Source Coding

(a subset of Data Compression)

Definition A *source code* C for a random variable X is a mapping from $\mathcal{X}$, the range of X, to $\mathcal{D}^*$, the set of finite-length strings of symbols from a D-ary alphabet. Let $C(x)$ denote the codeword corresponding to x and let $l(x)$ denote the length of $C(x)$.

For example, $C(\text{red}) = 00$, $C(\text{blue}) = 11$ is a source code for $\mathcal{X} = \{\text{red}, \text{blue}\}$ with alphabet $\mathcal{D} = \{0, 1\}$.

$$\underbrace{\mathcal{X} \longrightarrow}_{C(x)} \mathcal{D}^*$$

In almost all cases, we consider D=2 (i.e., 2 bits)

Definition The *expected length* $L(C)$ of a source code $C(x)$ for a random variable X with probability mass function $p(x)$ is given by

$$L(C) = \sum_{x \in \mathcal{X}} p(x) l(x),$$

where $l(x)$ is the length of the codeword associated with x.

***Example* 1** Let X be a random variable with the following distribution and codeword assignment:

$$\begin{aligned}
\Pr(X = 1) &= \tfrac{1}{2}, \quad \text{codeword } C(1) = 0 \\
\Pr(X = 2) &= \tfrac{1}{4}, \quad \text{codeword } C(2) = 10 \\
\Pr(X = 3) &= \tfrac{1}{8}, \quad \text{codeword } C(3) = 110 \\
\Pr(X = 4) &= \tfrac{1}{8}, \quad \text{codeword } C(4) = 111.
\end{aligned}$$

The entropy $H(X)$ of X is 1.75 bits, and the expected length $L(C) = El(X)$ of this code is also 1.75 bits. Here we have a code that has the same average length as the entropy. We note that any sequence of bits can be uniquely decoded into a sequence of symbols of X. For example, the bit string 0110111100110 is decoded as 134213.

Example 2 Consider another simple example of a code for a random variable:

$$\Pr(X = 1) = \tfrac{1}{3}, \quad \text{codeword } C(1) = 0$$
$$\Pr(X = 2) = \tfrac{1}{3}, \quad \text{codeword } C(2) = 10$$
$$\Pr(X = 3) = \tfrac{1}{3}, \quad \text{codeword } C(3) = 11.$$

Just as in Example 5.1.1, the code is uniquely decodable. However, in this case the entropy is $\log 3 = 1.58$ bits and the average length of the encoding is 1.66 bits. Here $El(X) > H(X)$.

***Example* 3** (*Morse code*) The Morse code is a reasonably efficient code for the English alphabet using an alphabet of four symbols: a dot, a dash, a letter space, and a word space. Short sequences represent frequent letters (e.g., a single dot represents E) and long sequences represent infrequent letters (e.g., Q is represented by "dash,dash,dot,dash").

Bijective mapping (invertible mapping):

Definition A code is said to be *nonsingular* if every element of the range of X maps into a different string in $\mathcal{D}^*$; that is,

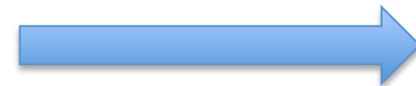
$$x \neq x' \Rightarrow C(x) \neq C(x').$$

...es suficiente....

Nonsingularity suffices for an unambiguous description of a single value of X. But we usually wish to send a sequence of values of X.

In such cases we can ensure decodability by adding a special symbol (a "comma") between any two codewords. But this is an inefficient use of the special symbol; we can do better by developing the idea of self-punctuating or instantaneous codes.

We want also to send a "stream" of bits....



In order to handle the “stream” of bits, we define:

Definition The *extension* C^* of a code C is the mapping from finite-length strings of $\mathcal{X}$ to finite-length strings of $\mathcal{D}$, defined by

$$C(x_1x_2\cdots x_n) = C(x_1)C(x_2)\cdots C(x_n),$$

where $C(x_1)C(x_2)\cdots C(x_n)$ indicates concatenation of the corresponding codewords.

Example If $C(x_1) = 00$ and $C(x_2) = 11$, then $C(x_1x_2) = 0011$.

Definition A code is called *uniquely decodable* if its extension is nonsingular.

In other words, any encoded string in a uniquely decodable code has only one possible source string producing it.

IMPORTANT OBSERVATION →→→→→ However, one may have to look at the entire string to determine even the first symbol in the corresponding source string.

The name should be “prefix-free” code

… in some codes, this is not needed:

Definition A code is called a *prefix code* or an *instantaneous code* if no codeword is a prefix of any other codeword.

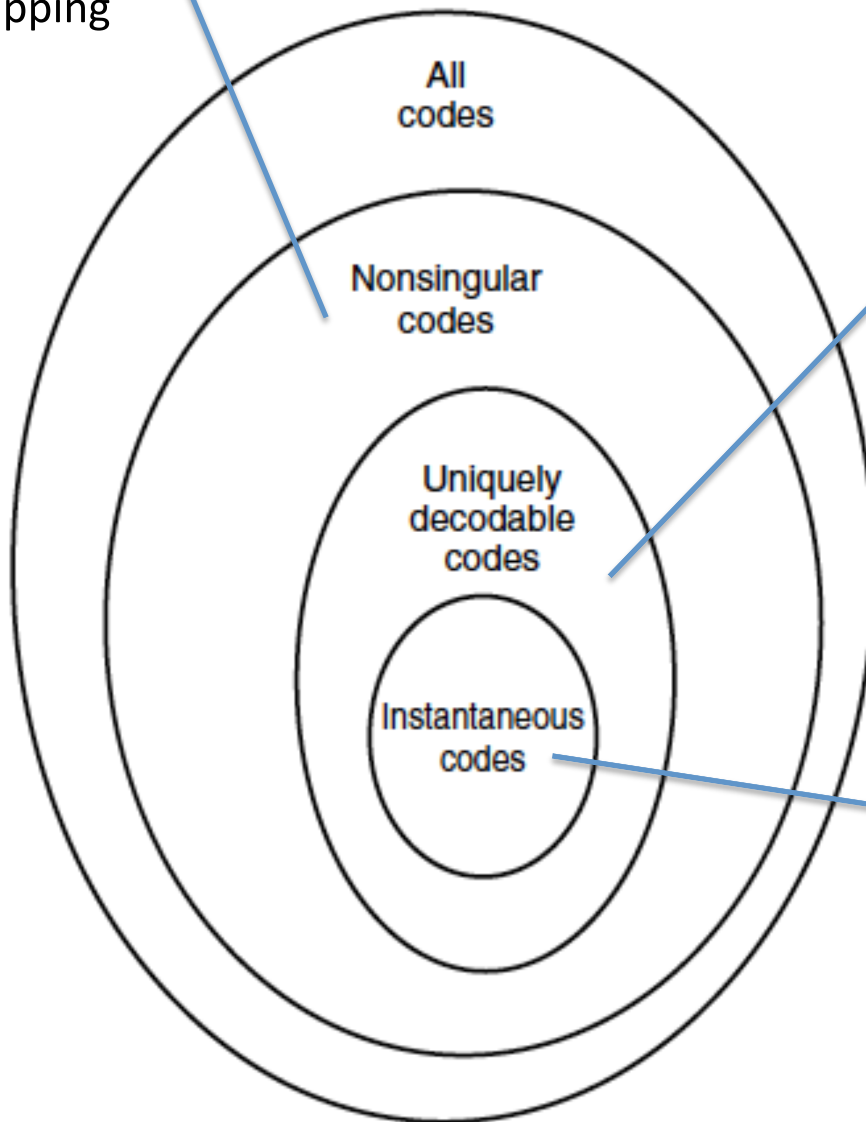
An instantaneous code can be decoded without reference to future codewords since the end of a codeword is immediately recognizable. Hence, for an instantaneous code, the symbol x_i can be decoded as soon as we come to the end of the codeword corresponding to it. We need not wait to see the codewords that come later. An instantaneous code is a *self-punctuating code*; we can look down the sequence of code symbols and add the commas to separate the codewords without looking at later symbols. For example, the binary string 01011111010 produced by the code of Example 1 is parsed as 0,10,111,110,10.

When we have “11” we know that we have to wait one bit more to decide (but we do not wait for other codeword, or the entire sequence)….

$$C(1) = 0$$
$$C(2) = 10$$
$$C(3) = 110$$
$$C(4) = 111.$$

$$x \neq x' \Rightarrow C(x) \neq C(x').$$

Bijective mapping



$$C(x_1 x_2 \cdots x_n) = C(x_1)C(x_2)\cdots C(x_n)$$

Bijective mapping of the extended code, creating a sequence, a concatenation of bits.

We can decode instantaneously, we know when a transmission of a codeword “starts” and “finishes” (we do not need to see all the sequence, we can decode “online”)

TABLE 1 Classes of Codes

X	Singular	Nonsingular, But Not Uniquely Decodable	Uniquely Decodable, But Not Instantaneous	Instantaneous
1	0	0	10	0
2	0	010	00	10
3	0	01	11	110
4	0	10	110	111

Help to understand the example ➔➔➔➔➔ For the nonsingular code, the code string 010 has three possible source sequences: 2 or 14 or 31, and hence the code is not uniquely decodable. The uniquely decodable code is not prefix-free and hence is not instantaneous. To see that it is uniquely decodable, take any code string and start from the beginning. If the first two bits are 00 or 10, they can be decoded immediately. If the first two bits are 11, we must look at the following bits. If the next bit is a 1, the first source symbol is a 3. If the length of the string of 0's immediately following the 11 is odd, the first codeword must be 110 and the first source symbol must be 4; if the length of the string of 0's is even, the first source symbol is a 3.

We have to see the complete sequence...

OTHER EXAMPLES WITH CORRESPONDING TREES:

TABLE Four different codes for a four-letter alphabet.

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

A code in which no codeword is a prefix to another codeword is called a *prefix code*. A simple way to check if a code is a prefix code is to draw the rooted binary tree corresponding to the code. Draw a tree that starts from a single node (the *root node*) and has a maximum of two possible branches at each node. One of these branches corresponds to a 1 and the other branch corresponds to a 0.

FREEDOM: We can choose the bit to give to each branch!

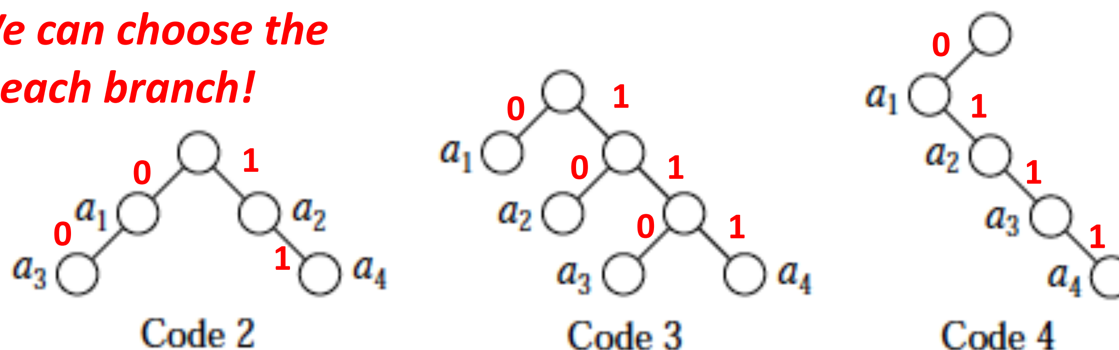


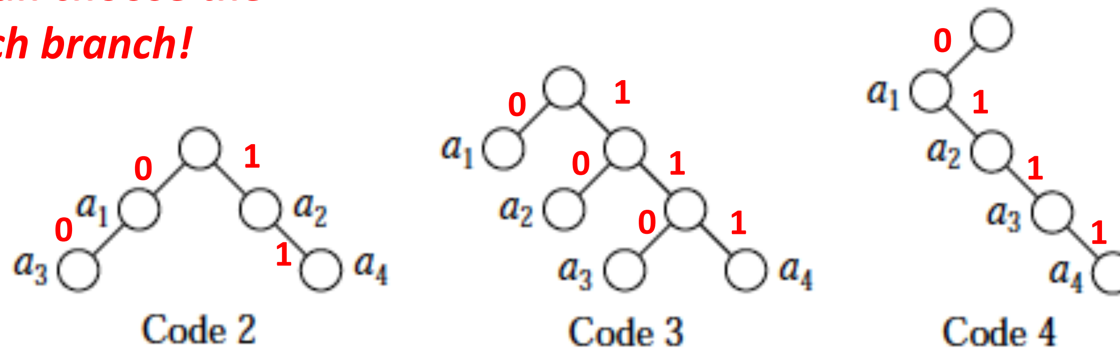
TABLE **Four different codes for a four-letter alphabet.**

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

..dan lugar...

Note that apart from the root node, the trees have two kinds of nodes—nodes that give rise to other nodes and nodes that do not. The first kind of nodes are called *internal nodes*, and the second kind are called *external nodes* or *leaves*. In a prefix code, the codewords are only associated with the external nodes. A code that is not a prefix code, such as Code 4, will have codewords associated with internal nodes.

FREEDOM: We can choose the bit to give to each branch!



Prefix code!!!

REMARK: hence, to have a prefix code, we only need to create a tree and consider the leaves of the tree as codewords.

Theorem (*Kraft inequality*) *For any instantaneous code (prefix code) over an alphabet of size D, the codeword lengths* $l_1, l_2, \ldots, l_m$ *must satisfy the inequality*

$$\sum_i D^{-l_i} \leq 1.$$

Conversely, given a set of codeword lengths that satisfy this inequality, there exists an instantaneous code with these word lengths.

In almost all cases, we consider D=2 (i.e., 2 bits)

BOUNDS AND OPTIMAL CODING

IMPORTANT:

BOUNDS ON THE OPTIMAL CODE LENGTH

$$H(X) \le L < H(X) + 1.$$

An optimal (shortest expected length) prefix code for a given distribution can be constructed by a simple algorithm discovered by Huffman

→→→→→ any other code for the same alphabet cannot have a lower expected length than the code constructed by the algorithm.

Theorem *Huffman coding is optimal; that is, if C^* is a Huffman code and C' is any other uniquely decodable code, $L(C^*) \le L(C')$.*

HUFFMAN CODING (OPTIMAL)

The Huffman encoding algorithm starts by constructing a list of all the alphabet symbols in descending order of their probabilities. It then constructs, from the bottom up, a binary tree with a symbol at every leaf. This is done in steps, where at each step two symbols with the smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codewords of the symbols.

Example Consider a random variable X taking values in the set $\mathcal{X} = \{1, 2, 3, 4, 5\}$ with probabilities 0.25, 0.25, 0.2, 0.15, 0.15, respectively. We expect the optimal binary code for X to have the longest codewords assigned to the symbols 4 and 5.

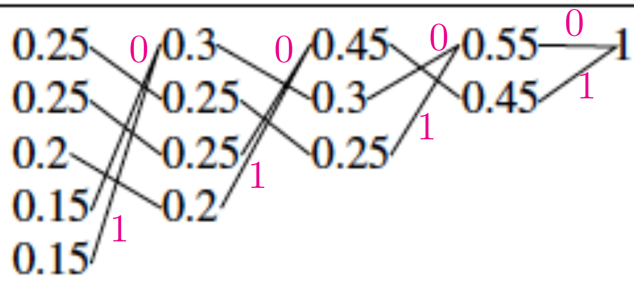
We start to construct a tree, FROM THE BOTTOM!

Codeword Length	Codeword	X	Probability				
2	01	1	0.25	0.3	0.45	0.55	1
2	10	2	0.25	0.25	0.3	0.45	
2	11	3	0.2	0.25	0.25		
3	000	4	0.15	0.2			
3	001	5	0.15				

This code has average length 2.3 bits.

Note that I
have different
alternatives!

Codeword Length	Codeword	X	Probability
2	01	1	0.25
2	10	2	0.25
2	11	3	0.2
3	000	4	0.15
3	001	5	0.15



This code has average length 2.3 bits.

We build this tree from the bottom!!

1
0.55
0.45
0.3
0.25
0.2
0.15

1 0 0.55 1 0 0.45 1 0 0.3 1 0

0.2 (3) 0.25 (2) 0.25 (1) (5) 0.15 (4) 0.15

EQUIVALENT ASSIGNATION OF THE BITS

Just another example; There are several possibilities!! (freedom in this choice)

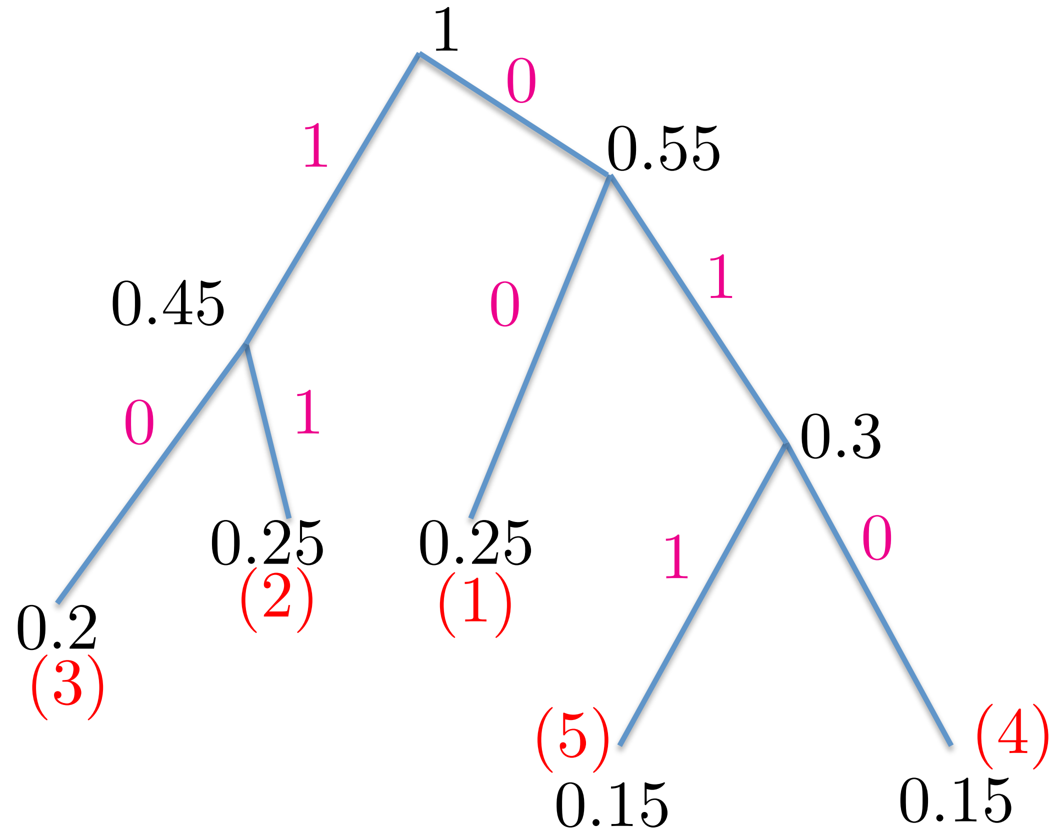
$$1 \Longrightarrow 00$$

$$2 \Longrightarrow 11$$

$$3 \Longrightarrow 10$$

$$4 \Longrightarrow 010$$

$$5 \Longrightarrow 011$$



Here we consider D=3, i.e., 0, 1 and 2. (just an example)

Example Consider a ternary code for the same random variable. Now we combine the three least likely symbols into one supersymbol and obtain the following table:

Codeword	X	Probability		
1	1	0.25	0.5	1
2	2	0.25	0.25	
00	3	0.2	0.25	
01	4	0.15		
02	5	0.15		

This code has an average length of 1.5 ternary digits.

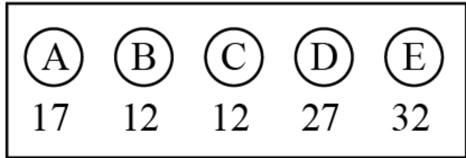
Other examples of Huffman coding (just with frequencies)

Table Frequency of characters

Character	A	B	C	D	E
Frequency	17	12	12	27	32

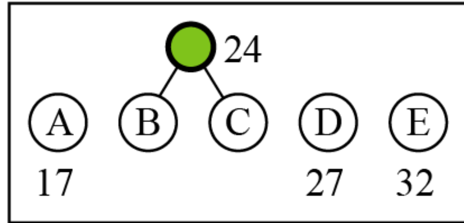
We build this tree from the bottom!!

Iteration -1



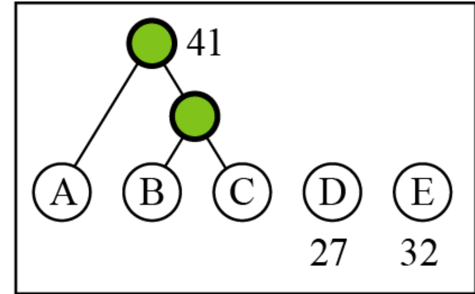
a.

Iteration -2



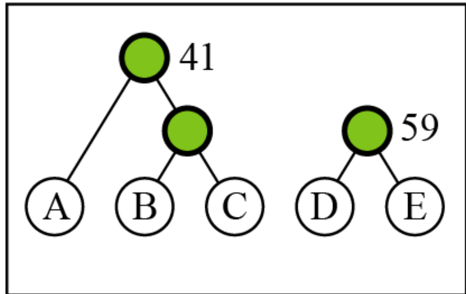
b.

Iteration -3

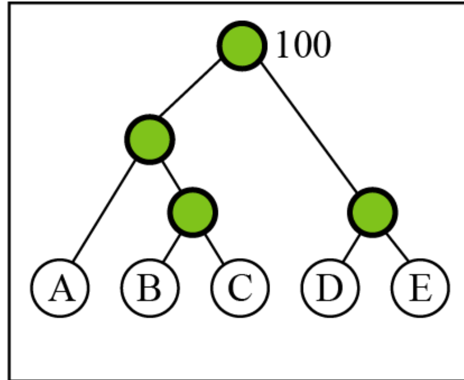


c.

Iteration -4



d.

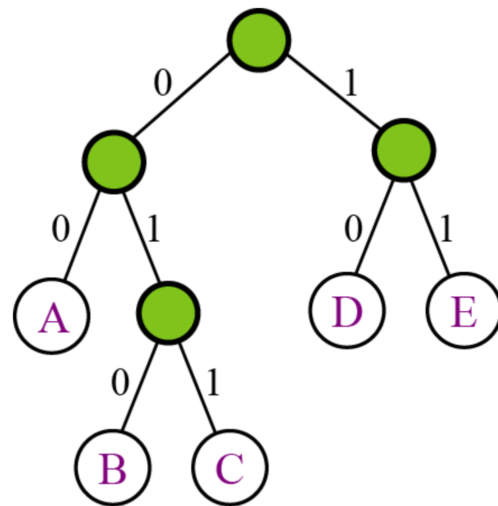


e.

Iteration -5

Huffman coding

The code itself is the bit value of each branch on the path, taken in sequence.



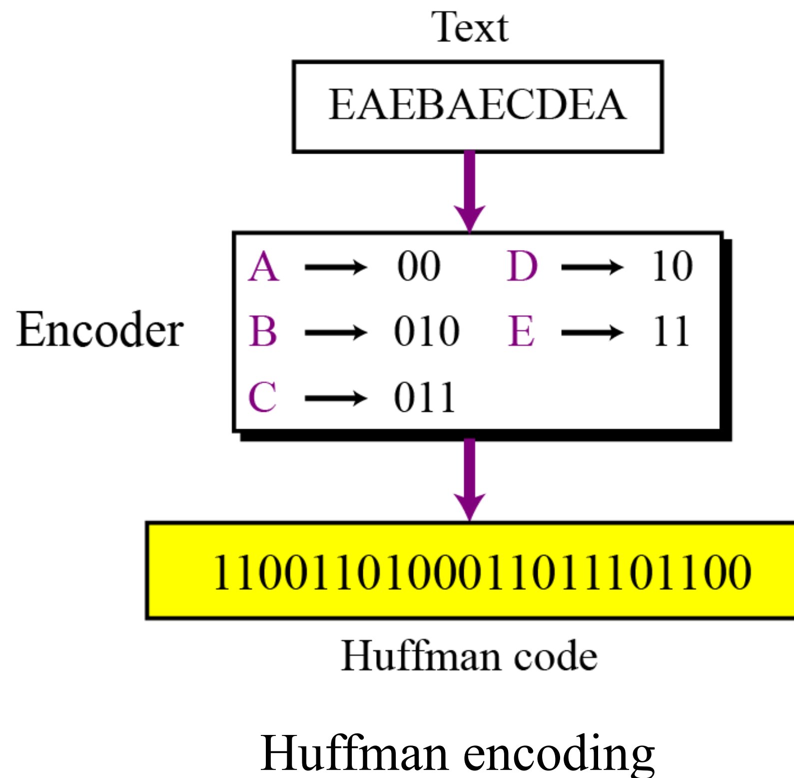
A: 00 D: 10
B: 010 E: 11
C: 011

Code

Final tree and code

Encoding

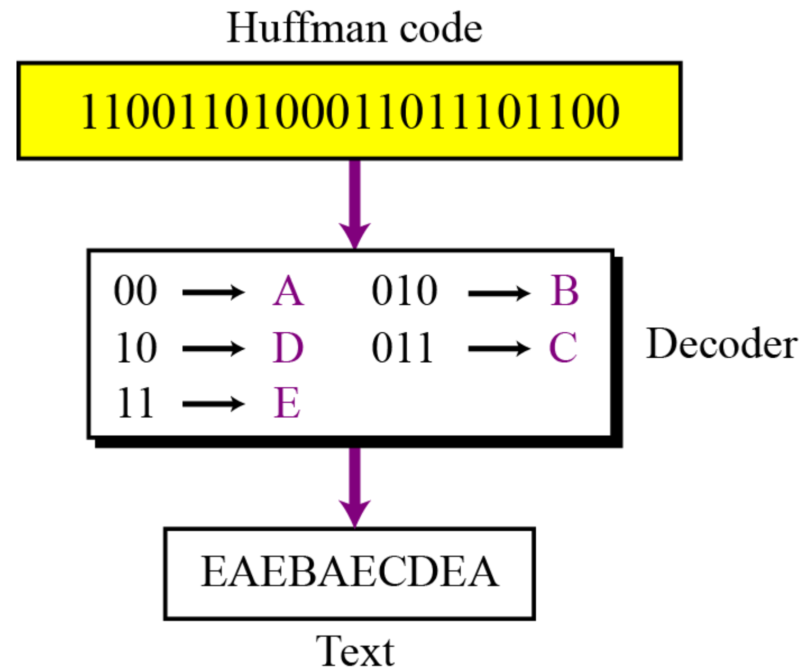
Let us see how to encode text using the code for our five characters. Figure 15.6 shows the original and the encoded text.



Huffman encoding

Decoding

The recipient has a very easy job in decoding the data it receives. Figure 15.7 shows how decoding takes place.



Huffman decoding

Other examples of Huffman coding (just with frequencies)

Consider the word “Hello”; the frequencies are:

$$H \longrightarrow 1$$

$$e \longrightarrow 1$$

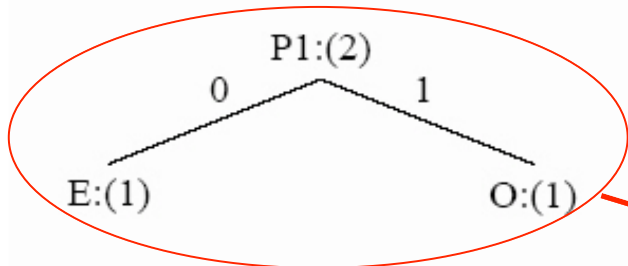
$$l \longrightarrow 2$$

$$0 \longrightarrow 1$$

Symbol	H	E	L	O
Count	1	1	2	1

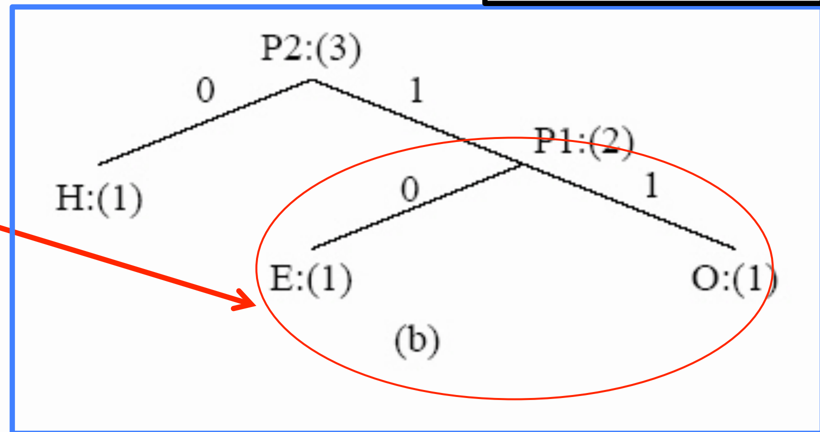
Bottom-up approach!!

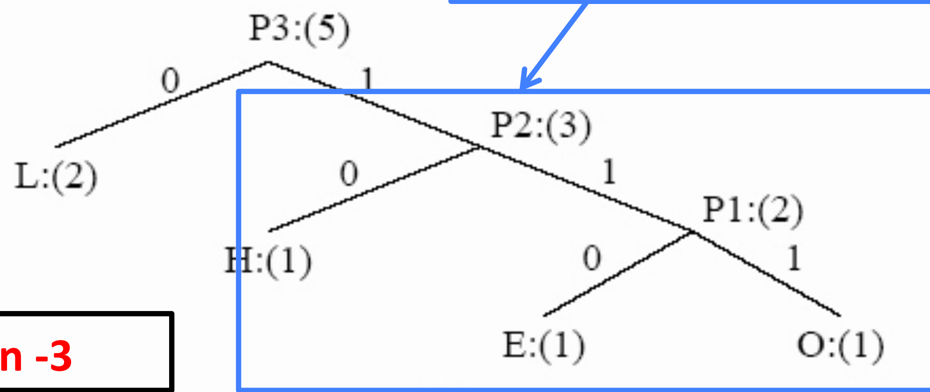
Example: compressing the word “Hello” con Huffman



Iteration -1

Iteration -2





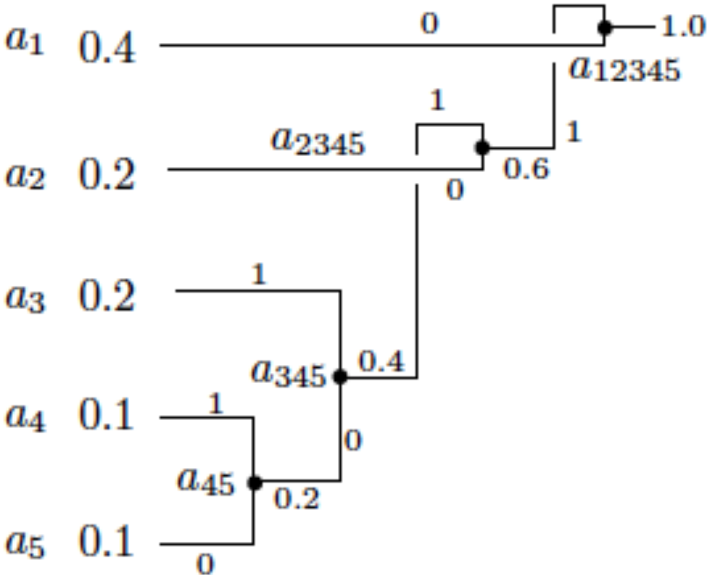
Iteration -3

VARIANCE of the code

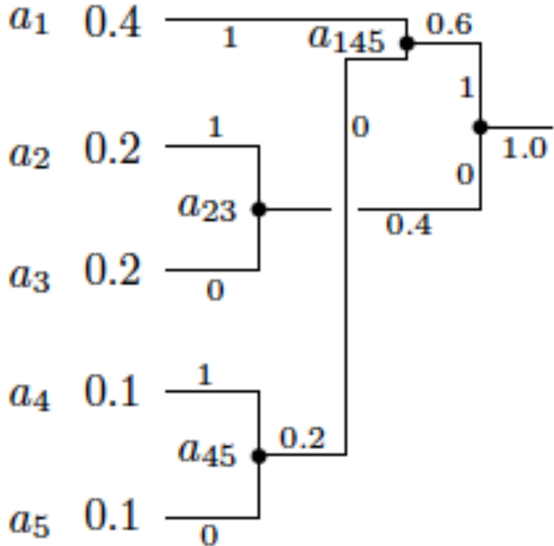
$$0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$$

They have the same expected length!

$$0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2 \text{ bits/symbol}$$



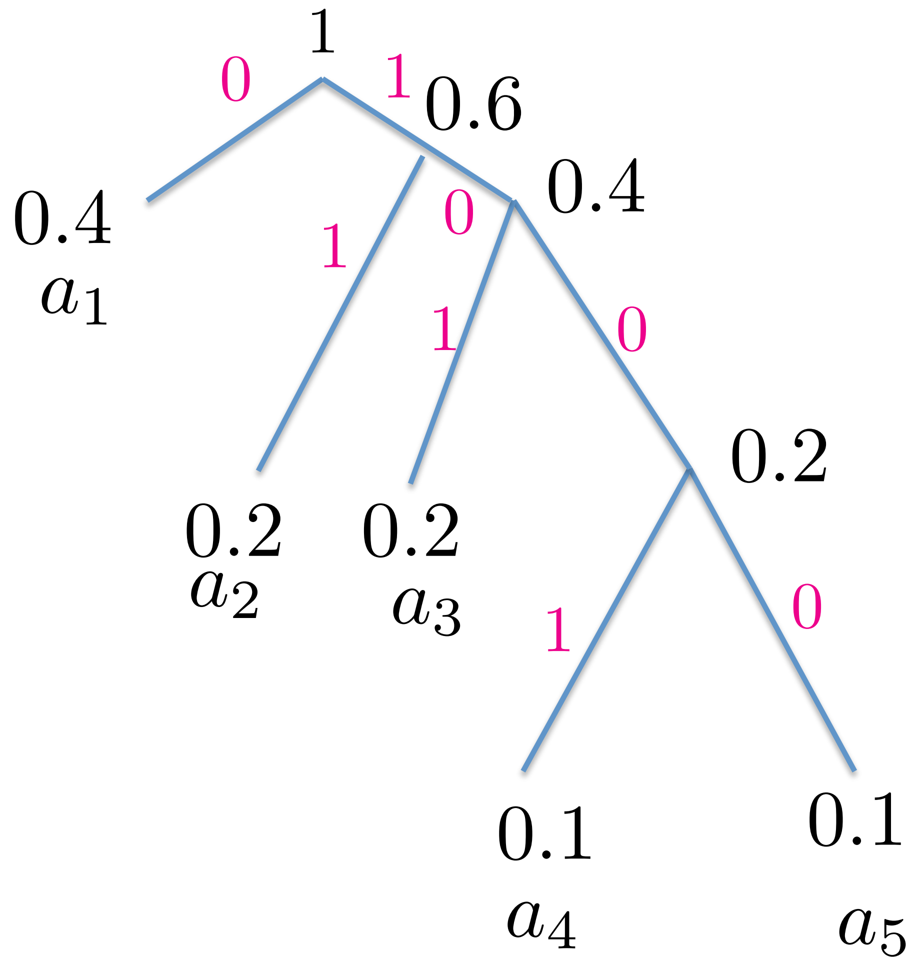
(a)



(b)

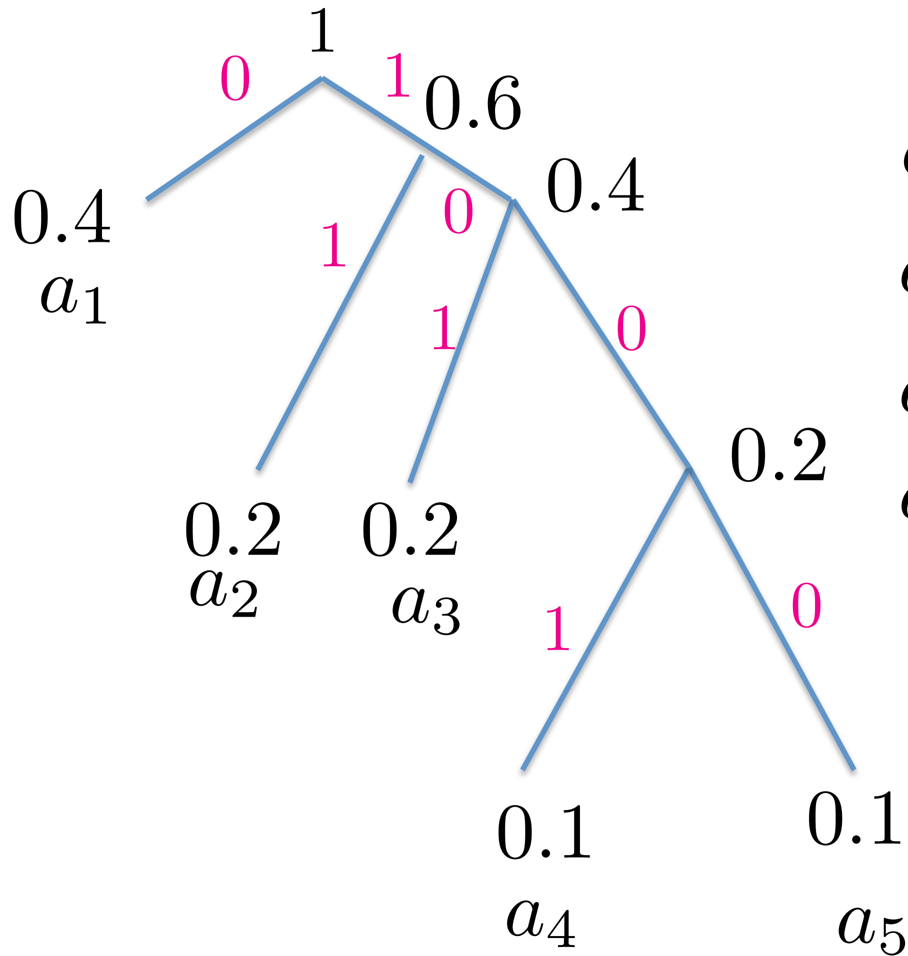
Figure 2.1: Huffman Codes.

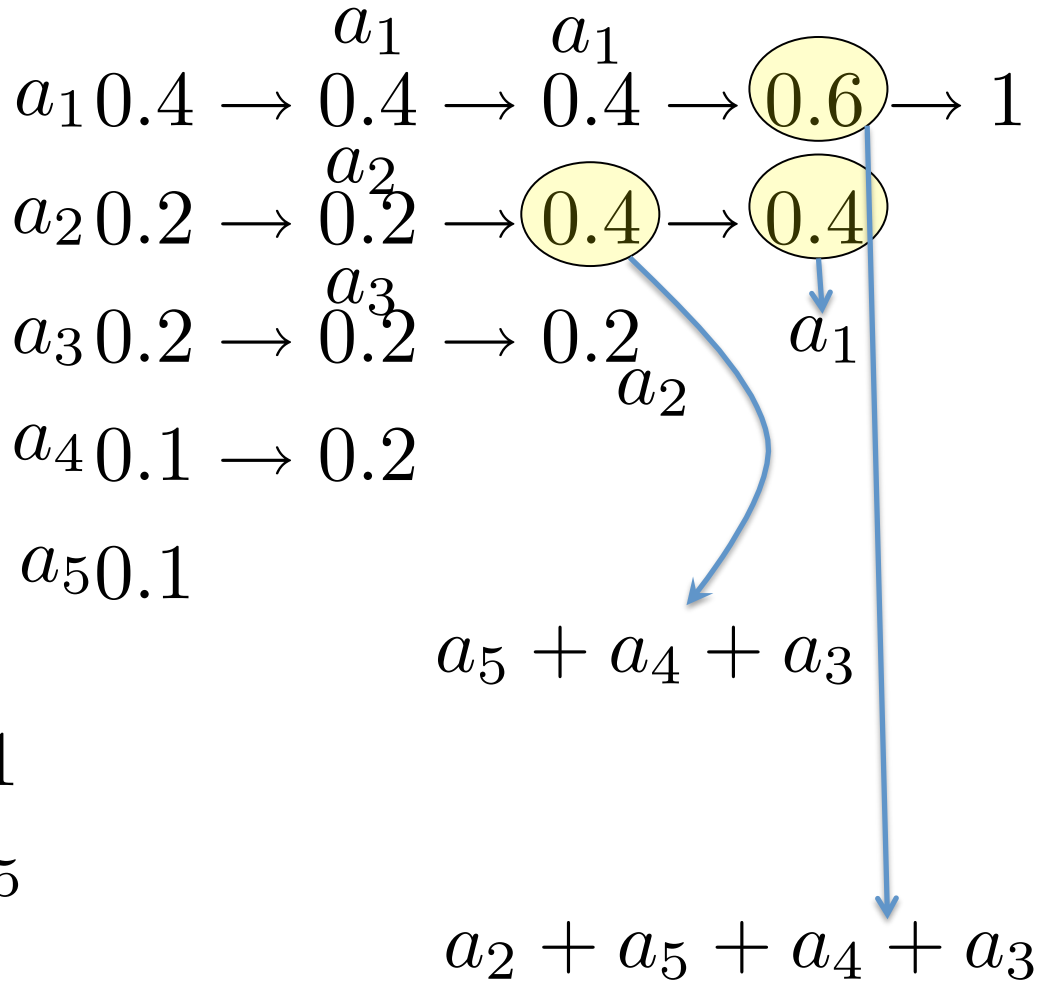
VARIANCE of the code



$$a_1 \longrightarrow 0$$
$$a_2 \longrightarrow 11$$
$$a_3 \longrightarrow 101$$
$$a_4 \longrightarrow 1001$$
$$a_5 \longrightarrow 1000$$

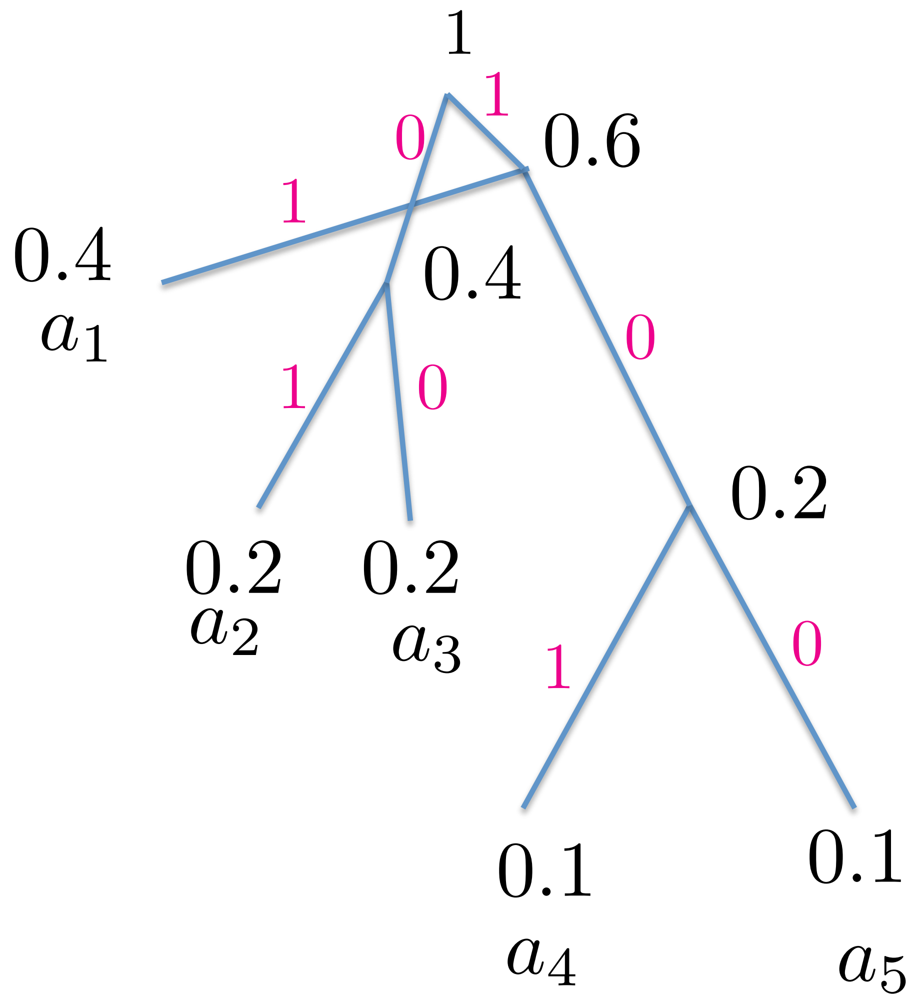
VARIANCE of the code





The depth of the tree is 4

VARIANCE of the code



$a_1 \longrightarrow 11$

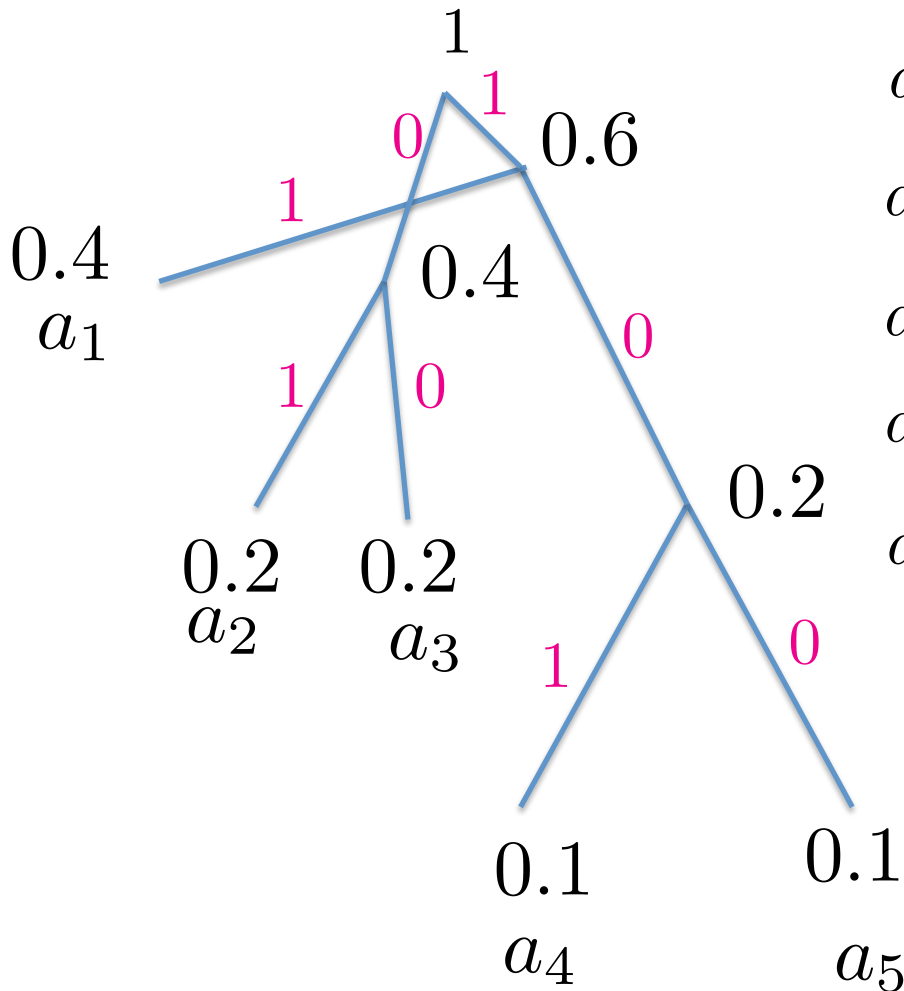
$a_2 \longrightarrow 01$

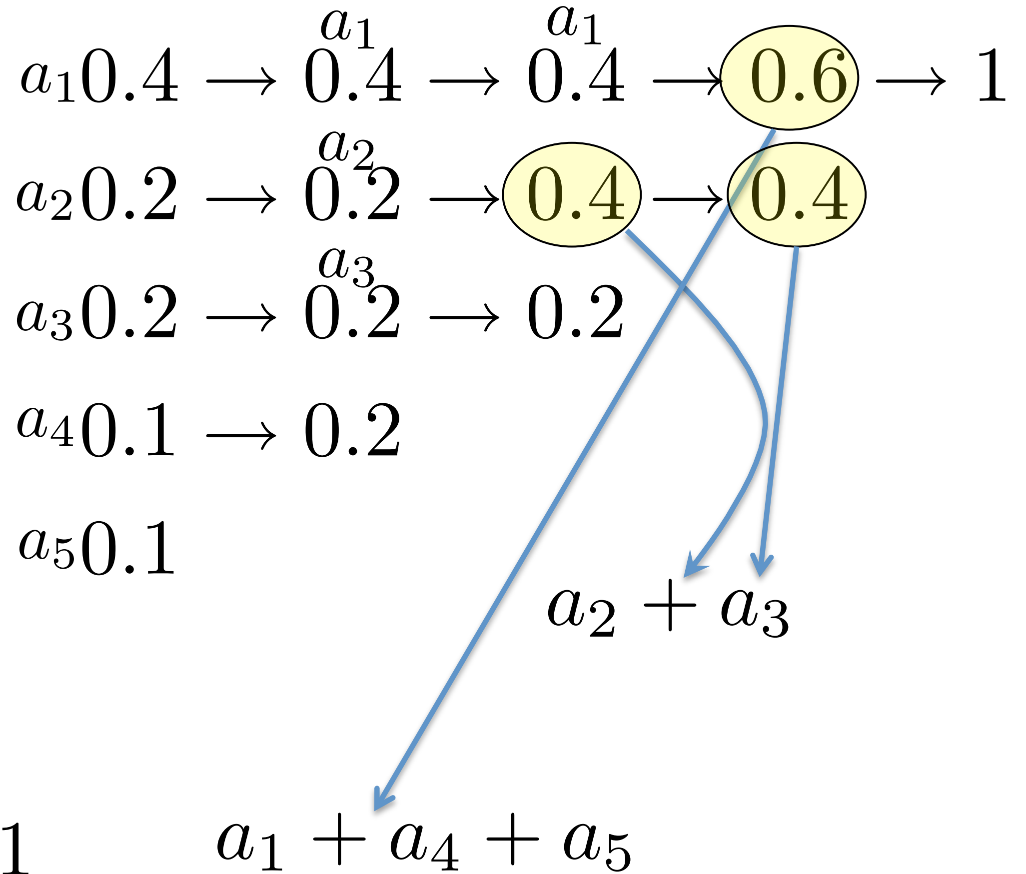
$a_3 \longrightarrow 00$

$a_4 \longrightarrow 101$

$a_5 \longrightarrow 100$

VARIANCE of the code

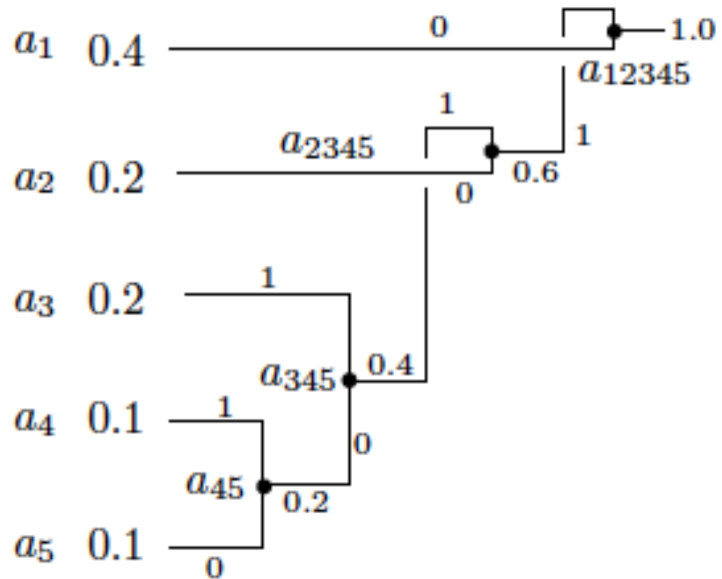




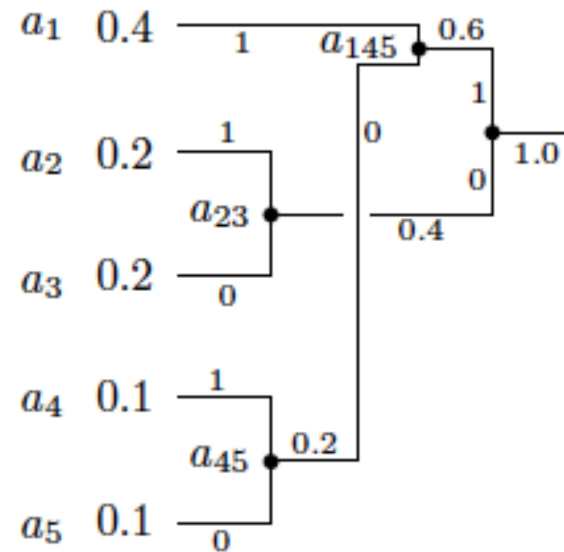
The depth of the tree is 3! It is less depth !

VARIANCE of the code

This is better!!! Since it has less variance



(a)



(b)

Figure 2.1: Huffman Codes.

$$0.4(1-2.2)^2 + 0.2(2-2.2)^2 + 0.2(3-2.2)^2 + 0.1(4-2.2)^2 + 0.1(4-2.2)^2 = 1.36,$$

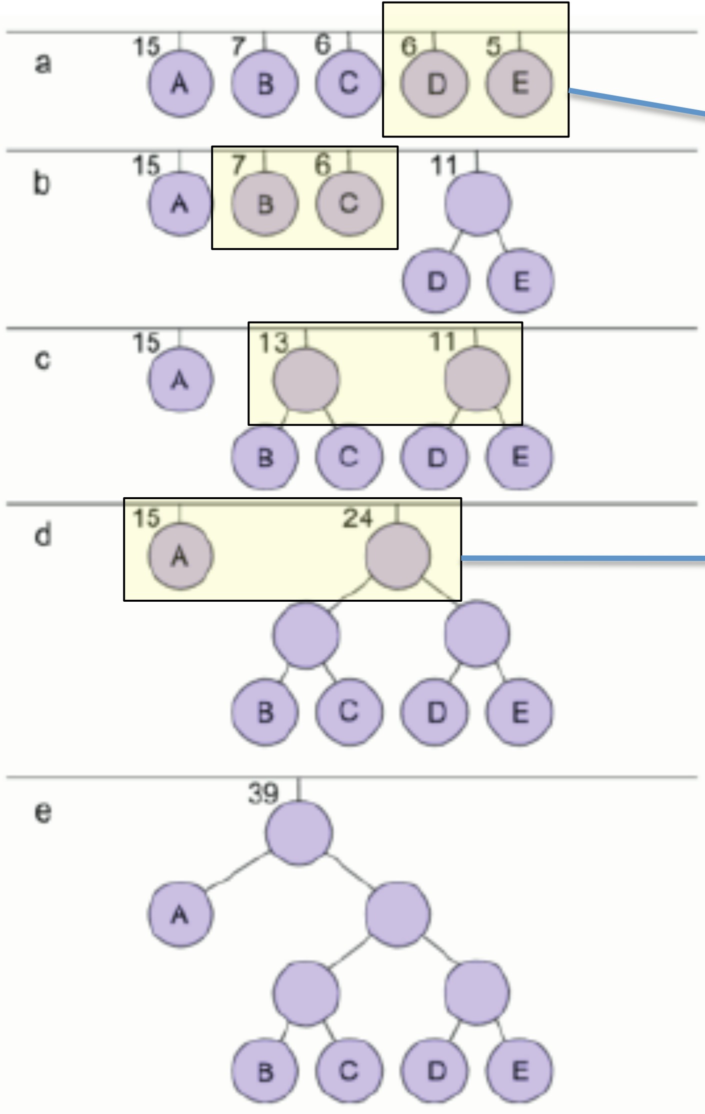
$$0.4(2-2.2)^2 + 0.2(2-2.2)^2 + 0.2(2-2.2)^2 + 0.1(3-2.2)^2 + 0.1(3-2.2)^2 = 0.16$$

Other example of Huffman coding

Symbol	A	B	C	D	E
Count	15	7	6	6	5
Probabilities	0.38461538	0.17948718	0.15384615	0.15384615	0.12820513

15+7+6+6+5=39

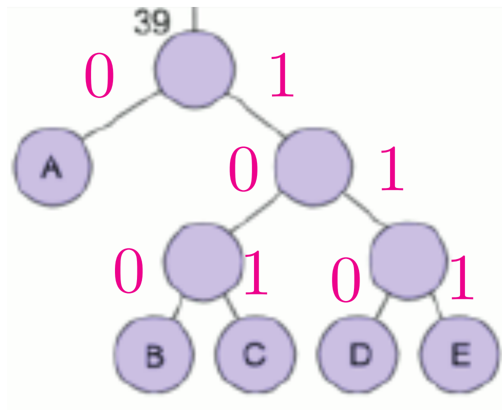
Other example of Huffman coding



The two smallest ones…

Just two…

Other example of Huffman coding

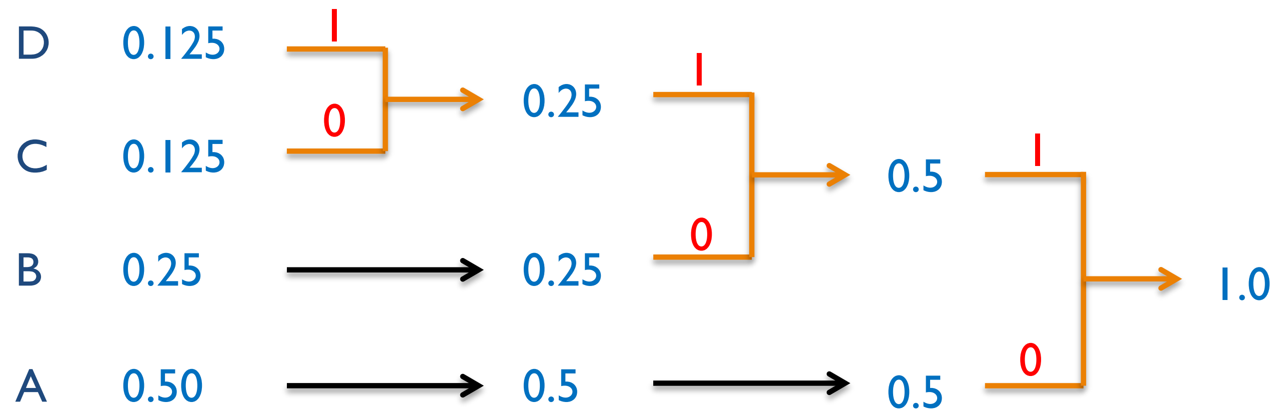


Symbol	A	B	C	D	E
Code	0	100	101	110	111

$$\frac{1\,\text{bit} \cdot 15 + 3\,\text{bits} \cdot (7 + 6 + 6 + 5)}{39\,\text{symbols}} \approx 2.23\,\text{bits per symbol.}$$

Other example of Huffman coding

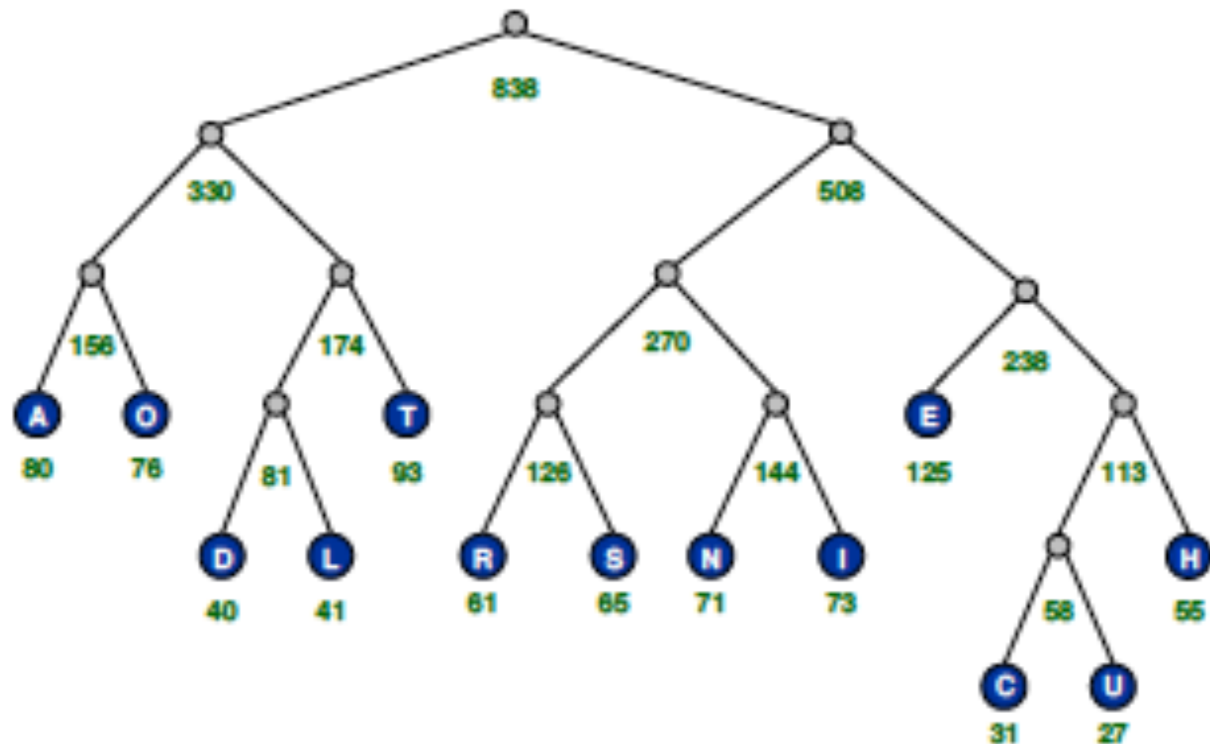
Símbolo (Si)	Pi	Código	Longitud (Li)
A	0.5	0	1
B	0.25	10	2
C	0.125	110	3
D	0.125	111	3



- Los códigos se asignan desde el nodo final a los nodos inicial de cada símbolo, (derecha a izquierda)

Huffman Coding Example

Char	Freq	Huff
E	125	110
T	93	000
A	80	000
O	76	011
I	73	1011
N	71	1010
S	65	1001
R	61	1000
H	55	1111
L	41	0101
D	40	0100
C	31	11100
U	27	11101
Total	838	3.62



SHANNON CODING
(it is not optimal)

- The idea is to create codewords of length:

$$\lceil -\log_2 p_i \rceil = \left\lceil \log_2 \frac{1}{p_i} \right\rceil$$

- And then assign the bits in order to construct an instantaneous code (a tree with codewords as leaves).

Huffman codes and Shannon codes. Using codeword lengths of $\lceil \log \frac{1}{p_i} \rceil$ (which is called *Shannon coding*) may be much worse than the optimal code for some particular symbol. For example, consider two symbols, one of which occurs with probability 0.9999 and the other with probability 0.0001. Then using codeword lengths of $\lceil \log \frac{1}{p_i} \rceil$ gives codeword lengths of 1 bit and 14 bits, respectively. The optimal codeword length is obviously 1 bit for both symbols. Hence, the codeword for the infrequent symbol is much longer in the Shannon code than in the optimal code.

Is it true that the codeword lengths for an optimal code are always less than $\lceil \log \frac{1}{p_i} \rceil$? The following example illustrates that this is not always true.

Comparison of SHANNON and Huffman coding

Example Consider a random variable X with a distribution $(\frac{1}{3}, \frac{1}{3}, \frac{1}{4}, \frac{1}{12})$. The Huffman coding procedure results in codeword lengths of (2, 2, 2, 2) or (1, 2, 3, 3)

Both these codes achieve the same expected codeword length. In the second code, the third symbol has length 3, which is greater than $\lceil \log \frac{1}{p_3} \rceil$. Thus, the codeword length for a Shannon code could be less than the codeword length of the corresponding symbol of an optimal (Huffman) code. This example also illustrates the fact that the set of codeword lengths for an optimal code is not unique (there may be more than one set of lengths with the same expected value).

Although either the Shannon code or the Huffman code can be shorter for individual symbols, the Huffman code is shorter on average. Also, the Shannon code and the Huffman code differ by less than 1 bit in expected codelength (since both lie between H and $H + 1$.)

FANO CODING
(sub-optimal procedure)
(a.k.a., Shannon-Fano coding)

The SHANNON-FANO algorithm

Fano codes. Fano proposed a suboptimal procedure for constructing a source code, which is similar to the idea of slice codes. In his method we first order the probabilities in decreasing order. Then we choose k such that $\left|\sum_{i=1}^{k} p_i - \sum_{i=k+1}^{m} p_i\right|$ is minimized. This point divides the source symbols into two sets of almost equal probability. Assign 0 for the first bit of the upper set and 1 for the lower set. Repeat this process for each subset. By this recursive procedure, we obtain a code for each source symbol. This scheme, although not optimal in general, achieves $L(C) \leq H(X) + 2$.

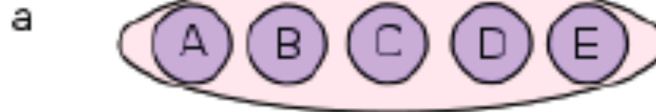
When he probability are sorted in descending order!!!

Example

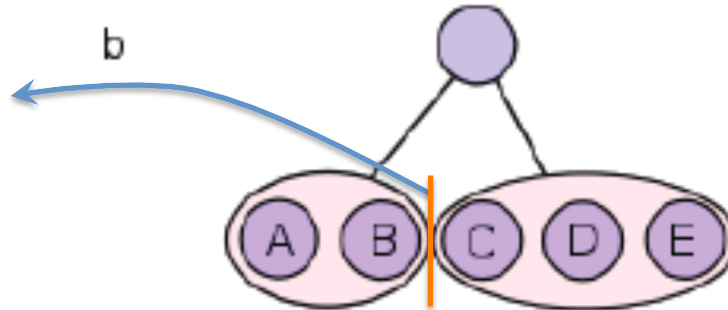
Symbol	A	B	C	D	E
Count	15	7	6	6	5
Probabilities	0.38461538	0.17948718	0.15384615	0.15384615	0.12820513

15+7+6+6+5=39

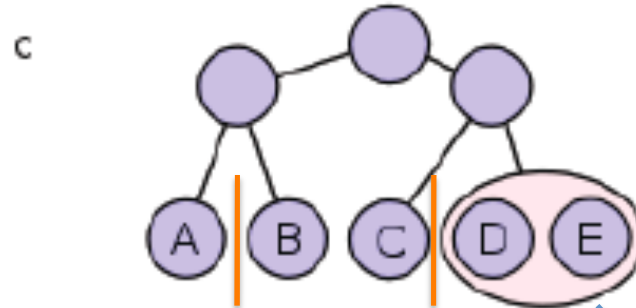
Example

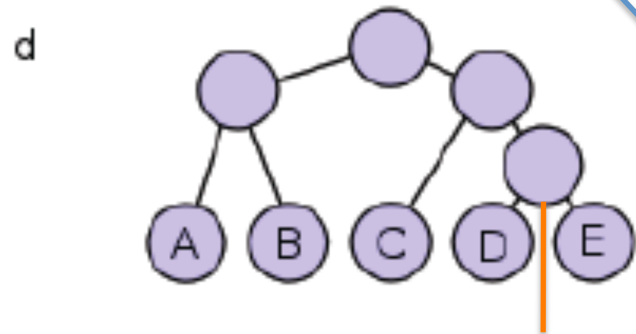


the sum is as close as possible....



It can be seen as a "TOP-BOTTOM", procedure





This was the last "super-symbol" to divide…

Example

Symbol	A	B	C	D	E
Code	00	01	10	110	111

$$\frac{2\,\text{bits} \cdot (15 + 7 + 6) + 3\,\text{bits} \cdot (6 + 5)}{39\,\text{symbols}} \approx 2.28\,\text{bits per symbol.}$$

Example

<table>
<tr><th>a_i</th><th>$p(a_i)$</th><th>1</th><th>2</th><th>3</th><th>4</th><th>Code</th></tr>
<tr><td>a_1</td><td>0.36</td><td rowspan="2">0</td><td colspan="3">00</td><td>00</td></tr>
<tr><td>a_2</td><td>0.18</td><td colspan="3">01</td><td>01</td></tr>
<tr><td>a_3</td><td>0.18</td><td rowspan="4">1</td><td colspan="3">10</td><td>10</td></tr>
<tr><td>a_4</td><td>0.12</td><td rowspan="3">11</td><td colspan="2">110</td><td>110</td></tr>
<tr><td>a_5</td><td>0.09</td><td rowspan="2">111</td><td>1110</td><td>1110</td></tr>
<tr><td>a_6</td><td>0.07</td><td>1111</td><td>1111</td></tr>
</table>

0.54
0.46

Other Example

Consider ***again*** the word “Hello”; the frequencies are:

$$H \longrightarrow 1$$

$$e \longrightarrow 1$$

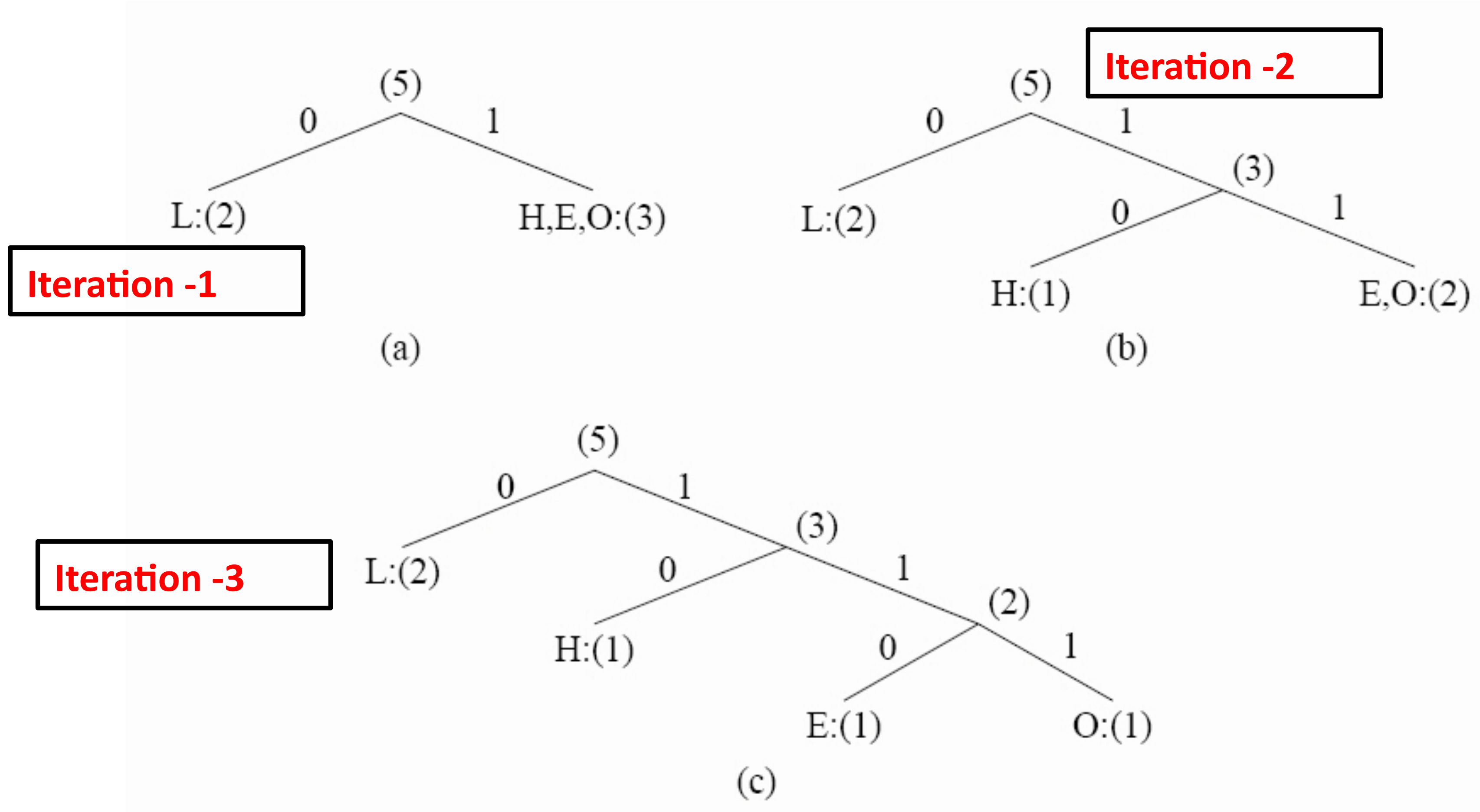
$$l \longrightarrow 2$$

$$0 \longrightarrow 1$$

Symbol	H	E	L	O
Count	1	1	2	1

TOP-DOWN APPROACH (opposite of Huffman…)

Example: compressing the word “Hello”



Shannon-Fano code on HELLO

Symbol	Count	$\text{Log}_2 \frac{1}{p_i}$	Code	# of bits used
L	2	1.32	0	2
H	1	2.32	10	2
E	1	2.32	110	3
O	1	2.32	111	3
			TOTAL # of bits:	10

ARITHMETIC CODING

Codificación Aritmética

- Si en lugar de asignar un código a un símbolo codificáramos un código a una secuencia de m símbolos, la longitud media por símbolo se aproximaría más a la entropía de la fuente en el rango:

$$H \leq L_h \leq H + \frac{1}{m}$$

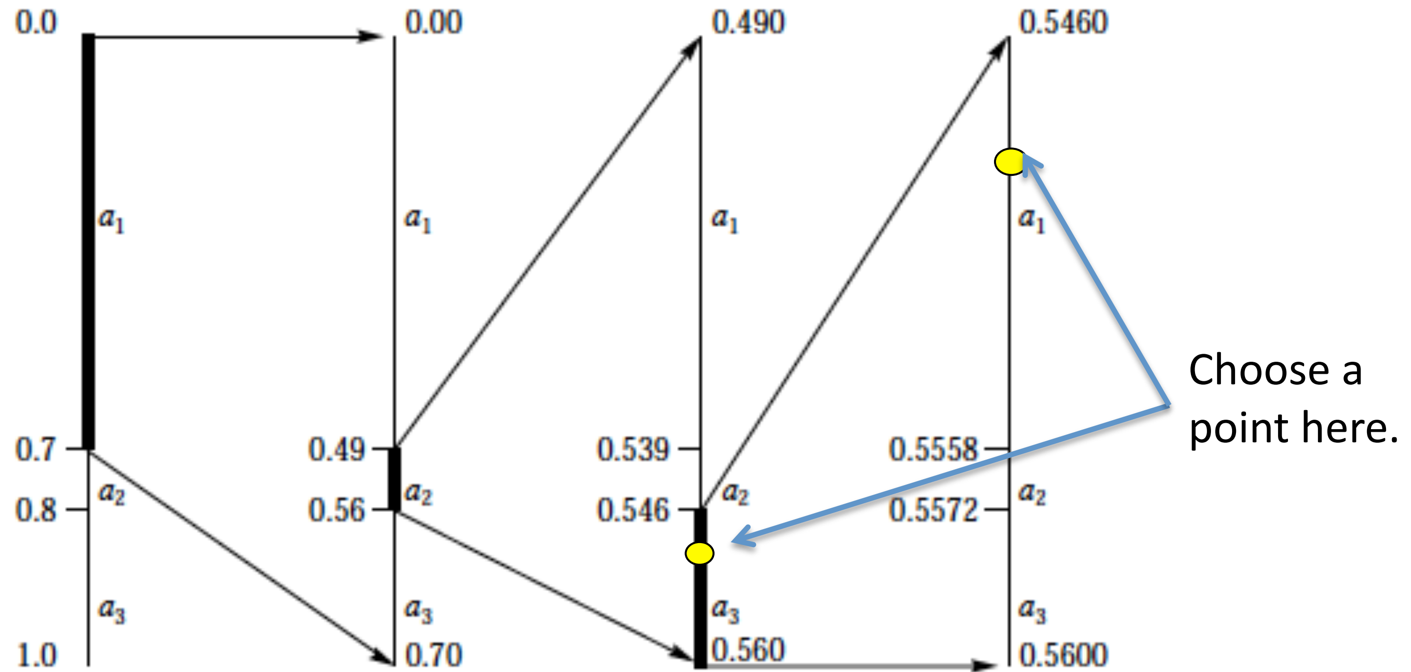
- La codificación aritmética se basa en la asignación de una representación binaria de la probabilidad acumulada de un conjunto de símbolos, cuya representación está en el intervalo [0,1).
- No hay una correspondencia directa entre símbolo y código.
- Se envía una cadena de bits que representa un punto en el intervalo de probabilidades obtenidas por la cadena de símbolos codificados.
- Obtiene mejor rendimiento que el codificador Huffman cuando las probabilidades de los símbolos no son potencias de 2, que es el caso más habitual.

Codificación Aritmética

- Interpretación conceptual:
  - Como la suma de probabilidades de un alfabeto de m símbolos ($s_1, s_2, .., s_{m-1}, s_m$) tiene que sumar 1, se divide el intervalo [0,1] en m subintervalo cuya longitud coincide con la probabilidad de cada símbolo ($p_1, p_2, .., p_{m-1}, p_m$).
  - Con la llegada del primer símbolo se selecciona el subintervalo de dicho símbolo.
  - Este subintervalo se vuelve a dividir en m subintervalos con longitudes proporcionales a las probabilidades p_i de cada símbolo.
  - Con la llegada del siguiente símbolo se selecciona el nuevo subintervalo, y se vuelve a dividir en m subintervalos con longitudes proporcionales a las probabilidades p_i de cada símbolo.
  - Se repite hasta que se envían los k códigos que se desean codificar de modo conjunto.
  - Se envía la representación binaria de un puno intermedio del subintervalo final
  - Es preciso enviar un símbolo especial de fin de cadena de símbolos.

Consider a three-letter alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with $P(a_1) = 0.7$, $P(a_2) = 0.1$, and $P(a_3) = 0.2$.

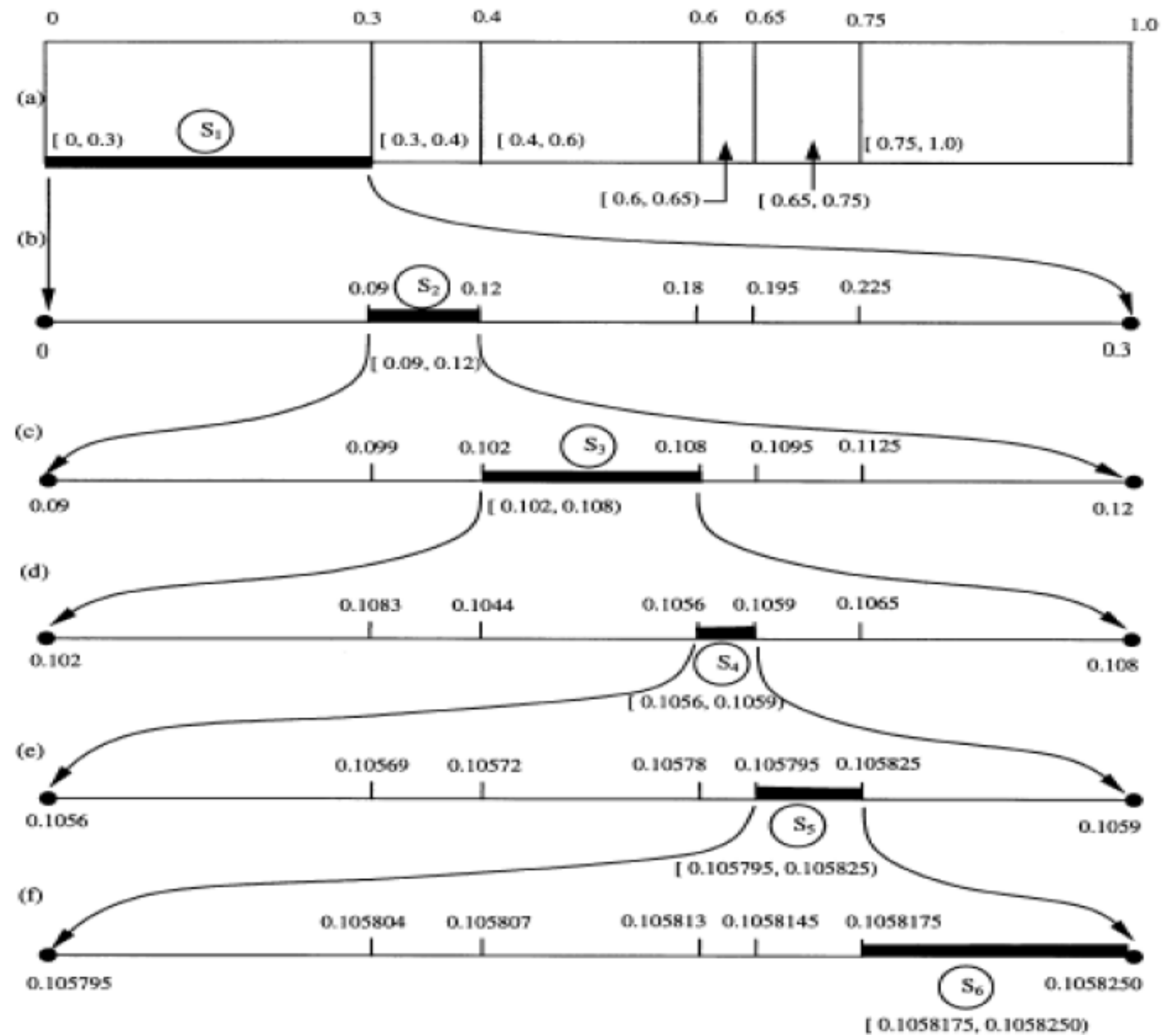
Consider to transmit the sequence = a1, a2, a3



$$0.7 \times 0.7 = 0.49$$
$$0.7 \times 0.8 = 0.56$$

$$(0.56 - 0.49) \times 0.7 + 0.49 = 0.539$$
$$(0.56 - 0.49) \times 0.8 + 0.49 = 0.546$$

Símbolo	Pi
s_1	0.3
s_2	0.1
s_3	0.2
s_4	0.05
s_5	0.1
s_6	0.25



- Se quiere transmitir una secuencia con los 6 símbolos en orden ($s_1, s_2, s_3, s_4, s_5, s_6$)
- El resultado es cualquier valor en binario en el intervalo [0.1058175, 0.1058250):
- [0.0001101100010110,

0.0001101100010111)

Codificación Aritmética

- ¿Cómo represento un número real no entero en formato binario?
  - El convenio es utilizar para la parte decimal potencias negativa de la base (2):
    - Ejemplo de representación de un número con 10 bits, 5 bits para la parte entera y 5 bits para la fraccionaria:

b_4	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	b_{-4}	b_{-5}
2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

    - Ejemplo de representación binaria del número decimal 19.34375: 10011.01011

b_4	b_3	b_2	b_1	b_0	b_{-1}	b_{-2}	b_{-3}	b_{-4}	b_{-5}
2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
1	0	0	1	1	0	1	0	1	1

OTHER EXAMPLE (for understanding how to DECODE)

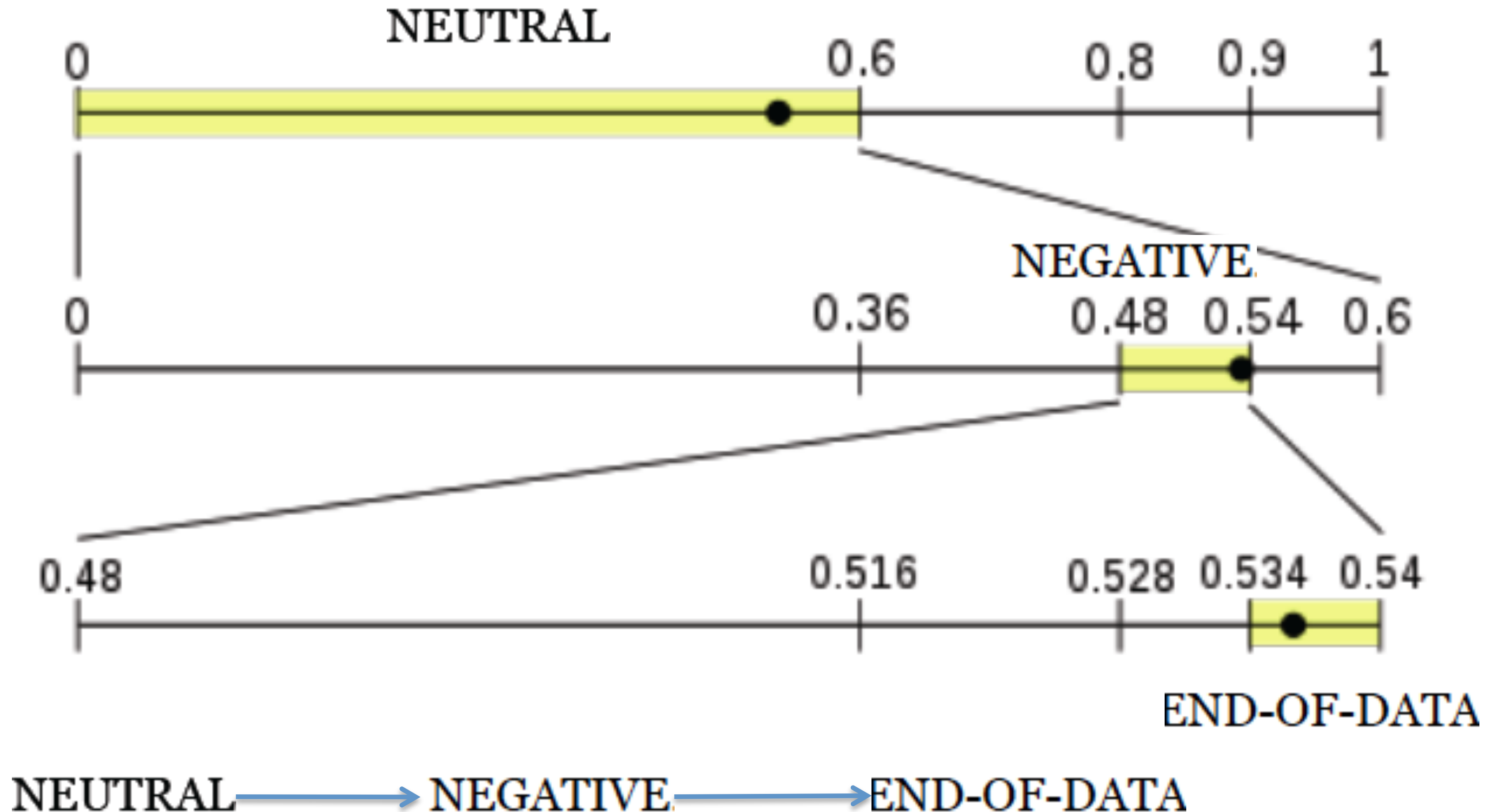
- 60% de probabilidad del símbolo NEUTRAL
- 20% de probabilidad del símbolo POSITIVE
- 10% de probabilidad del símbolo NEGATIVE
- 10% de probabilidad del símbolo END-OF-DATA (Fin de los datos). *(La presencia de este símbolo significa que la transmisión será 'terminada internamente', como es bastante común en compresión de datos; cuando este símbolo aparece en el flujo de datos, el decodificador sabrá que el flujo entero ha sido decodificado.)*

OTHER EXAMPLE (for understanding how to DECODE)

- el intervalo para NEUTRAL sería [0, 0.6)
- el intervalo para POSITIVE sería [0.6, 0.8)
- el intervalo para NEGATIVE sería [0.8, 0.9)
- el intervalo para END-OF-DATA sería [0.9, 1).

OTHER EXAMPLE (for understanding how to DECODE)

We have transmitted /received **0.538 (real number of base 2 number….)**



We need to know how many simbols we have, or the “end-of-data”