Parul® University

# Data Compression

**Mr. Prashant Sahatiya,** Assistant Professor
Information Technology Engineering

Parul®
University

DIGITAL LEARNING CONTENT

# The Course Outline

Chapter 1 : Compression Techniques

Chapter 2: Huffman coding algorithm

Chapter 3: Arithmetic Coding

Chapter 4: Scalar Quantization

Chapter 5: Vector Quantization

Parul®
University

DIGITAL LEARNING CONTENT

# CHAPTER-2

# Huffman Coding Algorithm


# What is Huffman Coding?

- Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

- Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.


# How Huffman Works?

- There are mainly two major parts in Huffman Coding

**1)** Build a Huffman Tree from input characters.

**2)** Traverse the Huffman Tree and assign codes to characters.


# Steps to build Huffman Tree

- Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

**1.** Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)


# Steps to build Huffman Tree

**2.** Extract two nodes with the minimum frequency from the min heap.

**3.** Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

**4.** Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

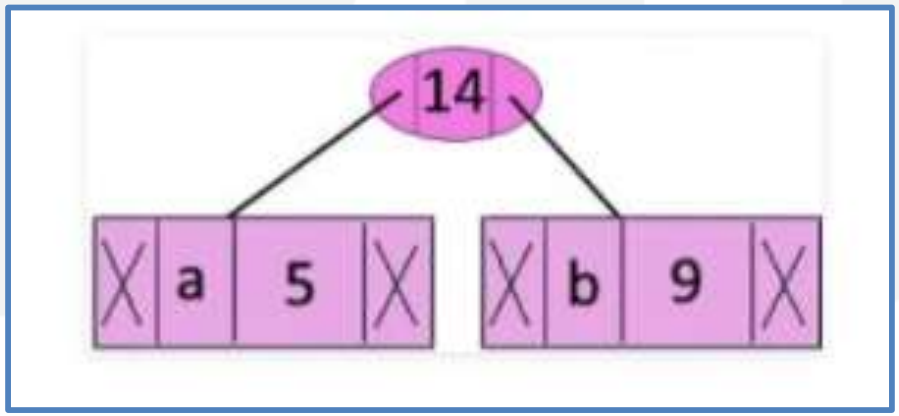
# Example: Huffman Coding Algorithm

| character | Frequency |
|---|---|
| a | 5 |
| b | 9 |
| c | 12 |
| d | 13 |
| e | 16 |
| f | 45 |

# Example: Huffman Coding Algorithm

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



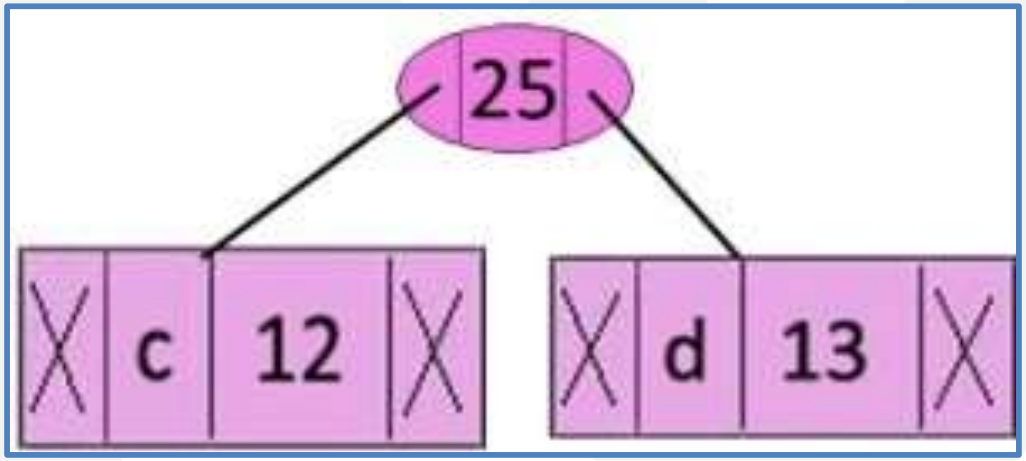
# Example: Huffman Coding Algorithm

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements.

| character | Frequency |
|---|---|
| c | 12 |
| d | 13 |
| Internal Node | 14 |
| e | 16 |
| f | 45 |

# Example: Huffman Coding Algorithm

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25.



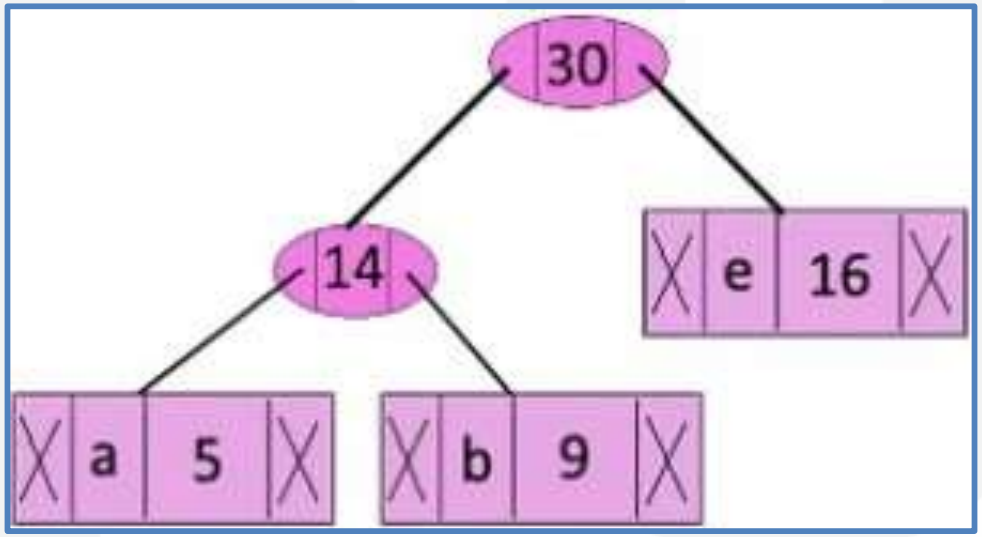
# Example: Huffman Coding Algorithm

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

| character | Frequency |
|---|---|
| Internal Node | 14 |
| e | 16 |
| Internal Node | 25 |
| f | 45 |

# Example: Huffman Coding Algorithm

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30.



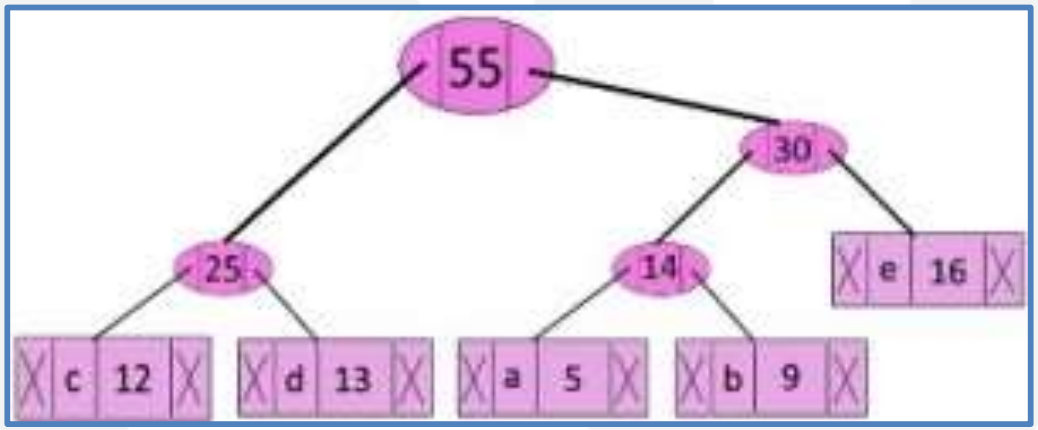
# Example: Huffman Coding Algorithm

Now min heap contains 3 nodes.

| character | Frequency |
|---|---|
| Internal Node | 25 |
| Internal Node | 30 |
| f | 45 |

# Example: Huffman Coding Algorithm

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55.



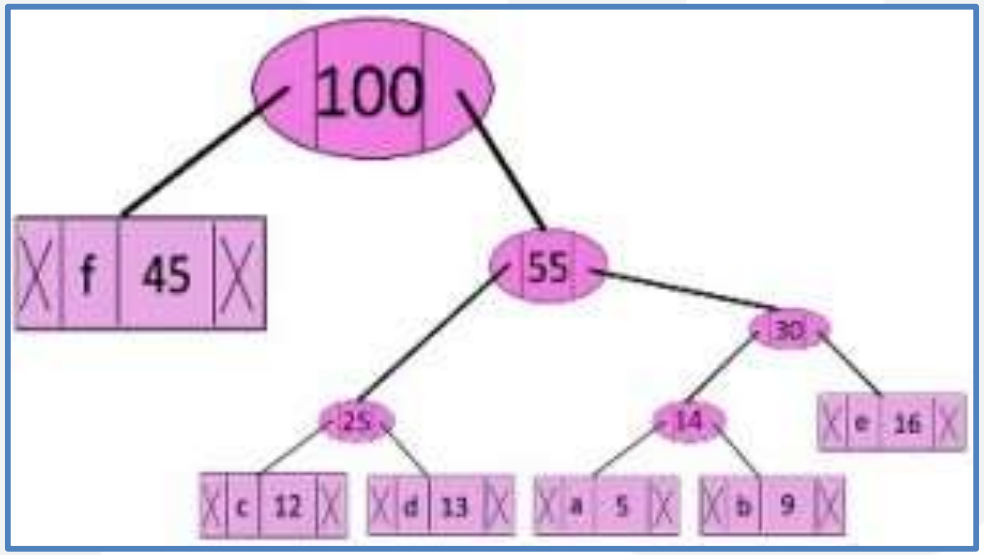
# Example: Huffman Coding Algorithm

Now min heap contains 2 nodes.

| character | Frequency |
|---|---|
| f | 45 |
| Internal Node | 55 |

# Example: Huffman Coding Algorithm

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



# Example: Huffman Coding Algorithm

Now min heap contains only one node.

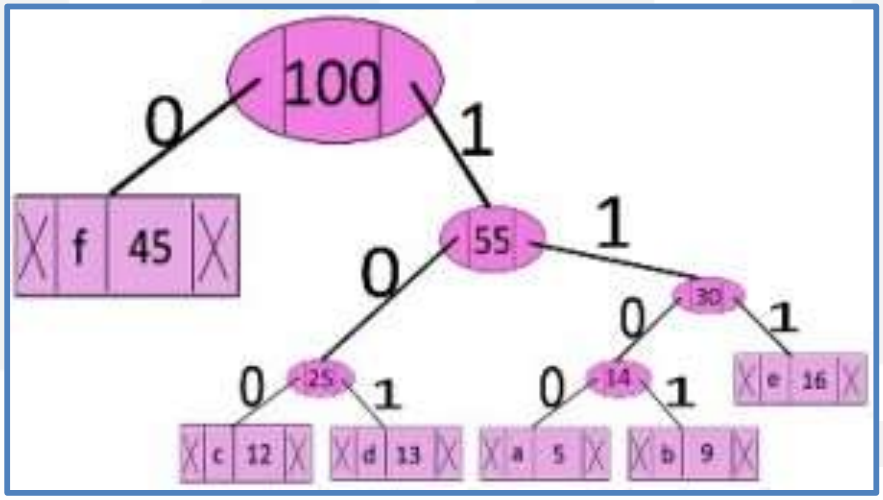
| character | Frequency |
| --- | --- |
| Internal Node | 100 |

Since the heap contains only one node, the algorithm stops here.

# Example: Huffman Coding Algorithm

***Steps to print codes from Huffman Tree:***

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



# Example: Huffman Coding Algorithm

The codes are as follows:

| character | code-word |
|---|---|
| f | 0 |
| c | 100 |
| d | 101 |
| a | 1100 |
| b | 1101 |
| e | 111 |


# Adaptive Huffman Coding Algorithm

Adaptive Huffman Coding is also known as Dynamic Huffman Coding. The implementation is done using Vitter Algorithm.

# Adaptive Huffman Coding Algorithm (ENCODING)

Adaptive Huffman coding for a string containing alphabets: Let m be the total number of alphabets. So m = 26. For Vitter Algorithm, find a parameters e & r such that

$m = 2e + r$ and $0 \leq r \leq 2e$

Therefore, for m = 26 we get e = 4 & r = 10

There are two type of code NYT Code & Fixed Code.

NYT code = Traversing tree from the root node to that particular NYT node.


# Adaptive Huffman Coding Algorithm (ENCODING)

For Fixed Code, it can be calculated from the following two conditions:

1. If $0 \leq k \leq 2r$ Then the letter Sk is encoded as the binary representation of (k-1) in (e+1) bits. (where k is position of alphabet in sorted order)
2. Else the letter Sk is encoded as the binary representation of (k-r-1) in e bits.

# Adaptive Huffman Coding Algorithm (ENCODING)

**Tree Updation:**

Tree Updation in Vitter Algorithm follows Implicit Numbering. In Implicit numbering,

- Nodes are numbered in increasing order i.e., by level and from left to right
- The Nodes that have the same weight and the type together form a block
- Blocks are related to each other as by increasing order of their weights
- Internal Node is represented by Oval shape. Weight of internal nodes = Sum of child node weights
- External Node is represented by a square shape. Weight of external nodes = Initially 1 and if repeated then increased the weight by 1

# Adaptive Huffman Coding Algorithm (ENCODING)

**Steps for Tree Updation:**

1. Initialize the tree with the NYT Node
2. For a symbol is recognized for the first time, the initial NYT node is further divided into an NYT Node and new Node initialize to that symbol and weight = 1.
3. Assign the sum of the weight of child nodes to the parent node
4. If a repeated symbol is encountered than weights are updated to that symbol.

**Note:** During Updation in Tree if the weight of the left subtree is greater than the right subtree, then nodes must be swapped.

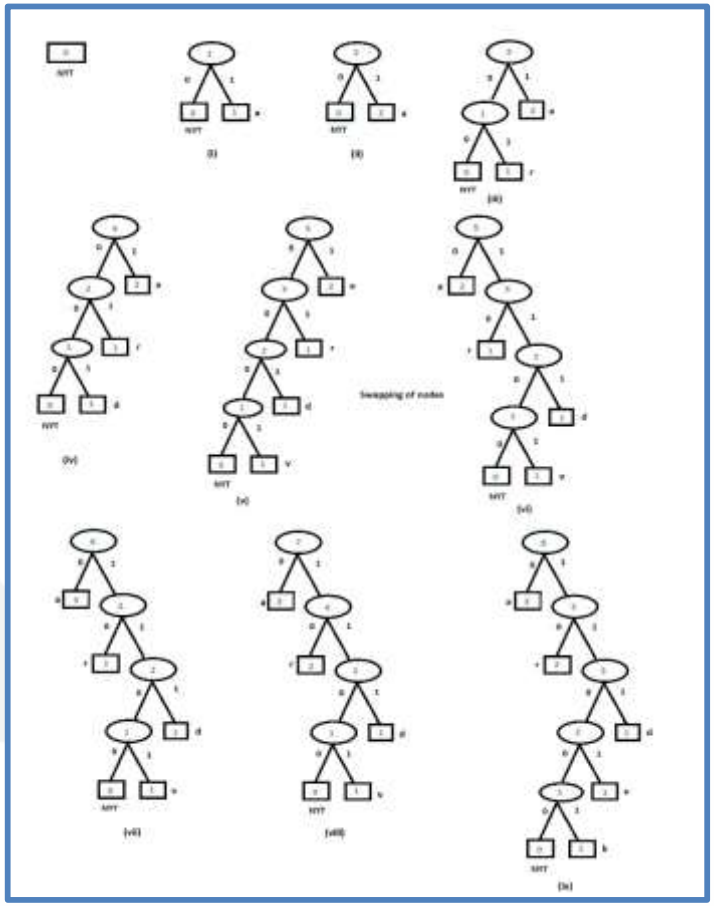
# Adaptive Huffman Coding Algorithm (ENCODING)

**code = "aardvark"**

The final Code we get is:

00000 1 010001 0000011 0001011 0 10 110001010

a a r d v a r k



# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

For string code = “aardvark”, e = 5, r = 10

As shown in the above image Tree is initialize with NYT Node with weight 0.

1. For symbol ‘a’, k = 1.

NYT Code = "" (initially tree is empty)

For Fixed Code: As k < 2r i.e, 1 < 2*10, satisfy condtion (1)

So Fixed Code is Binary Representation of (k-1) = 0 as 5-bit representation

Fixed Code = "00000“

Huffman Code for symbol for 'a' is "00000"

# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

2. For symbol ‘a’ which already exists in the tree. Traversing Tree up to symbol ‘a’, we get code = “1”

Huffman Code for symbol for 'a' is "1“

# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

3. For symbol ‘r’, k = 18.

NYT Code = "0" (traversing up to NYT Node)

For Fixed Code: As k > 2r i.e, 18 > 2*10, satisfy condtion (2)
So Fixed Code is Binary Representation of (k-1 = 17) as 5-bit representation

Fixed Code = "10001“

Huffman Code for symbol for 'r' is "010001"

# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

4. For symbol ‘d’, k = 4

NYT Code = "000" (traversing up to NYT Node)

For Fixed Code: As k < 2r i.e, 4 < 2*10, satisfy condtion (1)
So Fixed Code is Binary Representation of (k-1 = 3) as 5-bit representation

Fixed Code = "00011“

Huffman Code = "00000011"

# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

5. For symbol ‘v’, k = 22

NYT Code = "000" (traversing up to NYT Node)

For Fixed Code: As k > 2r i.e, 22 > 2*10, satisfy condtion (2)
So Fixed Code is Binary Representation of (k-r-1 = 11) as 4-bit representation

Fixed Code = “1011“

Huffman Code = "0001011"


# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

6. Swap the node of left subtree and right as the tree is violating property

7. For symbol ‘a’ which already exists in the tree. Traversing Tree up to symbol ‘a’, we get code = “0”

Huffman Code for symbol for 'a' is "0"


# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

8. For symbol ‘r’ which already exists in the tree. Traversing Tree up to symbol ‘a’, we get code = “10”

Huffman Code for symbol for 'r' is "10"

# Adaptive Huffman Coding Algorithm (ENCODING)

**Explanation:**

9. For symbol ‘k’, k = 11.

NYT Code = "1100" (traversing up to NYT Node)

For Fixed Code: As k < 2r i.e, 11 < 2*10, satisfy condtion (1)
So Fixed Code is Binary Representation of (k-1 = 10) as 5-bit representation

Fixed Code = "01010"

Huffman Code for symbol for 'r' is "110001010"

# Adaptive Huffman Coding Algorithm (DECODING)

**Steps for Decoding:**

1. Read Binary string
2. If encountered leaf node is NYT
   - Read next e bits
     1. If e bit value < r, Then to get required symbol convert (e+1) bits to decimal value of (e+1) bits + 1
     2. If e bit value > r, Then to get required symbol convert e bits to decimal value of e bits + r + 1

# Adaptive Huffman Coding Algorithm (DECODING)

**Example:**

**code** = "00000101000100000110001011010110001010"

We get final decoded code as

| 00000 | 1 | 0 | 10001 | 00 | 00011 | 000 | 1011 | 0 | 10 | 1100 | 01010 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | NYT | r | NYT | d | NYT | v | a | r | NYT | k |

# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Begin decoding by reading first e bits. So the first 4 bits are 0000, converting into decimal = 0.

  Now the value 0 < r , i.e, 0 < 10 satisfy condition (1). Now according to the condition (1), convert first e+1 = 5 bit into decimal and add 1 to it.

**00000 = 0**

**0 + 1 = 1, which is value for alphabet a.**

Update the tree and add a node for the symbol ‘a’ in the tree

# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Read the next bit in the given code and traverse the tree. We reach the external leaf node ‘a’. So the next decoded symbol is ‘a’.
- Read the next set of bits given code and traverse the tree. We have 0 as NYT Node. After reaching the NYT Node, read e bits which are 1000. Convert 1000 to decimal is 8. As 8 < r satisfy condition (1).

  Now Convert e+1 bits in decimal and add 1 to it.

  10001 = 17

  17 + 1 = 18, which is value for alphabet r.

Update the tree and add a node for the symbol ‘r’ in the tree.

# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Reading the next set of bits and traversing the Tree we reach NYT node at 00. Read e bits which are 0001. Convert 0001 to decimal is 1. As 1 < r satisfy condition (1).

  Now Convert e+1 bits in decimal and add 1 to it.

  00011 = 3

  3 + 1 = 4, which is value for alphabet d.

Update the tree and add a node for the symbol ‘d’ in the tree.

# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Reading the next set of bits and traversing the Tree we reach NYT node at 000. Read e bits which are 1011. Convert 1011 to decimal is 11. As 11 > r satisfy condition (2).

  Now Convert k+r+1 bits in decimal and decode the symbol.

  10110 = 22, which is value for alphabet v.

Update the tree and add a node for the symbol ‘v’ in the tree.


# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Reading the next set of bits and traversing the Tree we get symbol ‘a’ at 0. Update the tree and add a node for the symbol ‘a’ in the tree.
- Reading the next set of bits and traversing the Tree we get symbol ‘r’ at 10. Update the tree and add a node for the symbol ‘a’ in the tree.

# Adaptive Huffman Coding Algorithm (DECODING)

**Explanation:**

- Reading the next set of bits and traversing the Tree we reach NYT node at 1100. Read e bits which are 0101. Convert 0101 to decimal is 9. As 9 < r satisfy condition (1).

  Now Convert e+1 bits in decimal and and add 1 to it.

  01000 = 8,

  8 + 1 = 9. which is value for alphabet k.

Update the tree and add a node for the symbol ‘v’ in the tree.

# Application of Huffman Coding

- Arithmetic coding and Huffman coding produce equivalent results — achieving entropy — when every symbol has a probability of the form $1/2^k$. In other circumstances, arithmetic coding can offer better compression than Huffman coding because — intuitively — its "code words" can have effectively non-integer bit lengths, whereas code words in prefix codes such as Huffman codes can only have an integer number of bits.
- Therefore, a code word of length $k$ only optimally matches a symbol of probability $1/2^k$ and other probabilities are not represented optimally; whereas the code word length in arithmetic coding can be made to exactly match the true probability of the symbol. This difference is especially striking for small alphabet sizes.

# Application of Huffman Coding

- Prefix codes nevertheless remain in wide use because of their simplicity, high speed, and lack of patent coverage. They are often used as a "back-end" to other compression methods.
- DEFLATE (PKZIP's algorithm) and multimedia codecs such as JPEG and MP3 have a front-end model and quantization followed by the use of prefix codes; these are often called "Huffman codes" even though most applications use pre-defined variable-length codes rather than codes designed using Huffman's algorithm.


# Application of Huffman Coding

- Loss Less Image Compression
- Text Compression
- Audio Compression

# Application of Huffman Coding

- **Loss Less Image Compression**

**Lossless compression** is a class of data **compression** algorithms that allows the original data to be perfectly reconstructed from the **compressed** data. ... Some **image** file formats, like PNG or GIF, use only **lossless compression**, while others like TIFF and MNG may use either **lossless** or lossy methods


# Application of Huffman Coding

- **Text Compression**

**Text compression** algorithms aim at statistical reductions in the volume of data. One commonly used **compression** algorithm is Huffman coding [Huf52], which makes use of information on the frequency of characters to assign variable-length codes to characters.

# Application of Huffman Coding

- **Audio Compression**

Music **compression** is the process of reducing a signal's dynamic range. Dynamic range is the difference between the loudest and quietest parts of an **audio** signal. You need to reduce the dynamic range of most **audio** signals for them to **sound** natural on a recording.

# DIGITAL LEARNING CONTENT


Parul® University

www.paruluniversity.ac.in