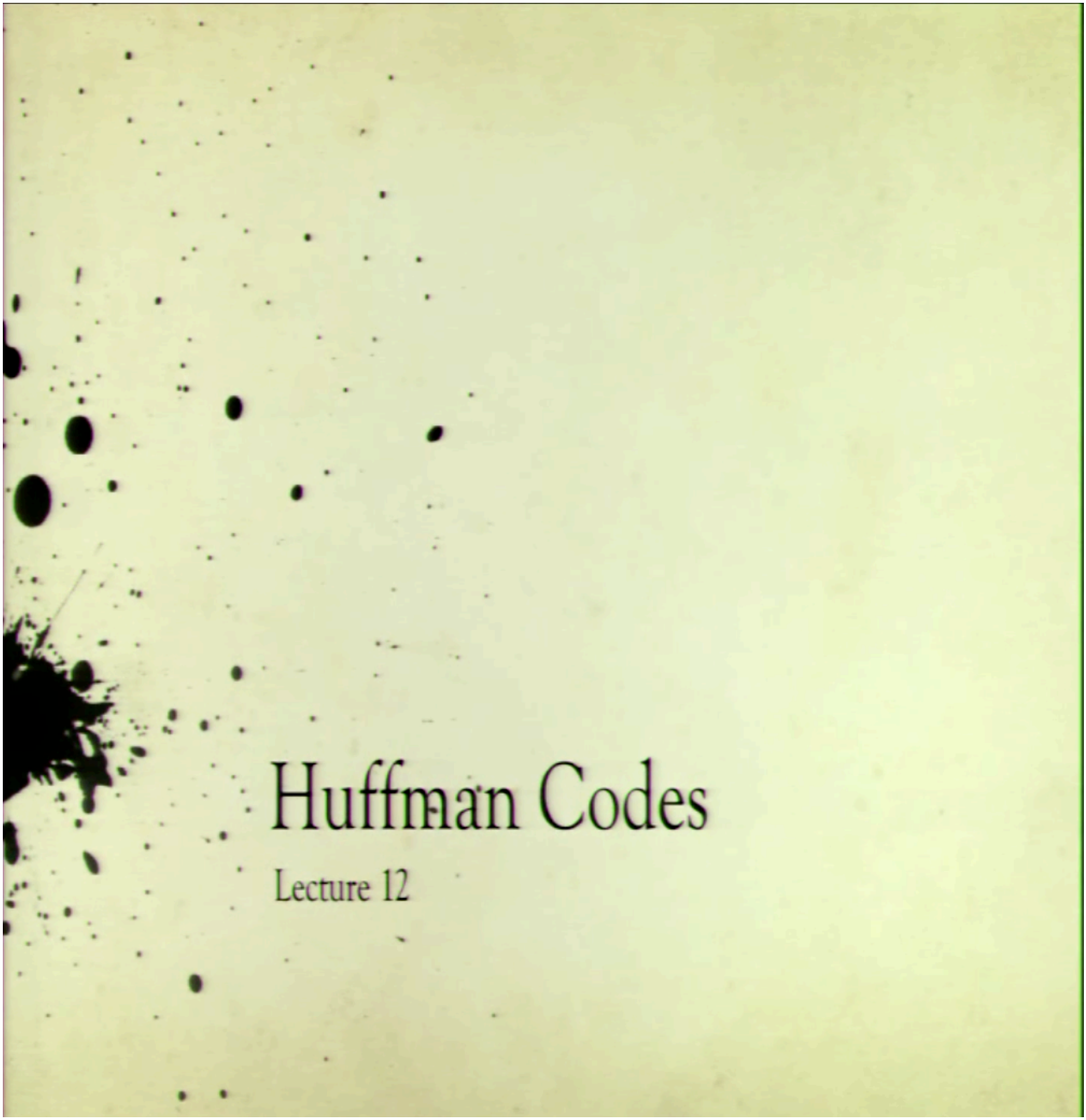


Huffman Codes

Lecture 12

Data Encoding

- Suppose we want to **encode** the alphabet in *binary*.
  - How many *bits* do we need to be able to encode every letter?
  - Five bits suffice as $2^5 \geq 26$

a	00000	i	01000	q	10000	y	11000
b	00001	j	01001	r	10001	z	11001
c	00010	k	01010	s	10010	.	11010
d	00011	l	01011	t	10011	,	11011
e	00100	m	01100	u	10100	:	11100
f	00101	n	01101	v	10101	;	11101
g	00110	o	01110	w	10110	?	11110
h	00111	p	01111	x	10111	!	11111

The Cost of an Encoding

- How do we measure the quality of this encoding?
- The most natural measure would be the **length** of the encoding.
  - So if f_i is the *frequency* at which letter i appears in alphabet A then:

$$\text{cost} = \sum_{i \in A} \ell_i \cdot f_i = \sum_{i \in A} 5 f_i$$

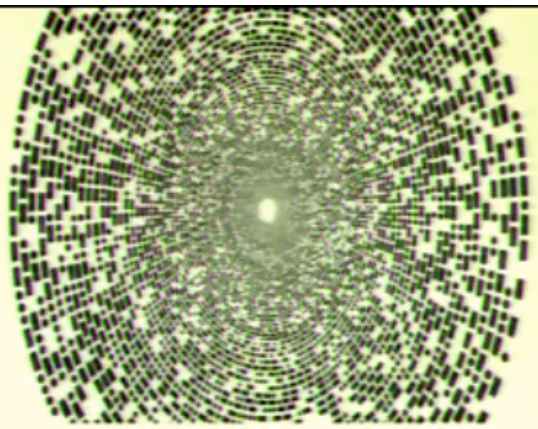
The Cost of an Encoding

- But some letters appear more often than others.

a	8.12	i	7.31	q	0.11	y	2.11
b	1.49	j	0.10	r	6.02	z	0.07
c	2.71	k	0.69	s	6.28		
d	4.32	l	3.98	t	9.10		
e	12.02	m	2.81	u	2.88		
f	2.30	n	6.95	v	1.11		
g	2.03	o	7.68	w	2.09		
h	5.92	p	1.82	x	0.17		

- So we could reduce the cost if more frequent letters have smaller representations than less frequent letters.
- *Indeed a well-known code does exactly this...*

Morse Code

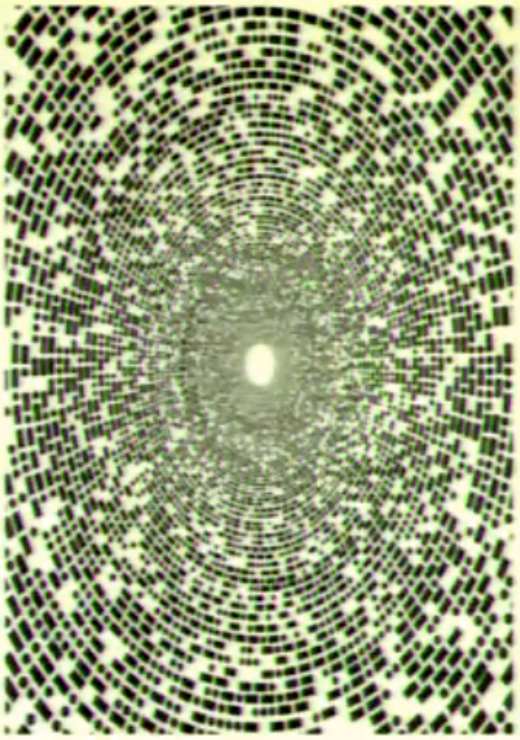


- The Morse Code is:

a	01	i	00	q	1101	y	1011
b	1000	j	0111	r	010	z	1100
c	1010	k	101	s	000		
d	100	l	0100	t	1		
e	0	m	11	u	001		
f	0010	n	10	v	0001		
g	110	o	111	w	011		
h	0000	p	0110	x	1001		

- Note that, by allowing a different number of bits per letter, only a maximum of four bits are required as: $2^1 + 2^2 + 2^3 + 2^4 \geq 26$

a	01	i	00	q	1101	y	1011
b	1000	j	0111	r	010	z	1100
c	1010	k	101	s	000		
d	100	l	0100	t	1		
e	0	m	11	u	001		
f	0010	n	10	v	0001		
g	110	o	111	w	011		
h	0000	p	0110	x	1001		



A Malfunction?

- But there is a big problem with this code. It is ambiguous!

  e.g. "1101" = "q" "1101" = "ttet"

  "1101" = "ma" "1101" = "gt"

  "1101" = "tk" "1101" = "met"

- *Morse code does work, though. How?*

a	01*	i	00*	q	1101*	y	1011*
b	1000*	j	0111*	r	010*	z	1100*
c	1010*	k	101*	s	000*		
d	100*	l	0100*	t	1		
e	0*	m	11*	u	001*		
f	0010*	n	10*	v	0001*		
g	110*	o	111*	w	011*		
h	0000*	p	0110*	x	1001*		

A Ternary Code

- Morse code is actually a **ternary code** not a binary code.
- It has a third symbol, "**pause**" or " * ", that it places after each letter.

e.g. " 1101* " = "q"

" 11*01* " = "ma"

" 1*101* " = "tk"

" 1*1*0*1* " = "ttet"

" 110*1* " = "gt"

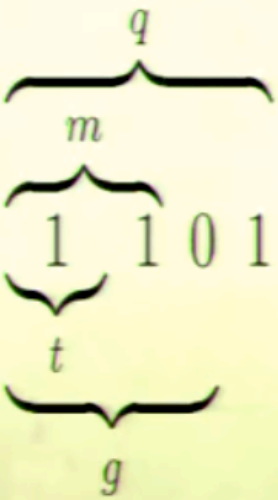
" 11*0*1* " = "met"

- *Is it possible to avoid ambiguities with a true binary code?*

Prefix Codes

- To answer this question we must consider **prefix codes.**
- A coding system is prefix-free if no codeword is a prefix of another codeword.
  - Morse code is **not** prefix free.

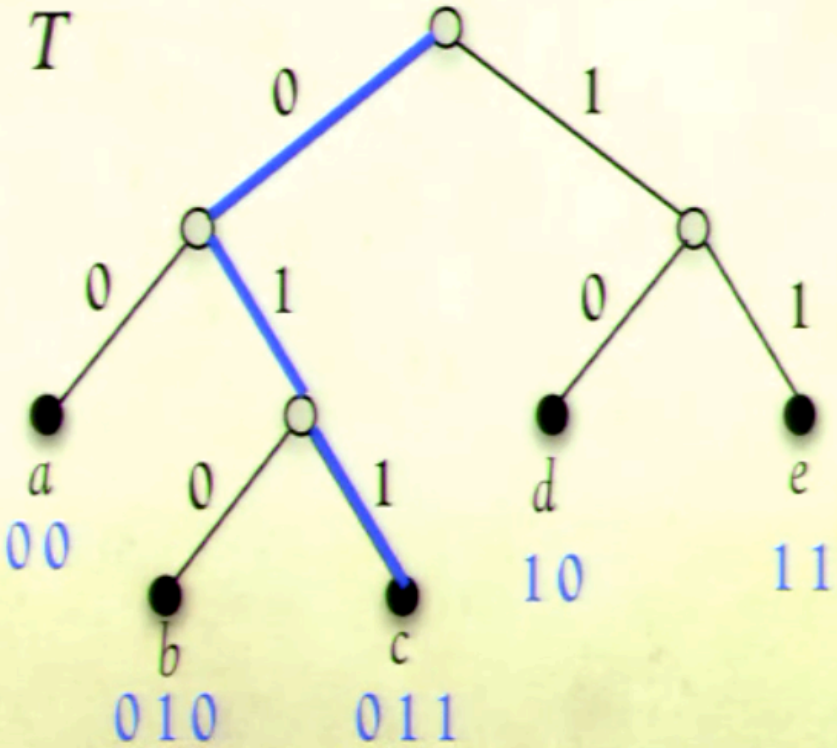
e.g.



Binary Tree Representations

- We can use **binary tree** to represent a prefix-free binary code:
  - There is a binary tree T.
  - Each left edge has **label 0** and each right edge has **label 1**.
  - The *leaf vertices* are the *letters* of the alphabet.
  - The **codeword** for a letter are the labels on the path from root to leaf.

e.g. T



A Characterization of Unambiguous Codes

Theorem. A binary coding system is prefix-free *if and only if* it has a binary tree representation.

Proof.

$(\Leftarrow)$

- In a binary tree representation the letters are at the **leaves**.
- This means the path P_x from the root to leaf x and the path P_y to a leaf y must diverge at some point.

  $\Longrightarrow$ The codeword for x **cannot** be a prefix of the codeword for y.

$(\Rightarrow)$

- Given a binary coding system we define the binary tree *recursively*.
- A letter whose code word starts with a 0 is placed in the **left subtree**, otherwise it is placed in the **right subtree**.

□

Observation 1: $\text{cost}(T) = \sum_{i \in A} f_i \cdot d_i(T)$

Proof.

$$\begin{aligned}
\text{cost}(T) &= \sum_{i \in A} f_i \cdot \ell_i(T) \\
&= \sum_{i \in A} f_i \cdot \sum_{e : e \in P_i} 1 \\
&= \sum_{i \in A} f_i \cdot d_i(T)
\end{aligned}$$

□

Observation 1: $\text{cost}(T) = \sum_{i \in A} f_i \cdot d_i(T)$

$A = \{a, b, c, d, e\}$ Depth

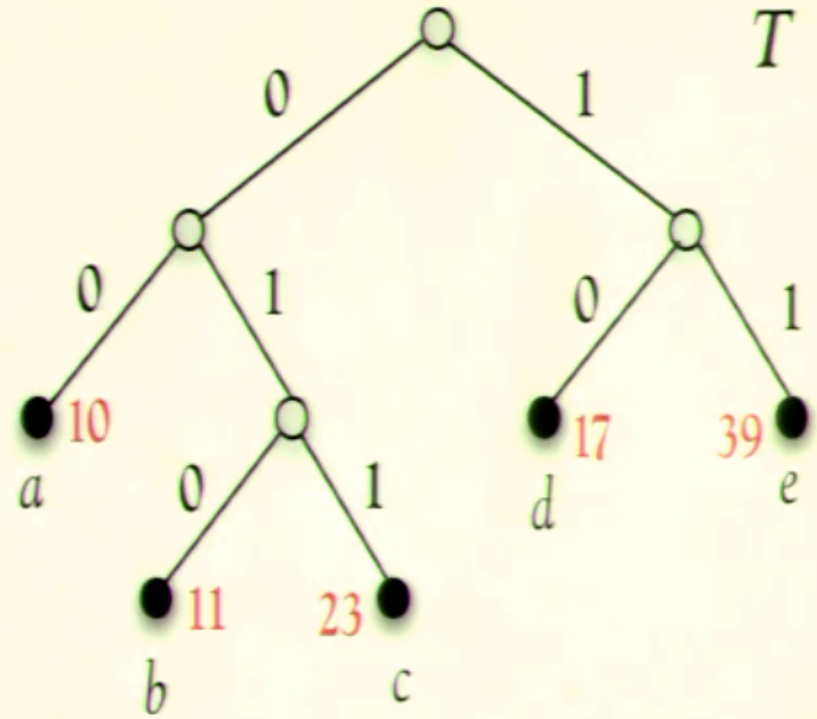
$f_a = 10$

$f_b = 11$

$f_c = 23$

$f_d = 17$

$f_e = 39$



$$\text{cost}(T) = \sum_{i \in A} f_i \cdot d_i(T)$$

$$= 2 \cdot (10 + 17 + 39) + 3 \cdot (11 + 23)$$

$$= 234$$

Letter to Leaf Assignment

Observation 1: $\text{cost}(T) = \sum_{i \in A} f_i \cdot d_i(T)$

- Observation 1 tells us something very useful.
  - If we must use a tree of **shape** T then it is easy to determine the best way to assign the letters to the leaves.
  - Specifically, **sort** the letters in increasing order of frequency.
  - Then iteratively assign the *least* frequent letter to the *deepest* leaf.
- *Let's investigate what this implies about our previous example...*

$\mathcal{A} = \{a, b, c, d, e\}$ Depth

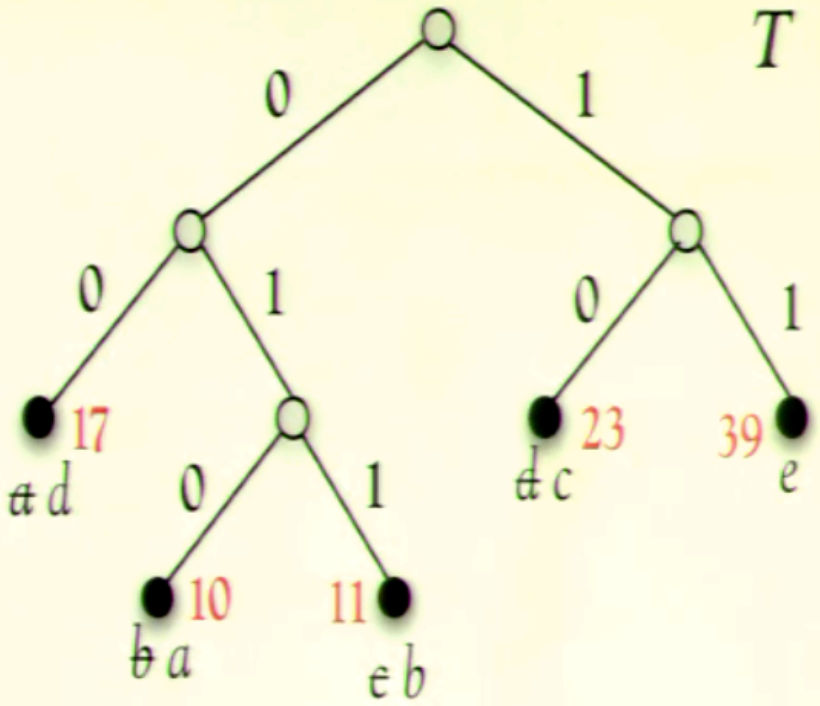
$f_a = 10$

$f_b = 11$

$f_c = 23$

$f_d = 17$

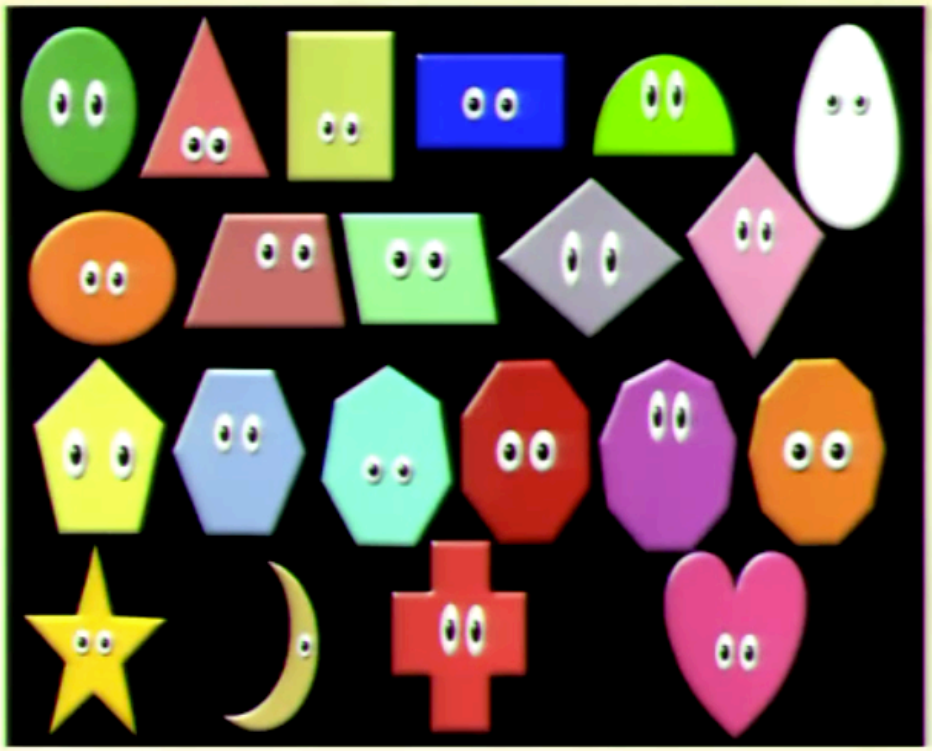
$f_e = 39$



$$\text{cost}(T) = \sum_{i \in \mathcal{A}} f_i \cdot d_i(T)$$

$$= 2 \cdot (17 + 23 + 39) + 3 \cdot (10 + 11)$$

$$= 221$$



But Observation 1 **does not** *tell us what the best shape for the tree is.*

To understand this, we require a second observation...

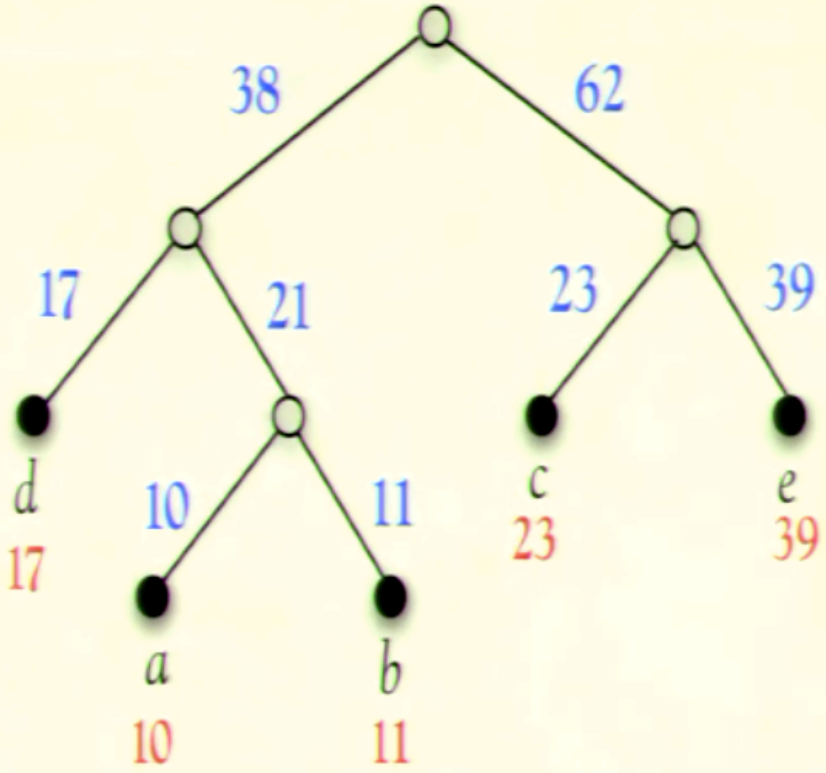
- Let $n_e = \sum_{i \in A: e \in P_i} f_i$ be the number of letters (*weighted by frequency*) whose root-leaf paths use edge e in T.

Observation 2: $\text{cost}(T) = \sum_{e \in T} n_e$

Proof. $$\begin{aligned}
\text{cost}(T) &= \sum_{i \in A} f_i \cdot d_i(T) \\
&= \sum_{i \in A} f_i \cdot \sum_{e: e \in P_i} 1 \\
&= \sum_{e \in T} 1 \cdot \sum_{i \in A: e \in P_i} f_i \\
&= \sum_{e \in T} \sum_{i \in A: e \in P_i} f_i \\
&= \sum_{e \in T} n_e
\end{aligned}$$

□

Observation 2: $\text{cost}(T) = \sum_{i \in A} f_i \cdot |P_i|$

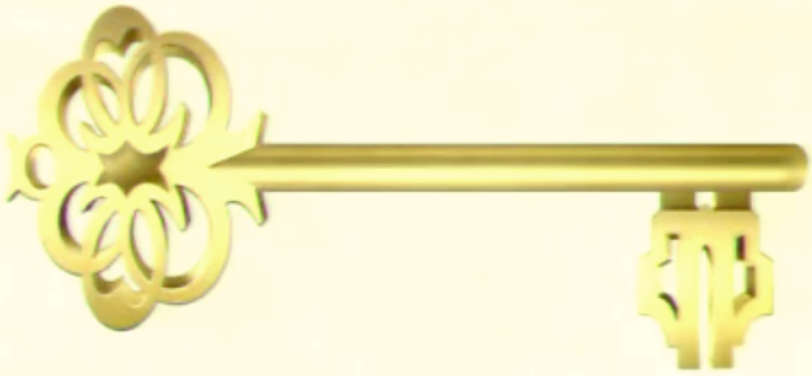


$$\text{cost}(T) = \sum_{e \in T} n_e$$

$$= 38 + 62 + 17 + 21 + 23 + 39 + 10 + 11$$

$$= 221$$

The Key Formula



- The key now to designing a good coding system is the following formula:

Key Formula. Let $\hat{T}$ be the tree formed from T by removing a pair of sibling-leaves a and b and labelling its parent by z where $f_x = f_a + f_b$. Then:

$$\text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$$

- *As always, let's illustrate this via our example...*

$\mathcal{A} = \{a, b, c, d, e\}$

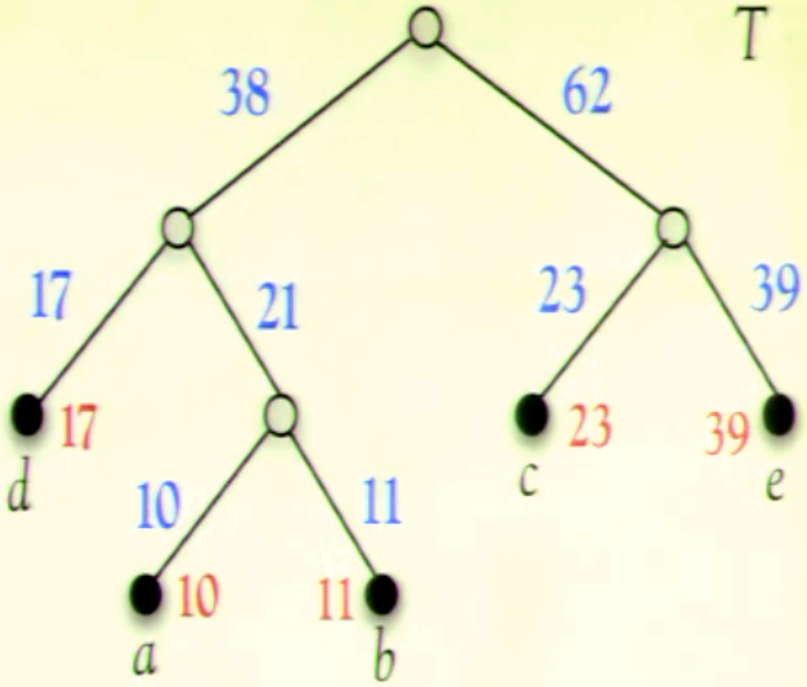
$f_a = 10$

$f_b = 11$

$f_c = 23$

$f_d = 17$

$f_e = 39$



$\text{cost}(T) = 221$

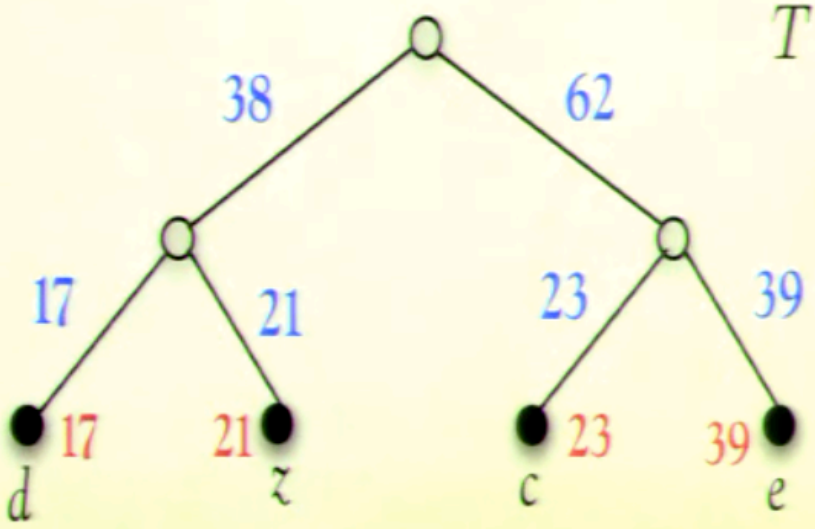
$\mathcal{A} = \{z, c, d, e\}$

$f_z = 21$

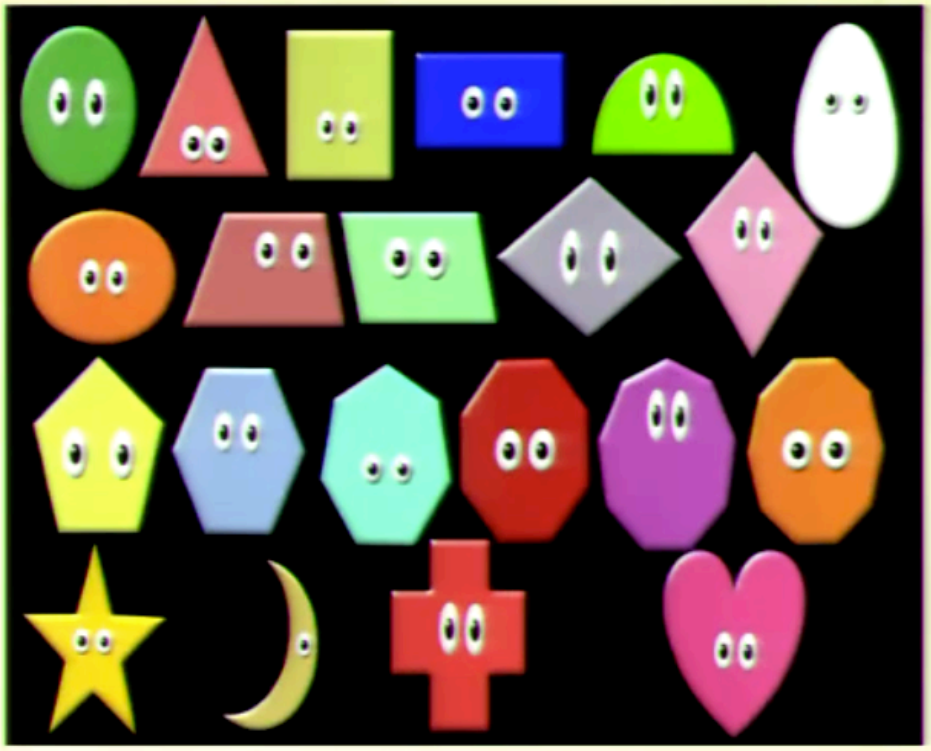
$f_c = 23$

$f_d = 17$

$f_e = 39$



$\text{cost}(\hat{T}) = 200 \implies \text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$



Observation 1 tells us the two least frequent letters should be sibling-leaves.

Observation 2 tells us how to then recursively find the optimal shape of the tree.

Huffman's Algorithm

Huffman Coding [Huffman 1952]

Huffman(A, f)

If A has two letters **then**

Encode one letter with 0 and the other with 1.

Otherwise

Let a and b be the most infrequent letters.

Set $\hat{A} \leftarrow (A \setminus \{a, b\}) \cup \{z\}$ and set $f_z = f_a + f_b$

Let $\hat{T} =$ **Huffman**$(\hat{A}, \mathrm{f})$

Create T by adding a and b as children of the leaf z in $\hat{T}$.

$A = \{a, b, c, d, e\}$

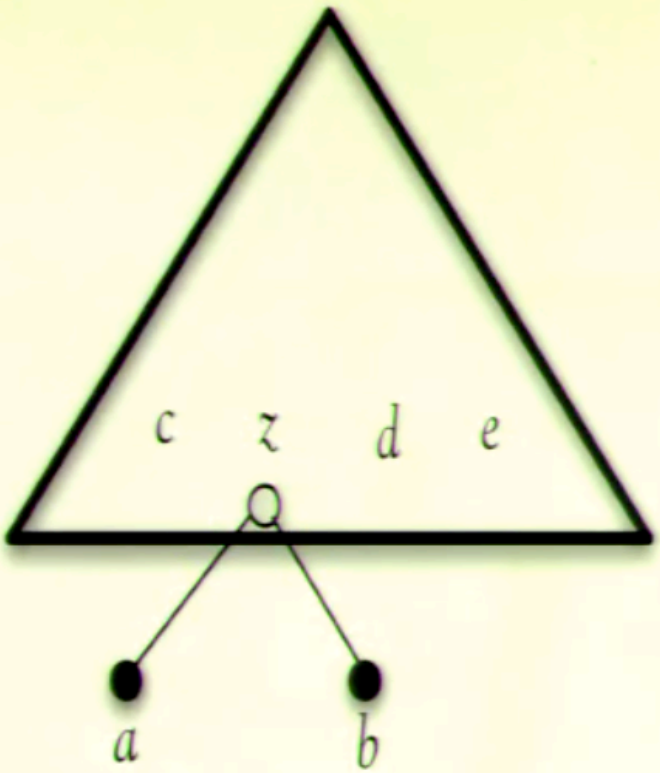
$f_a = 10$

$f_b = 11$

$f_c = 23$

$f_d = 17$

$f_e = 39$



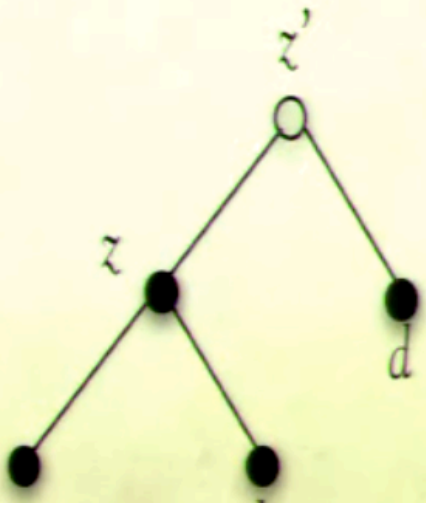
$A = \{z, c, d, e\}$

$f_z = 21$

$f_c = 23$

$f_d = 17$

$f_e = 39$



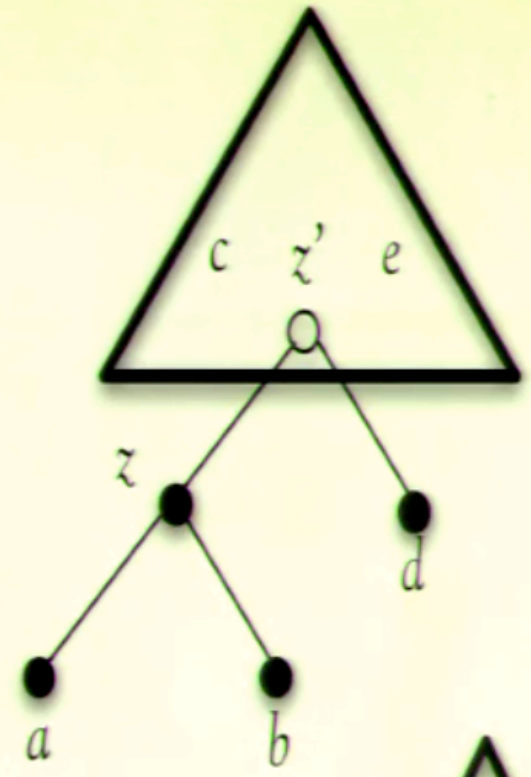
$A = \{z, c, d, e\}$

$f_z = 21$

$f_c = 23$

$f_d = 17$

$f_e = 39$

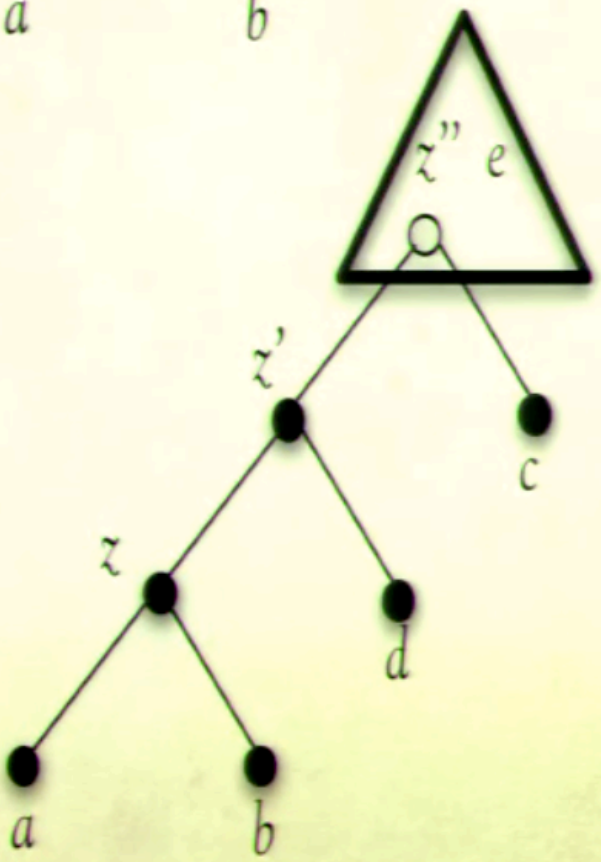


$A = \{z', c, e\}$

$f_{z'} = 38$

$f_c = 23$

$f_e = 39$

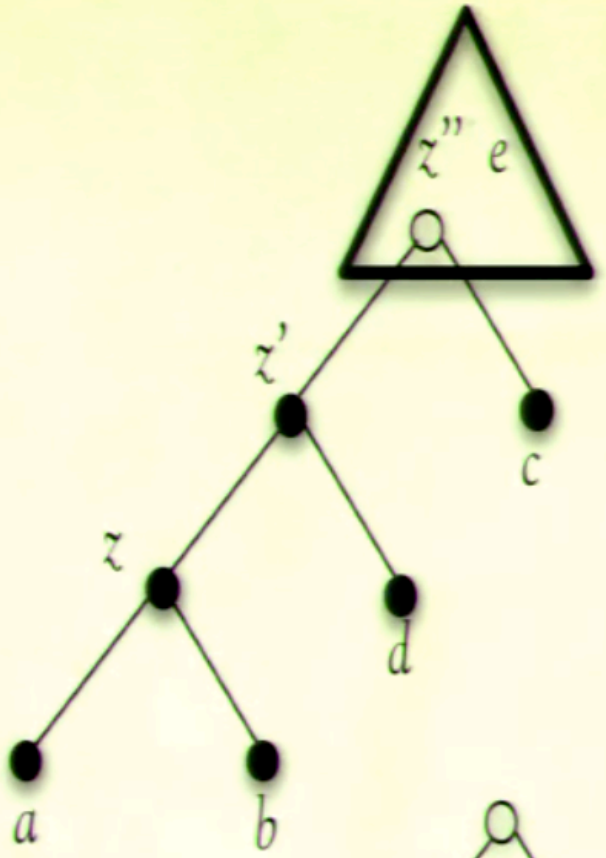


$A = \{z', c, e\}$

$f_{z'} = 38$

$f_c = 23$

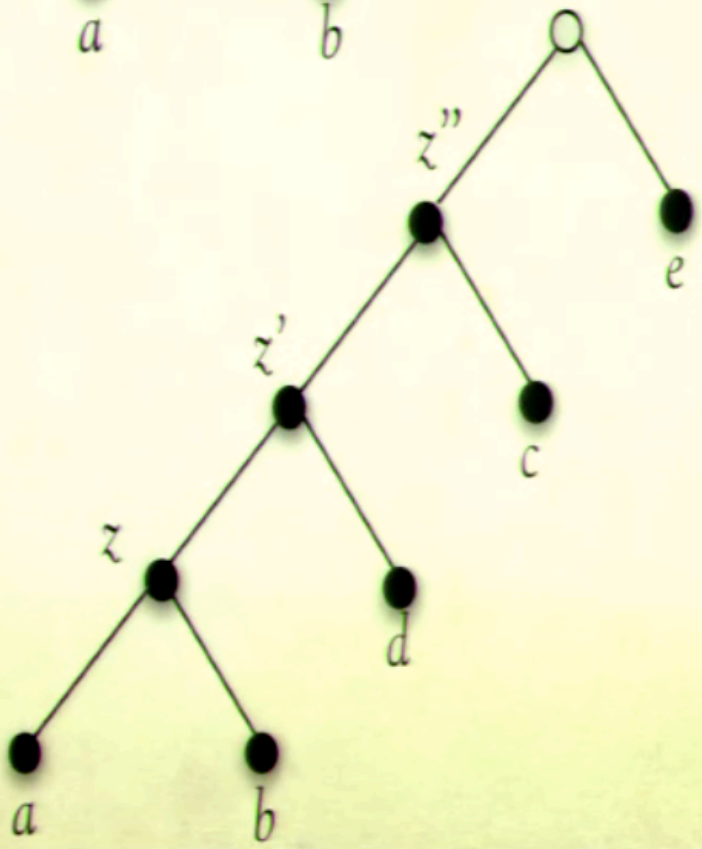
$f_e = 39$

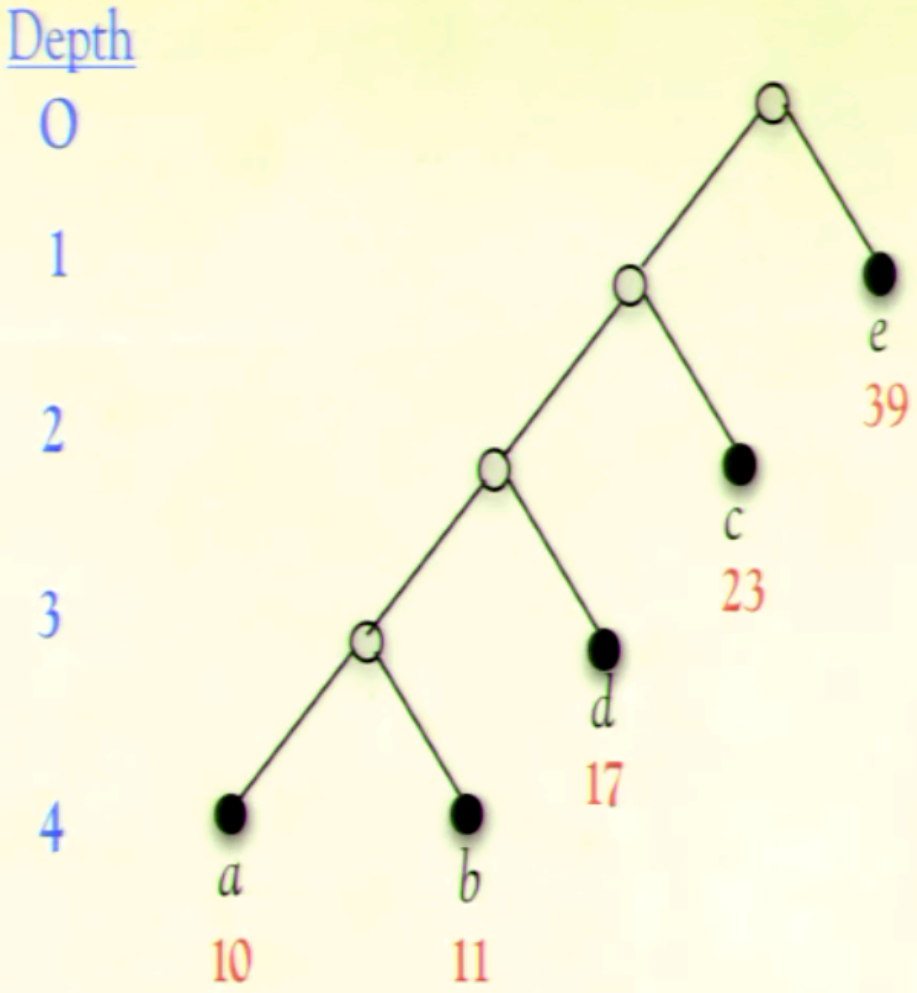


$A = \{z'', e\}$

$f_{z''} = 61$

$f_e = 39$





$$\text{cost}(T) = \sum_{i \in \mathcal{A}} f_i \cdot d_i(T)$$

$$= 1 \cdot 39 + 2 \cdot 23 + 3 \cdot 17 + 4 \cdot (10 + 11)$$

$$= 220$$

Theorem. The Huffman Coding algorithm gives the minimum cost encoding.

Proof.

Base Case: $|\mathcal{A}| = 2$

- Both letters are given **codewords** of length 1 so this cannot be improved.

Induction Hypothesis

- Assume Huffman Coding works if $|\mathcal{A}| = k$.

Induction Step:

- Assume $|\mathcal{A}| = k + 1$.
- Let a and b be the *least frequent* letters.
- Then a and b are siblings in an optimal solution and for any $\hat{T}$

$$\text{cost}(T) = \text{cost}(\hat{T}) + f_a + f_b$$

- So the best choice for $\hat{T}$ is the optimal solution for $\hat{\mathcal{A}}$.
- But, by the induction hypothesis, this is exactly the choice made recursively by the algorithm.

□

The Running Time

- How long does the Huffman coding algorithm take?
  - There are n-2 iterations.
  - In each iteration it takes $O(n)$ time to find the *two least frequent letters* and update the alphabet.

$$\Longrightarrow \quad \text{Runtime} = O(n^2)$$

- In fact, with the use of a data structures called a **heap** we can implement the algorithm in time $O(n \cdot \log n)$.

Applications of Huffman Codes

- Huffman coding lies at the heart of modern **data compression:**

ZIP File Compression

MP3 Audio Files

JPEG Video Files