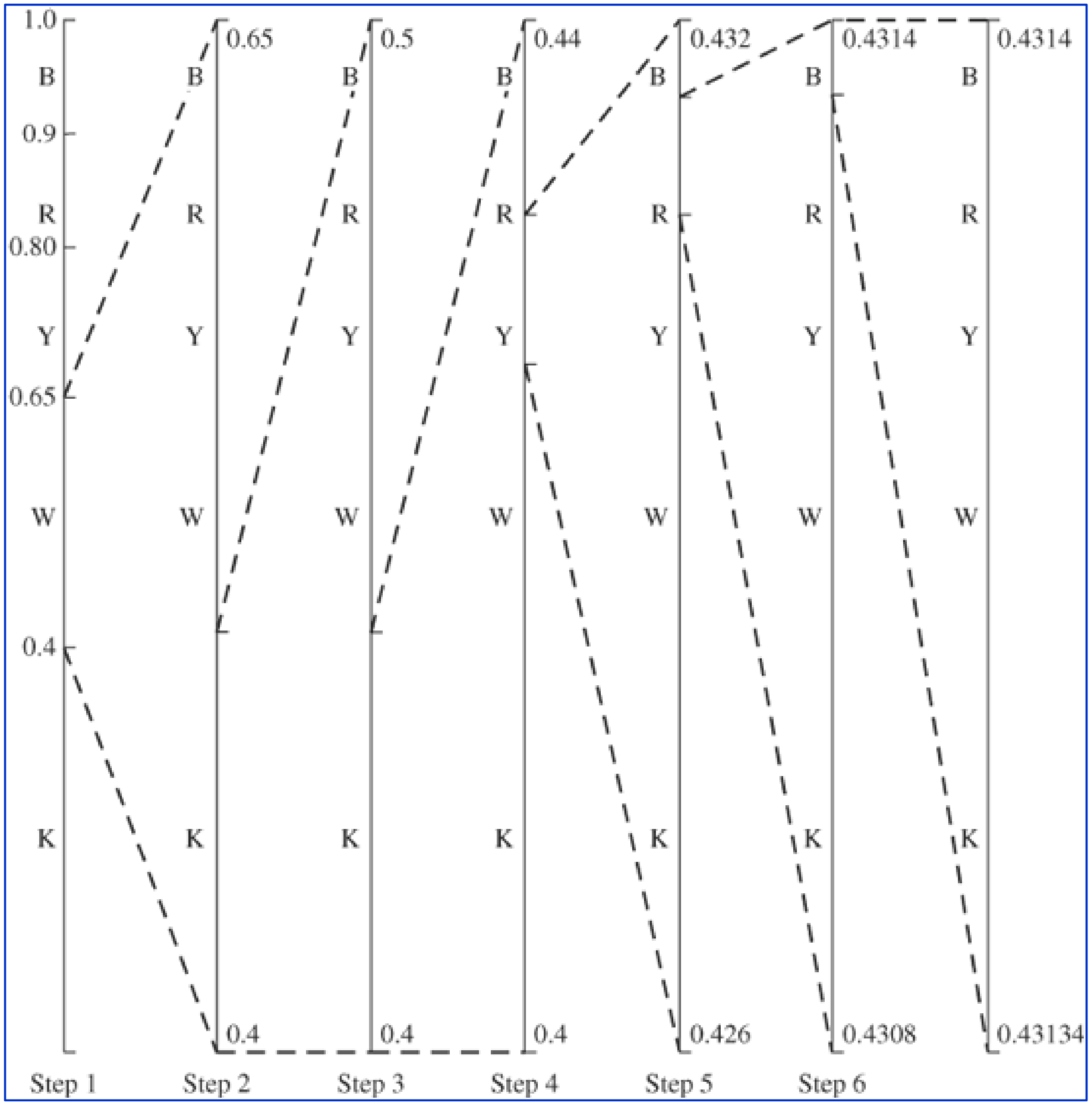


Values for encoding the example problem

| Range | Low Value for Probability interval | High Value for Probability Interval | Symbol |
|---|---|---|---|
| 1 − 0 = 1 | 0 + 1 * 0.4 = 0.4 | 0 + 1 * 0.65 = 0.65 | White |
| 0.65 − 0.4 = 0.25 | 0.4 + 0.25 * 0 = 0.4 | 0.4 + 0.25 * 0.4 = 0.5 | Black |
| 0.5 − 0.4 = 0.1 | 0.4 + 0.1 * 0 = 0.4 | 0.4 + 0.1 * 0.4 = 0.44 | Black |
| 0.44 − 0.4 = 0.04 | 0.4 + 0.04 * 0.65 = 0.426 | 0.4 + 0.04 * 0.8 = 0.432 | Yellow |
| 0.432 − 0.426 = 0.006 | 0.426 + 0.006 * 0.8 = 0.4308 | 0.426 + 0.006 * 0.9 = 0.4314 | Red |
| 0.4314 − 0.4308 = 0.0006 | 0.4308 + 0.0006 * 0.9 = 0.43134 | 0.4308 + 0.0006 * 1 = 0.4314 | Blue |

# Arithmetic Encoding (6/13)

- In our example, the first symbol is W. This means that the floating point number encoding the total string will fall somewhere in the probability interval assigned to W, between 0.4 and 0.65, as shown at the step 2 of the Figure below:



## Arithmetic Encoding (7/13)

- The second symbol is K.
  - At step 2, the range has a size of 0.25, going from 0.4 to 0.65. K was originally assigned the probability interval 0 to 0.4.
  - Moving to step 3, we want to narrow the probability range so that it begins at 0.4 and ends at 0.4 + 0.25 * 0.4. Thus, the range at step 3 goes from 0.4 to 0.5.
  - At successive steps, we narrow the range in accordance with the size and position of the symbol just read, as described in the Algorithm  given in the next slide
  - In the final step, the low value of the range can be taken as the encoding of the entire string. We've cut our example short by considering only the first six pixels in the file, but the idea is the same

## Arithmetic Encoding (8/13)

- Arithmetic Encoding Algorithm

ALGORITHM

```
algorithm arithmetic_encoding
/*Input: A string of symbols and their assigned probability intervals.
Output: A floating point number that encodes the string*/
  low = 0.0
  high = 1.0
  while input symbols remain {
    s = the next input symbol
    range = high – low
    /*s_high (s) represents the high value of symbol s's assigned probability interval,
and s_low (s) represents the low value of symbols's assigned probability interval. */
    high = low + range * s_high(s)
    low = low + range * s_low(s)
  }
  return (low + high)/2
}
```

# Arithmetic Encoding (9/13)

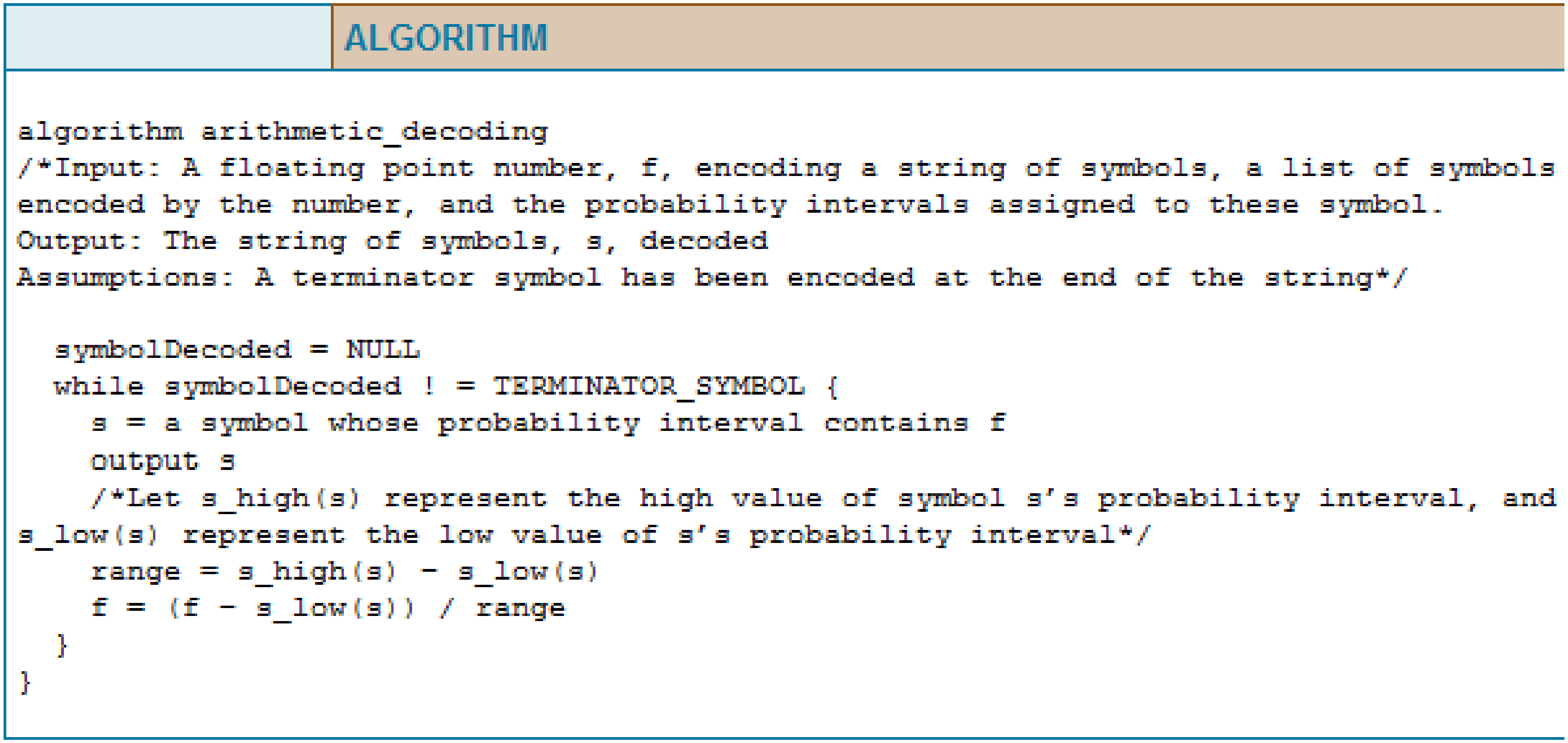
- Arithmetic Decoding
  - Given a floating point number, you can determine the first symbol of the encoded string by finding, from the initial probability range, the probability interval into which this number fits.
  - Suppose our final encoding from the previous example was 0.43137. The number 0.43137 fits in the interval assigned to W, so W is the first symbol from the encoded string.
  - You now remove the scaling of W's interval by subtracting the low value from W's probability interval and dividing by the size of W's interval, yielding (0.43137 − 0.4)/0.25 = 0.12548
  - The value 0.12548 lies in the probability interval of K. Thus, you subtract K's low range, 0, and divide by the size of K's interval, 0.4, giving 0.12548/0.4 = 0.3137
  - This value again lies in K's probability interval

## Arithmetic Encoding (10/13)

- The computation proceeds as described in the decoding procedure given in the Algorithm, yielding the values in the Table (see the next two slides)
- Although our example worked out nicely, with a maximum of five digits after the decimal point, you should be able to see that a larger example would require very high precision arithmetic.
- In fact, this appears to make arithmetic encoding impractical in the implementation, but there are ways around the problem:
  - Actual implementations use integer arithmetic and bit shifting operations that effectively accomplish the procedure described here, without requiring infinite precision floating point operations—in fact, without using any floating point operations at all.
  - There are other issues not discussed here that have been fine-tuned in various implementations, including how to terminate the input string and how to speed up what can be a time-consuming compression method

## Arithmetic Encoding (11/13)

- Arithmetic Decoding Algorithm

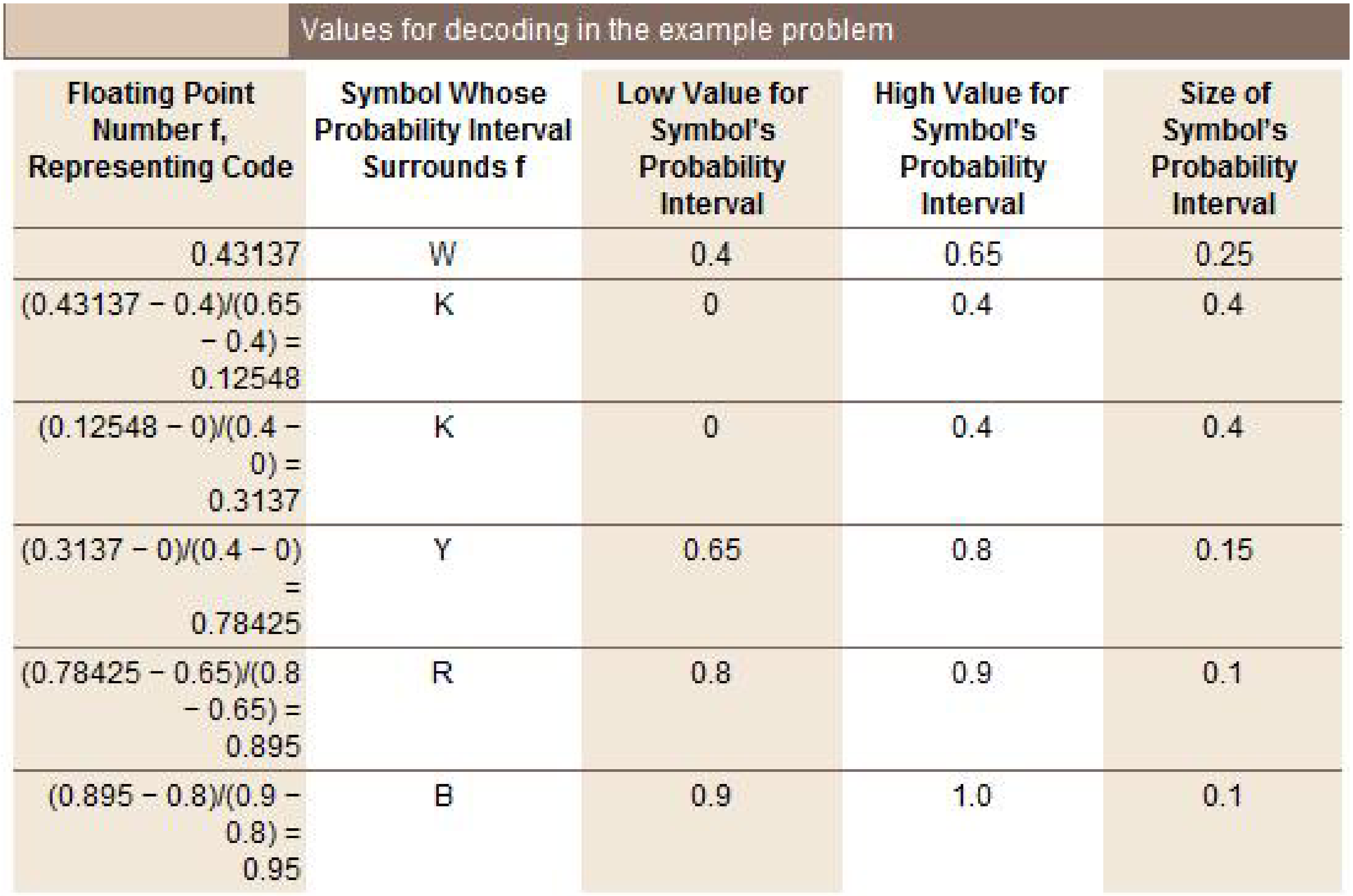


ALGORITHM

```
algorithm arithmetic_decoding
/*Input: A floating point number, f, encoding a string of symbols, a list of symbols
encoded by the number, and the probability intervals assigned to these symbol.
Output: The string of symbols, s, decoded
Assumptions: A terminator symbol has been encoded at the end of the string*/

  symbolDecoded = NULL
  while symbolDecoded ! = TERMINATOR_SYMBOL {
    s = a symbol whose probability interval contains f
    output s
    /*Let s_high(s) represent the high value of symbol s's probability interval, and
s_low(s) represent the low value of s's probability interval*/
    range = s_high(s) - s_low(s)
    f = (f - s_low(s)) / range
  }
}
```

## Arithmetic Encoding (12/13)



Values for decoding in the example problem

| Floating Point Number f, Representing Code | Symbol Whose Probability Interval Surrounds f | Low Value for Symbol's Probability Interval | High Value for Symbol's Probability Interval | Size of Symbol's Probability Interval |
|---|---|---|---|---|
| 0.43137 | W | 0.4 | 0.65 | 0.25 |
| (0.43137 − 0.4)/(0.65 − 0.4) = 0.12548 | K | 0 | 0.4 | 0.4 |
| (0.12548 − 0)/(0.4 − 0) = 0.3137 | K | 0 | 0.4 | 0.4 |
| (0.3137 − 0)/(0.4 − 0) = 0.78425 | Y | 0.65 | 0.8 | 0.15 |
| (0.78425 − 0.65)/(0.8 − 0.65) = 0.895 | R | 0.8 | 0.9 | 0.1 |
| (0.895 − 0.8)/(0.9 − 0.8) = 0.95 | B | 0.9 | 1.0 | 0.1 |

## Arithmetic Encoding (13/13)

- The important facts to note about arithmetic encoding are:
  - Arithmetic encoding is a form of entropy encoding, where the number of bits that a symbol contributes to the compressed string is proportional to the probability of that symbol appearing in the original input string
  - In arithmetic encoding, an entire input string is encoded as one value. In comparison, Huffman encoding—another example of entropy encoding—uses variable-length codes assigned on a symbol-by-symbol basis
  - Because arithmetic encoding theoretically can encode a symbol in a fractional number of bits, it is closer to optimal than Huffman encoding
  - Arithmetic encoding can be applied as one step in JPEG compression of photographic images
  - IBM and other companies hold patents on algorithms for arithmetic encoding