ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Multimedia Technology

Section # 6: Entropy Coding

Instructor: George Xylomenos

Department: Informatics

Contents

- Optimal Coding
- Shannon-Fano Coding
- Huffman Coding
- Adaptive Huffman Coding
- Arithmetic Coding
- Window-based Coding
- Dictionary-based Coding

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Optimal Coding

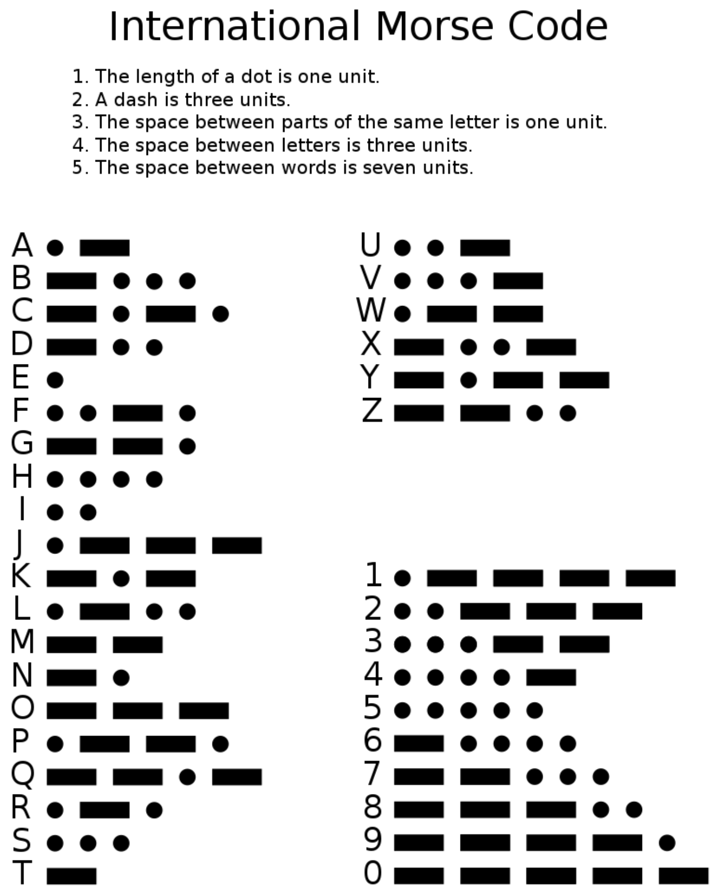
Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

Optimal coding (1 of 4)

- Fixed length coding
  - Example: ASCII (the original)
  - Each character encoded with 7 bits
- Variable length coding (VLC)
  - Example: Morse code
  - Three different code symbols (dot/dash/space)
  - More code symbols for rare characters
  - Spaces between codes

Optimal coding (2 of 4)



Optimal coding (3 of 4)

- Optimal entropy coding
  - As many bits as the information of the symbol
    - Average length = source entropy
  - What if information is not an integer?
    - Efficiency drops accordingly
- Uses only 0 and 1 (no spaces)
  - How do we know a code is finished
  - Unique prefix property
    - No code is the prefix of any other code

Optimal coding (4 of 4)

- Requires symbol probabilities
  - First read the file to find the probabilities
    - What if we do not have the entire file?
  - Assume a probability distribution
  - Gradually compute probabilities
- So, optimal under specific conditions!
  - Can we find construct such codes?
  - Yes – we will see multiple methods

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Shannon-Fano coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

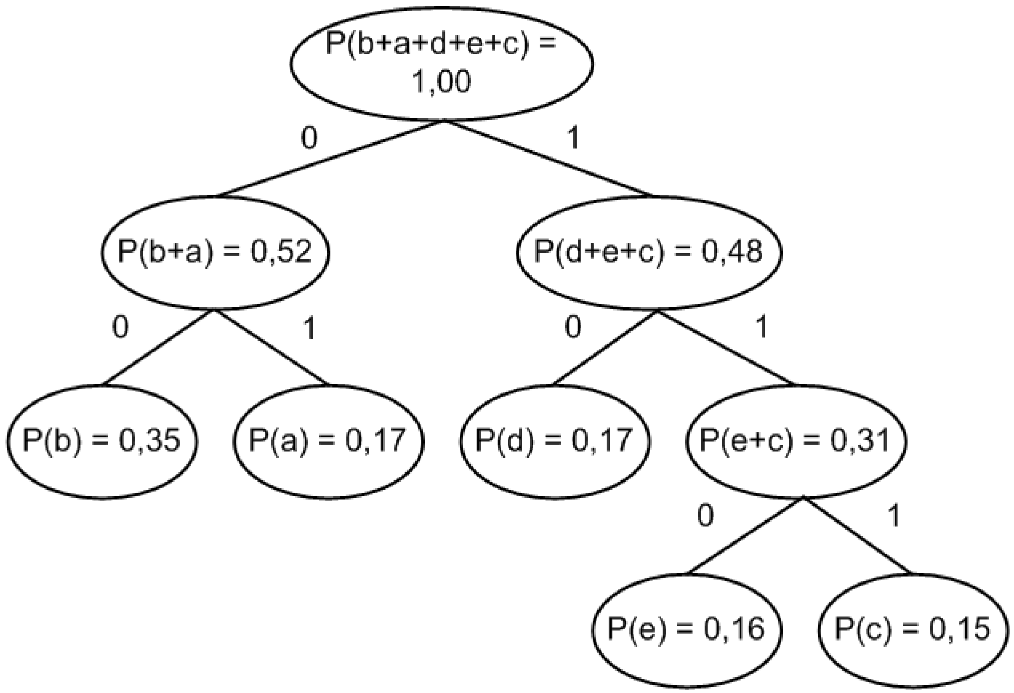
Shannon-Fano (1 of 6)

- Shannon-Fano coding
  - Uses codewords with integer length
    - Diverges from theoretically optimal
  - No code is the prefix of any other code
    - This is the key to VLCs
  - Binary coding tree
    - Leafs: symbols and probabilities
    - Nodes: symbol sets and probabilities
  - Exact same tree used for decoding

Shannon-Fano (2 of 6)

- Tree construction
  - We first sort all symbols by probability
    - Either increasing or decreasing order
  - Break symbols in left and right set
    - Each set has the sum of symbol probabilities
    - We want sets with as equal probabilities as possible
    - **Note: we never re-sort the symbols**
    - The two sets become children of a new node
    - Assign 0 to one child and 1 to the other
  - Repeat until we only have leaves left
    - Each leaf is a different symbol

Shannon-Fano (3 of 6)



- Example coding tree
  - P(a)=0,17, P(b)=0,35, P(c)=0,15, P(d)=0,17, P(e)=0,16
  - We start with the sorted sequence b, a, d, e, c
  - Average code length: 2,31

Shannon-Fano (4 of 6)

- Coding: replace symbol x with code w(x)
  - Each symbol x corresponds to a leaf
  - The labels along its path (from the root) are w(x)
- Decoding
  - You need to know the encoding tree
  - Match input against paths in the tree
    - Each prefix corresponds to a different path
    - We always know when to stop (at the leaf)
  - Start each decoding cycle from the root

Shannon-Fano (5 of 6)

- Code tree construction
  - Compute probabilities from file to encode
  - Use pre-existing trees
    - Basically, assume a specific probability distribution
- Decode tree construction
  - Transmit code tree
  - Transmit probabilities
    - And fix the code tree contsruction rules
  - Use pre-existing trees (send a tree ID if many exist)

Shannon-Fano (6 of 6)

- What happens if we have two options?
  - Example: set abc with P(a)=P(b)=P(c)=0.1
    - We can either split it ab – c or a – bc
    - Or we could have sorted it cba in the beginning
  - It actually makes NO difference!
    - Different trees but same average code length
  - But, we need to know what the rules are!
    - This allows the decoder to build the same tree

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Huffman coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

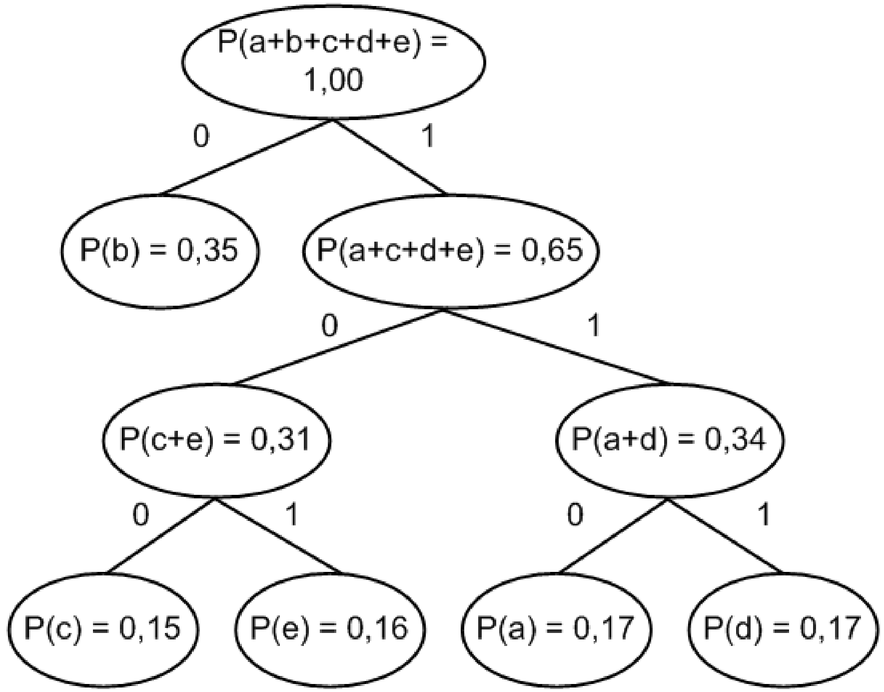
Huffman (1 of 3)

- Very similar to Shannon-Fano
  - Variable length codes per symbol
  - Need to know symbol probabilities
  - Binary coding/decoding tree
    - May differ from Shannon-Fano tree
  - Same coding/decoding algorithm
    - Only the tree differs!
  - Tree created bottom-up rather than top-down

Huffman (2 of 3)

- Tree construction
  - Each symbol becomes a leaf node
  - All nodes are added to a set
  - Select the two nodes with the lowest probabilities
  - Replace nodes in the set with a binary subtree
    - The parent has the sum of the probabilities
    - Assign 0 and 1 to the children
  - Stop when there is a single tree left
  - Slightly better trees than Shannon-Fano

Huffman (3 of 3)



- Example coding tree
  - P(a)=0,17, P(b)=0,35, P(c)=0,15, P(d)=0,17, P(e)=0,16
  - Average code length: 2,3 (better than Shannon-Fano)

Huffman vs Shannon-Fano (1 of 3)

- Huffman or Shannon-Fano?
  - Nearly identical schemes
  - Only the tree may be different
  - Same coding/decoding algorithm
- Shannon-Fano tree is easier to create
  - We do NOT sort symbols at each step
  - Easy way to find how to split set
    - Start adding probabilities from the left
    - When we have more than half, choose a split

Huffman vs Shannon-Fano (2 of 3)

- Huffman is more efficient
  - Shannon-Fano does not lead to optimal splits
    - By re-sorting the set, we can find a better one
  - Huffman partially re-sorts the set at every step
    - Always selects the two lowest probability nodes
  - Do we actually need a full sort?
    - In each step I select two nodes and add a new one
    - A binary heap can do this much faster

Huffman vs Shannon-Fano (3 of 3)

- Disadvantages of Huffman/Shannon-Fano
  - Need to know the symbol probabilities
  - Coding is not really optimal
    - Need an integer number of bits per symbol
    - Diverges from the ideal
  - Can we improve efficiency?
    - Why not code n symbols at each step?
    - This makes the tree huge (k^n for k initial symbols)

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Adaptive Huffman Coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

Adaptive Huffman (1 of 11)

- Adaptive Huffman coding
  - Does not need to know symbol probabilities
    - Tree is built as the input is processed
    - Automatically adapts to input probabilities
  - Start with an initial encoding
    - Could be simply the 8 bits in extended ASCII
    - The codes gradually change
      - Depending on symbol frequency

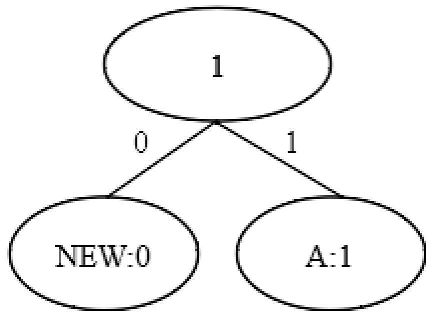
Adaptive Huffman (2 of 11)

- Adaptive Huffman coding
  - For every symbol we maintain a counter
    - Increased whenever it shows up in the input
    - All counters start at 0
  - The tree starts with the symbol NEW:0
    - NEW means that a new symbol has appeared
    - NEW is never an actual input (its counter is 0)
    - But, it has a code

Adaptive Huffman (3 of 11)

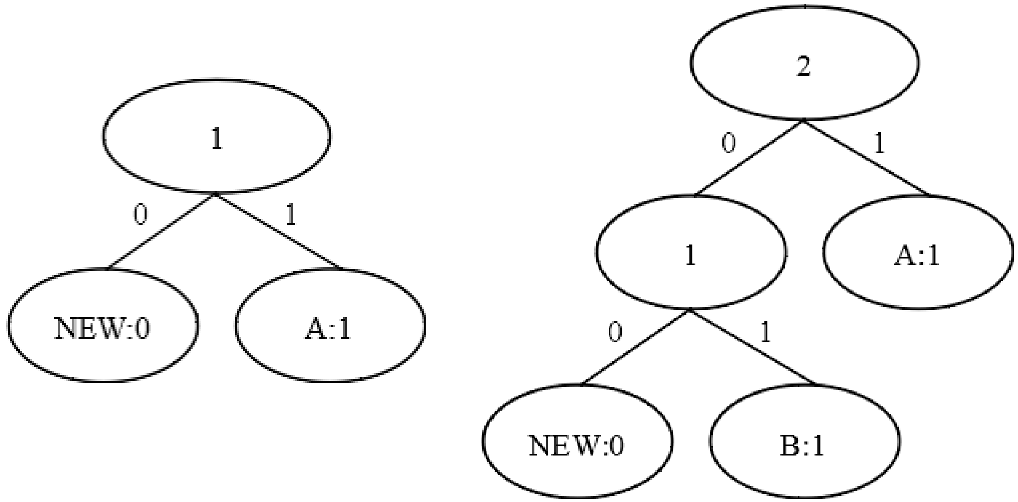
- Whenever we encounter a new symbol
  - Output the code for NEW
  - Then output the initial code for the symbol
  - Finally, add the new symbol to the tree
    - Split the NEW symbol at the bottom
  - Its counter is now 1
- Whenever we encounter an existing symbol
  - Output its current code
  - Increase its counter

Adaptive Huffman (4 of 11)



- Example: input is ABCCA
  - Initial codes: A=01, B=10, C=11
  - Initial tree: NEW:0 (code 0) and root
  - <u>A</u>BCCA: output 0 01 (NEW and A), add A:1 to tree

Adaptive Huffman (5 of 11)

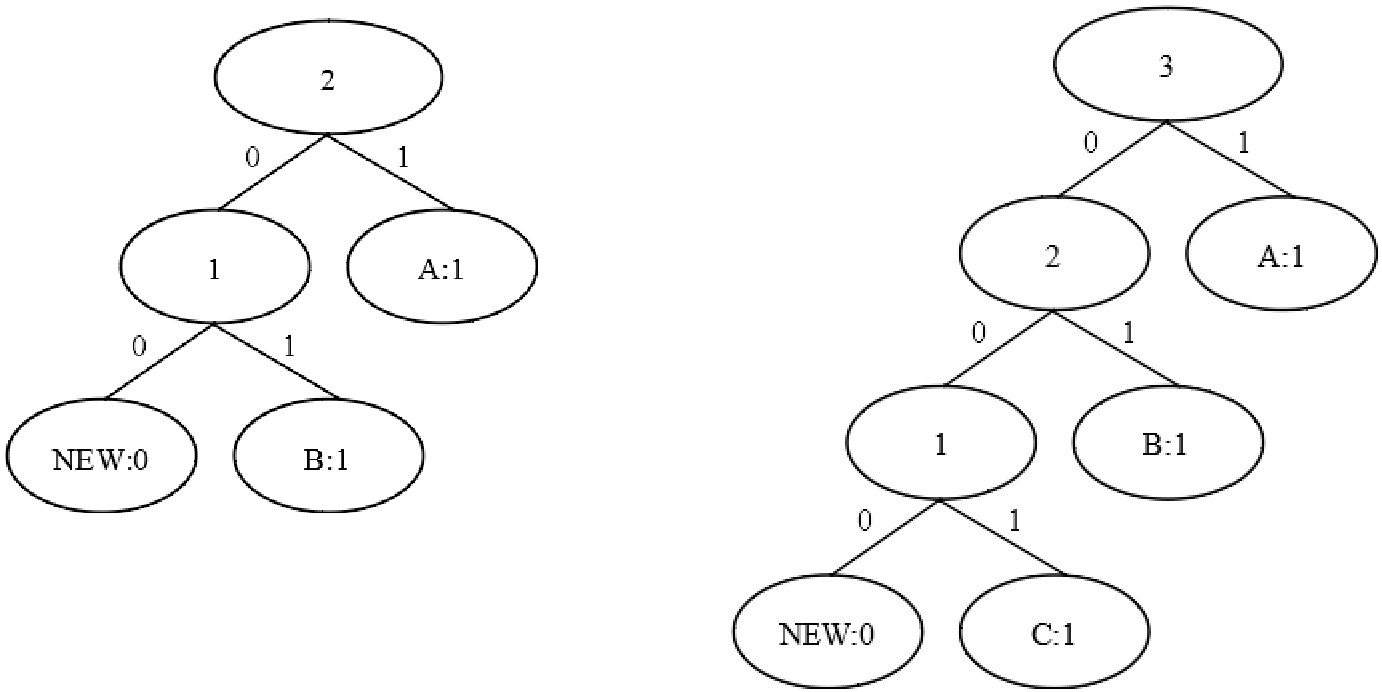


- Example: input is ABCCA
  - Internal nodes hold the sum of their child counters
  - A<u>B</u>CCA: output 0 10 (NEW and B), add B to tree
  - Update counters at internal nodes

Adaptive Huffman (6 of 11)

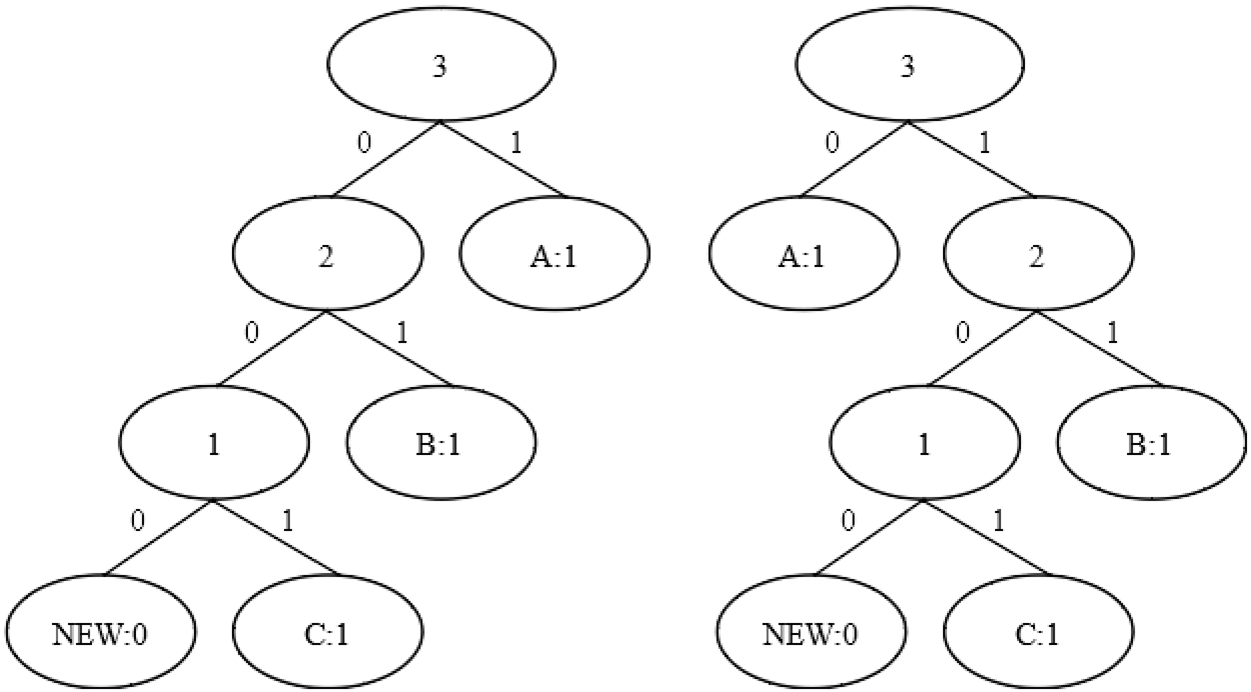
- The tree must always be sorted
  - According to the counters
    - Bottom to top in each path, left to right in each level
  - Say that a counter changed from N to N+1
    - If the node is not in the right position anymore
    - Find the furthest node with a counter of N
    - Swap the two nodes (or subtrees)
    - Repeat until the tree is sorted
  - The decoder does the exact same job

Adaptive Huffman (7 of 11)



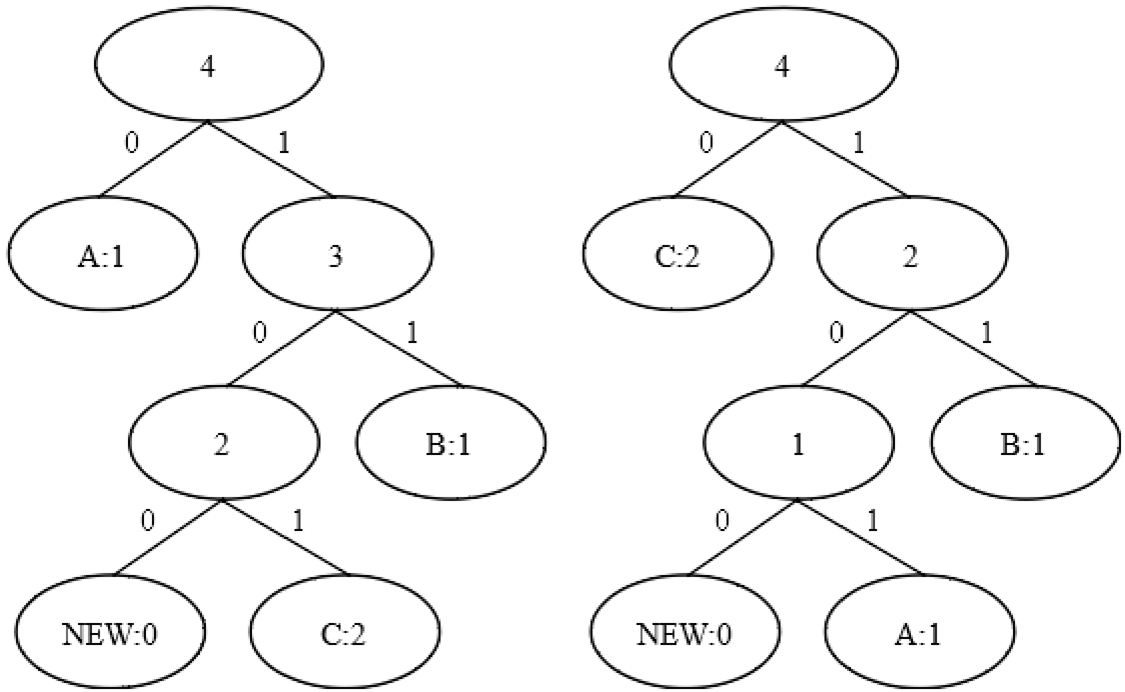
– AB<u>C</u>CA: output 00 11 (NEW and C), add C to tree
  - NEW got a longer code in the previous step
– Now, the first level has the wrong order

Adaptive Huffman (8 of 11)



– We need to move A to a different spot
– Swap A with the furthest node with a counter of 2
    - We basically swap the children of the root

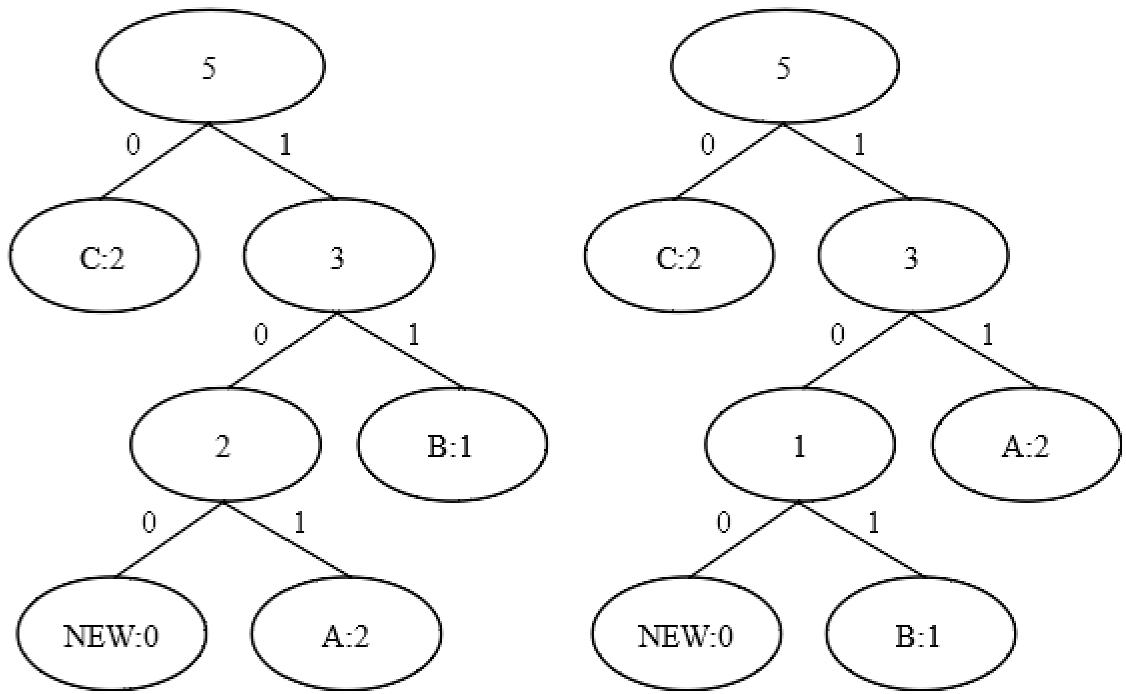
Adaptive Huffman (9 of 11)



– ABCCA: output 101 (just C, increase its counter)

– Now C is swapped with A (the furthest node with 1)

Adaptive Huffman (10 of 11)



– ABCCA: output 101 (just A, increase counter)

– Now A is swapped with B

Adaptive Huffman (11 of 11)

- Encoder and decoder always in sync
  - We first export the code
  - And then change the tree
    - The decoder sees the code and follows
    - It knows which symbol changed its counter
  - Output NEW before new codes
    - Followed by initial code, to notify decoder of new node
  - Note: NEW also changes its code

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Arithmetic Coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

Arithmetic (1 of 10)

- Codes input into a single fractional number
  - Fraction length depends on input length
    - May be <u>very</u> long!
  - No need for fixed bits/symbol
    - Avoids divergence from the optimal
  - Needs to know symbol probabilities
  - Needs a terminal symbol in the end
    - This is used to end decoding

Arithmetic (2 of 10)

- Preparatory stage
  - Sort all symbols (usually, alphabetically)
  - Symbol x_i assigned the interval $[a_i, b_i)$
    - The interval satisfies $b_i - a_i = p(x_i)$
  - Example
    - P(a) = 0.4, P(b) = 0.3, P(c) = 0.2 and P($) = 0.1 (terminal)
    - Interval a: [0, 0.4), Interval b: [0.4,0.7)
    - Interval c: [0.7, 0.9), Interval $: [0.9,1.0)

Arithmetic (3 of 10)

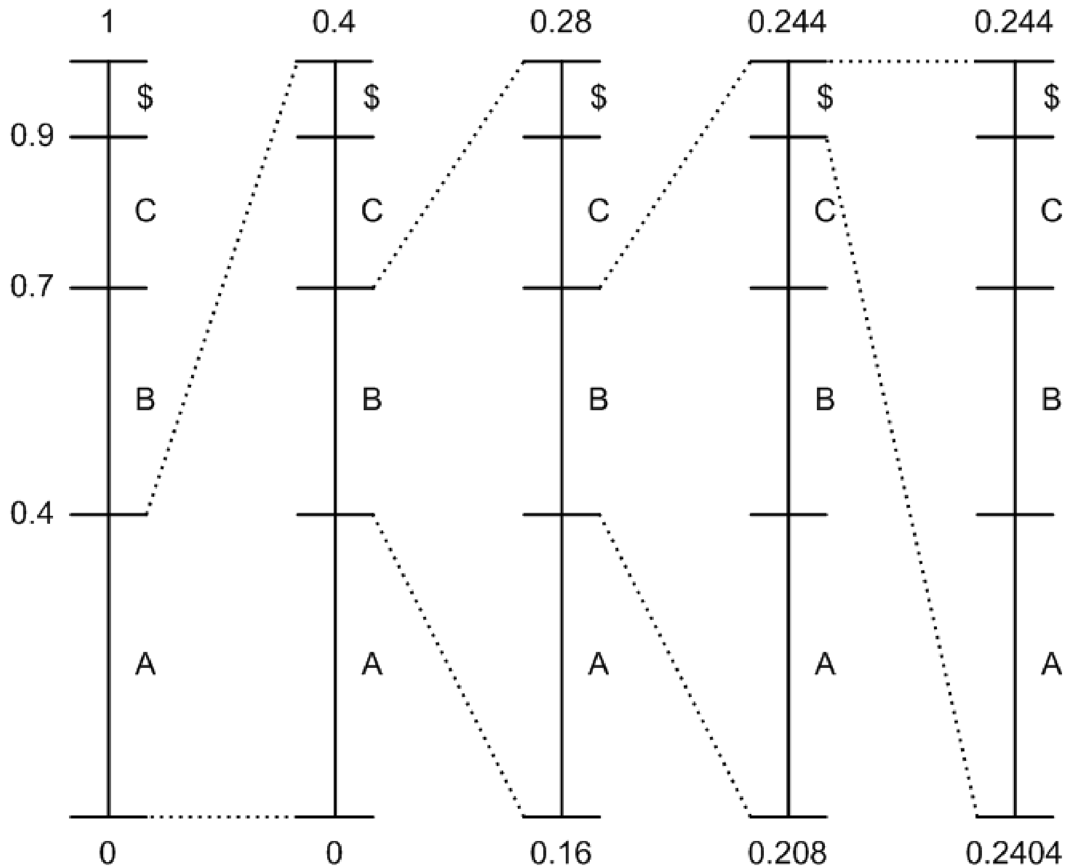
- Coding algorithm

```
low = 0.0;
high = 1.0;
repeat {
   input s;
   range = high - low;
   high = low + range * highrange[s];
   low = low + range * lowrange[s];
   } until s = $;
output any number in [low, high);
```

Arithmetic (4 of 10)

- What does the encoder do?
  – Lowrange[s]: low end of interval for s
  – Highrange[s]: high end of interval for s
  – The input is encoded as a <u>interval</u>
    - The interval is initialized to [0,1)
    - In every step, the interval shrinks
    - Depending on the input symbol
    - The longer the input, the smaller the interval

Arithmetic (5 of 10)



- Arithmetic coding example

Arithmetic (6 of 10)

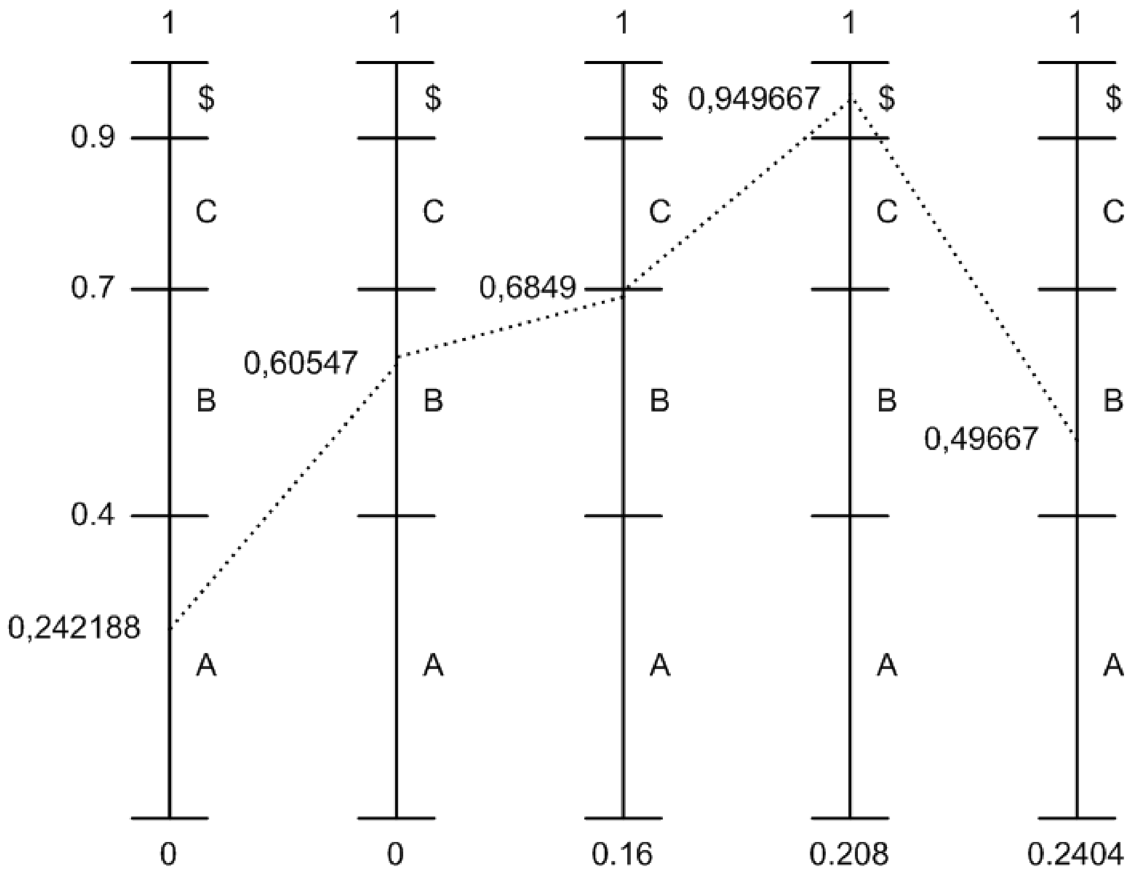
- Output calculation
  - We need a number within the interval
  - But, with the shortest fractional part
  - We start with 0. and add bits
  - Concatenate a 1 at the right end
    - If the fraction is over the high end, switch to 0
    - Repeat until the fraction is within the interval
  - No need to send the initial 0.

Arithmetic (7 of 10)

- Decoding algorithm

```
input n;
repeat {
    find s so that n is in
   [lowrange[s], highrange[s]);
    output s;
    range = highrange[s] - lowrange[s];
    n = (n - lowrange[s]) / range;
} until s = $;
```

Arithmetic (8 of 10)



- Arithmetic decoding example

Arithmetic (9 of 10)

- Why do we need a terminal symbol?
  - Decoding produces a single number
    - Not intervals, like encoding
  - It is not clear when to stop
    - We could continue forever!
  - The terminal symbol is the stop sign
    - If we hit its interval, we are done
  - We do not need an actual symbol
    - We just need to assign it an interval

Arithmetic (10 of 10)

- Issues with arithmetic encoding
  - Fractional numbers with arbitrary bit length
    - Needs special libraries
- Block-based encoding
  - Break the input into fixed length blocks
  - Each blocks requires fewer bits
  - Small drop in efficiency
  - No need for terminal symbol

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Window-based coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

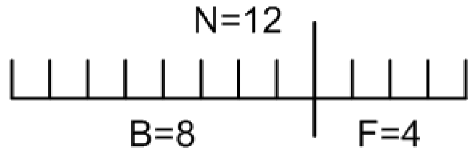
Why a window? (1 of 2)

- Limitations of entropy coding
  - VLC: needs handling of bit sequences
  - Arithmetic: needs very long numbers
  - And they cannot do better than the entropy!
- An alternative: code sequences of symbols
  - Ideally, variable length ones
  - Which sequences are common?
  - How can we represent them?

Why a window? (2 of 2)

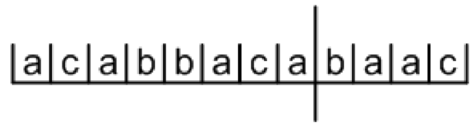
- At any given time the encoder
  - Has coded the input up to a point
  - Needs to code the input that follows
- Window-based coding
  - Looks for input prefixes…
  - …which have already been coded…
  - …so as to replace the prefix with a code

LZ77 (1 of 4)



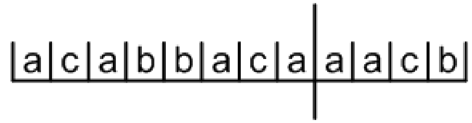
- LZ77 Algorithm (due to Lempel & Ziv, 1977)
  - At any given time, a “window” over the input
    - Left side: already encoded
    - Right side: next piece to encode
  - Replace longest possible prefix with (O,L,C)
    - O: position of prefix on the left side
    - L: length of match
    - C: first non-matching symbol

LZ77 (2 of 4)



- Example of LZ77 encoding
  - Replace baa with (4,2,a)
    - “ba” found at position 4 on the left side
      - First position is 0
    - Length of “ba” is 2
    - Next symbol is “a”
  - If no match, set the length to 0

LZ77 (3 of 4)



- Overlapping example
  – Match can overlap the right side!
  – Replace aac with (7,2,c)
- LZ77 encoder implementation
  – Windows is usually a power of 2
  – Example: 4096+4096 symbols
  – Position: 12 bit can point at entire left side
  – Length: 12 bit can match entire right side

LZ77 (4 of 4)

- Starting the encoder
  - Assume a specific (known) left side
- Disadvantages of LZ77
  - Each triple requires some bytes per match
  - The file can <u>grow</u> with bad matches
  - Symbols are initially encoded as (0,0,c)
    - Encoding starts with a loss!
    - Improvement: put all symbols in initial window

LZSS (1 of 2)

- LZSS algorithm (Storer and Szymanski)
  - Improves upon LZ77
  - Differs in its output codes
  - Two options: match or symbol (no match)
  - Distinguished by first bit of the output
    - Either (O,L): position O, length L
    - Or C: character C (no match)
  - Triples are broken in two

LZSS (2 of 2)

- Implementing LZSS
  - We do not want to deal with 9 bit codes!
  - We split the output in groups of eight codes
  - The first byte describes what follows
    - One bit per code
    - Shows if it is a match or a symbol
    - The next bytes are interpreted accordingly
  - We always process entire bytes (or words)

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Dictionary-based coding

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics

LZ78 (1 of 2)

- Search a dictionary instead of a window
  - Due to the same Lempel and Ziv (1978)
  - Longest input prefix found in the dictionary
  - Replace prefix with (P,C)
    - P: index of prefix in dictionary
    - C: first non-matching symbol
  - Prefix + symbol are added to the dictionary
  - The decoder builds the same dictionary
    - And uses it for decoding

LZ78 (2 of 2)

Input	Output	Dictionary
a	(0,a)	Index 1: a
aa	(1,a)	Index 2: aa
b	(0,b)	Index 3: b
ba	(3,a)	Index 4: ba
ab	(1,b)	Index 5: ab

- Example: input aaabbaab
  - The dictionary gradually gets longer strings
    - In LZ77 you can have long matches much earlier
  - The decoder builds the dictionary from the codes
    - All references are made to previous entries

LZW (1 of 8)

- LZW Algorithm (extended by Welch)
  - LZ78: same logic as LZ77
    - Match prefix + next non-matching symbol
    - Guaranteed progress, even without a match
  - LSW produces only codes, no symbols!
    - The dictionary is initialized with all symbols
    - The next entries are built from those
    - But how can we extend the dictionary?

LZW (2 of 8)

- LZW coding
  - Find longest input prefix in dictionary
  - Replace it with is index
    - Do NOT consume the next symbol
  - Add prefix + next symbol to dictionary
    - This extends the dictionary
  - Move input pointer BEFORE the next symbol
    - The next symbol is the beginning of the next match

LZW (3 of 8)

```
input s;
while not EOF {
   input c;
   if [s+c] is in dictionary
      s = [s+c];
   else {
      output code(s);
      add [s,c] to dictionary with next code;
      s = c; }
}
output code(s);
```

LZW (4 of 8)

Input	Output	Dictionary
		Index 1: a
		Index 2: b
a+a	1	Index 3: aa
aa+b	3	Index 4: aab
b+b	2	Index 5: bb
b+a	2	Index 6: ba
aab+b	4	Index 7: aabb

- Example: input aaabbaabb
  - Dictionary starts with all input symbols

LZW (5 of 8)

- LZW decoding
  - Read the next code
  - If it is in dictionary, replace it in the output
    - Do not add current match in dictionary yet
    - We do not know the next symbol
  - Add instead previous match + first symbol
    - Because we know now what that symbol was
    - So the decoder is always one step behind

LZW (6 of 8)

- What if the code is not in the dictionary?
  - This occurs if the code is for the latest entry
  - But we have not yet added it on our side!
    - We do not know what the next symbol is, yet
  - The match must have been of the form C???C
    - This is the only way for this problem to appear
    - So, we get the previous match
    - And add its first symbol at its end

LZW (7 of 8)

```
s = NIL; // previous string
while not EOF {
  input c;
  entry = string(c); // current string
  if entry not in dictionary
   entry = s + s[0];
  output entry;
  if (s != NIL) // only happens once
   add [s,entry[0]] to dictionary with next code;
  s = entry; // current string becomes previous
  }
```

LZW (8 of 8)

Input	Output	Dictionary
		Index 1: a
		Index 2: b
1	a	
3	aa	Index 3: aa
2	b	Index 4: aab
2	b	Index 5: bb
4	aab	Index 6: ba

- Example LZW decoding
  - Code 3 points at an empty index
  - Must be previous match + first symbol (a+a)

Optimizations (1 of 3)

- Dictionaries for LZ78/LZW
  - They grow in each step!
  - Extensible pointers/indexes
    - We start with (say) 4 bit indexes (16 θέσεις)
    - When dictionary full, add 1 bit to indexes
  - What happens if it grows too much?
    - Either stop adding entries
    - Or drop least used ones

Optimizations (2 of 3)

- LZ78/LZW dictionary compression
  - Each new entry extends a previous one
    - By one symbol
  - Store pointer to previous entry
    - Plus the new symbol
  - Can this be made efficient?
    - During coding, we need to search the dictionary
    - Can we do this by following pointers?

Optimizations (3 of 3)

- TRIE data structure (lexicographic tree)
  - Each node has characters as childern
  - Each match is a path through the trie
    - Each symbol is a branch
    - When we reach a leaf, we have a match
    - The next symbol is added as a new leaf
  - Improves coding speed
  - Decoding follows a similar logic

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

End of Section # 6

Class: Multimedia Technology, **Section # 6:** Entropy Coding

Instructor: George Xylomenos, **Department:** Informatics