Strings: Data Compression

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Wednesday 17th August, 2022

Data Compression

Compression reduces the size of a file:

- To save space when storing it.
- To save time when transmitting it.
- Most files have lots of redundancy.

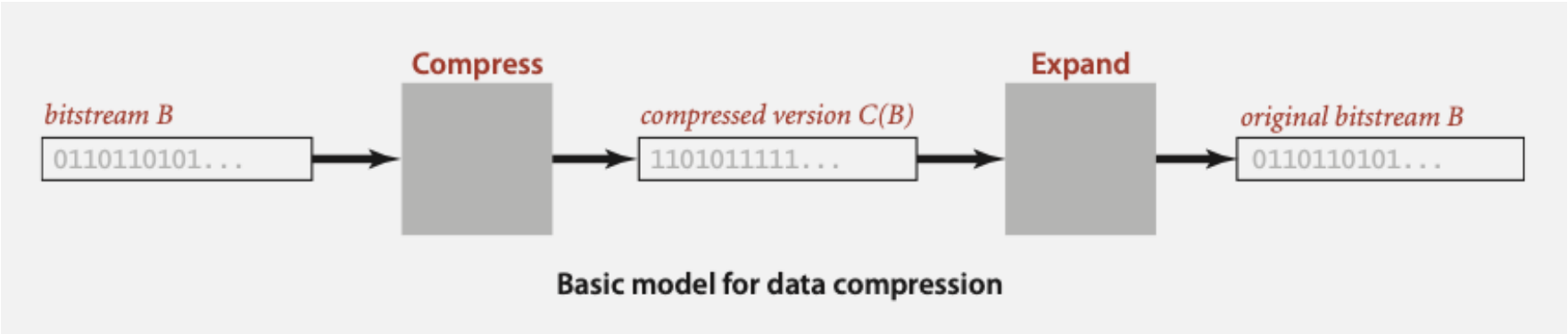
All of the types of data that we process with modern computer systems have something in common: they are ultimately represented in binary.

Lossless compression and expansion

Message. Binary data B we want to compress.

Compress. Generates a "compressed" representation C(B). [uses fewer bits (you hope)]

Expand. Reconstructs original bitstream B. [**no information is lost**]



Basic model for data compression

Compression ratio. Bits in C (B) / bits in B.

Ex. 50-75% or better compression ratio for natural language.

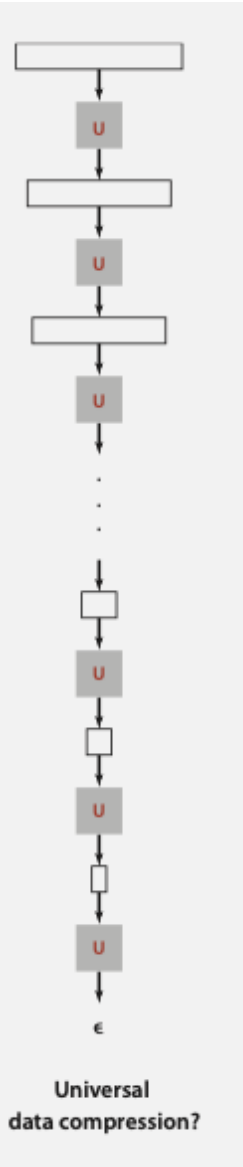
Universal data compression

Proposition. No algorithm can compress every bit string.

Pf 1. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bitstring B_0, compress it to get smaller bit string B_1.
- Compress B_1 to get a smaller bit string B_2.
- Continue until reaching bit string of size 0.

Implication: all bit strings can be compressed to 0 bits!



Redundancy in English Language

Q. How much redundancy is in the English language?

" ... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to denmtrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in meniang. " — Graham Rawlinson

A. Quite a bit.

Run-length encoding - bitstream

Run length encoding for bitstream

Used for simple type of redundancy in a bitstream. In particular, for long runs of alternate blocks of 0's and 1's repeated bits.

Example: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 [40 bits]

Representation. 4-bit counts to represent alternating runs of 0s and 1s: 15 0s, then 7 1s, then 7 0s, then 11 1s.

$\underbrace{1\ 1\ 1\ 1}_{15}\ \underbrace{0\ 1\ 1\ 1}_{7}\ \underbrace{0\ 1\ 1\ 1}_{7}\ \underbrace{1\ 0\ 1\ 1}_{11}$ – 16 bits (instead of 40)!

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Run-length encoding - characters

Run length encoding for characters

$$w = \underbrace{aaaaaaaaaaaaaa}_{14}\underbrace{bbbbbbbbbbb}_{11}\underbrace{ccccccccccccccc}_{15}$$

Representing w in ASCII requires 320 (8*40) bits, its run-length encoding is: a14b11c15.

It requires 48 bits (8 bits each to represent the numbers and characters in ASCII).

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Different encoding models

Static model. Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

Dynamic model. Generate model based on text.

- Preliminary scan of the text needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Ex: LZW.

Variable-length codes

Variable-length encoding: Uses different number of bits to encode different chars.

Why and how to use variable-length encoding?

- Enable most-frequently used characters to use the fewest number of bits, and the least-frequently used characters to use the most.
- To encode a string X, we then represent each character in X with its associated variable-length code word, and we concatenate all these code words in order to produce a binary coded representation, Y, for X.
- However, we need to address the ambiguity caused by the variable length code in distinguishing when the code for a letter ends and when the code for the next letter begins.

Variable-length codes

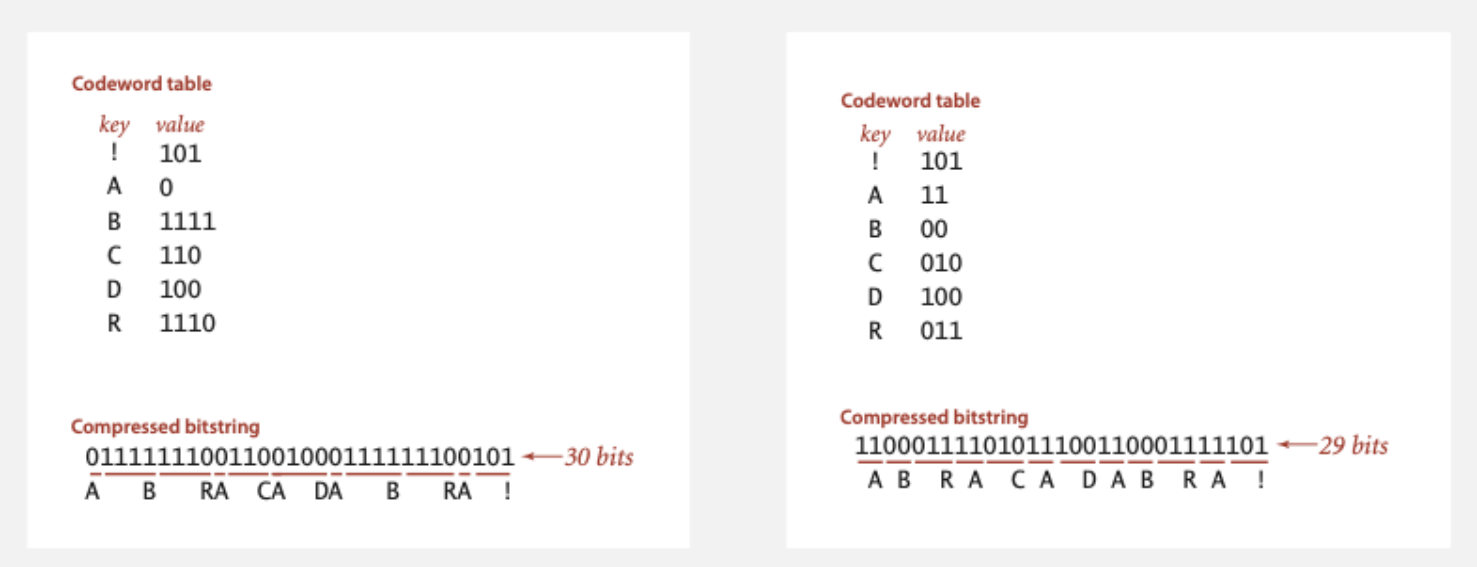
Q. How do we avoid/address the ambiguity?

A. Ensure that no codeword is a prefix of another.

Ex 1. Stick to fixed-length code.

Ex2. Append special stop char to each codeword.

Ex 3. Generate prefix-free code.



Variable-length codes advantage

The space savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

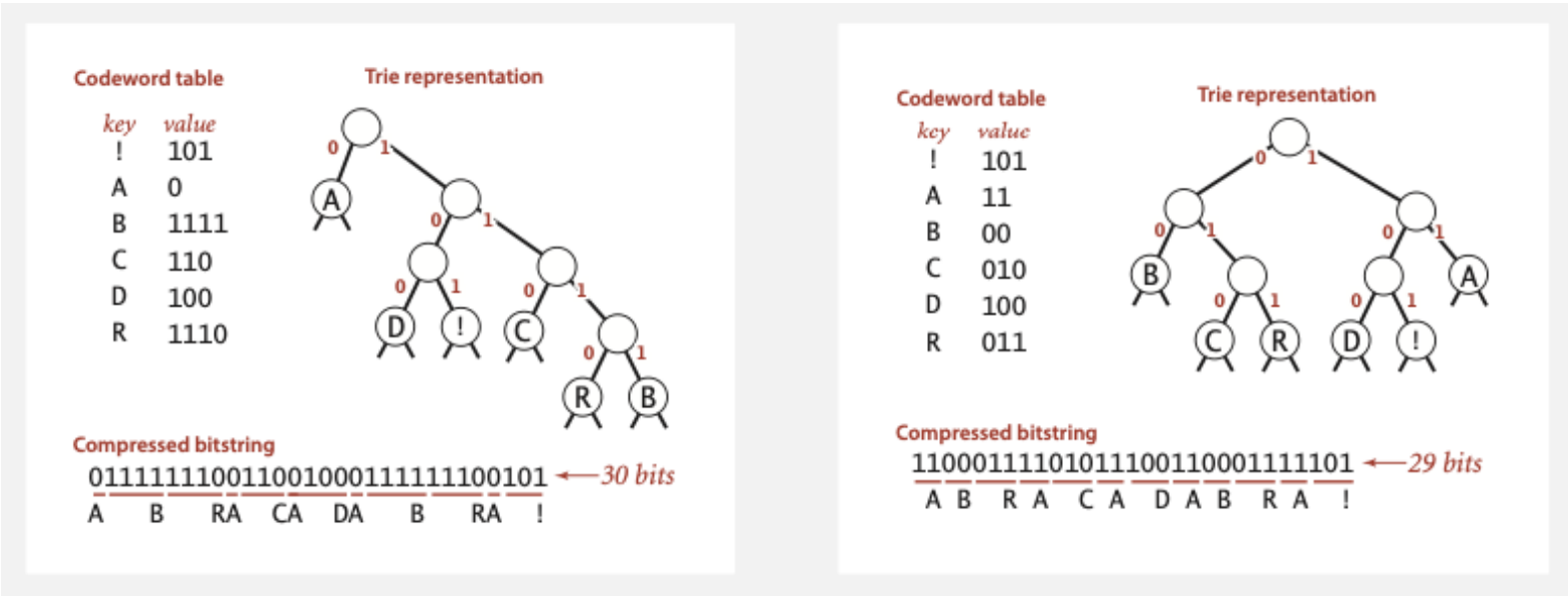
Processing advantage: The advantage of using a prefix code is that decoding can be accomplished by using the greedy strategy of processing the bits of Y in order, repeatedly matching bits to the first code word they represent.

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. Use a binary trie!

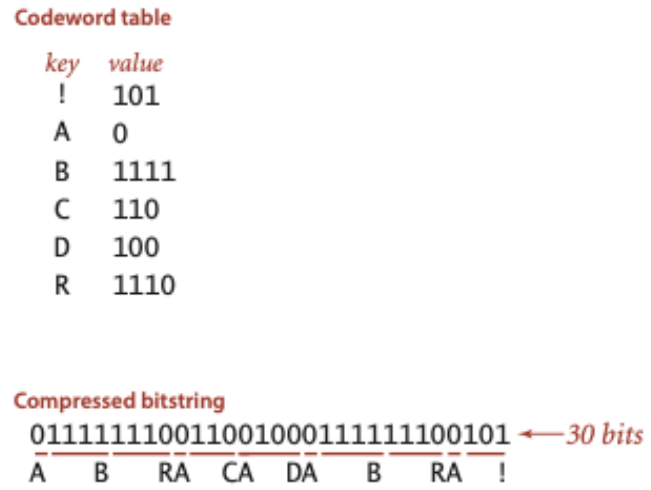
- Chars in leaves.
- Codeword is path from root to leaf.
- Left links labelled as 0 and right links labelled as 1.



Prefix-free codes: compression and expansion

Compression.

- Method 1: create ST of key-value pairs.
- Method 2: start at leaf; follow path up to the root; print bits from root to leaf to encode the node at leaf.
- Example: Prefix code for $A = 0, B = 1111, C = 110, D = 100, R = 1110$.



Prefix-free codes: compression and expansion

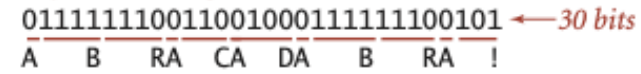
Expansion (Method 1).

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.

Codeword table

key	*value*
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring



Huffman coding

- Saves a substantial amount of space in natural language files, and many other kinds of files.
- Instead of using the usual 7 or 8 bits for each (ASCII) character to store text files, use shorter binary codes for each character using fewer bits.
- It uses a custom prefix-free code for each message.
- It uses shorter code for more frequently occurring symbols, than for those that appear rarely.

Huffman coding overview

Dynamic model. Use a custom prefix-free code for each message.

Compression.

1. Read message.
2. Build best prefix-free code for message. How?
3. Write prefix-free code (as a trie) to file.
4. Compress message using prefix-free code and write to file.

Expansion.

1. Read prefix-free code (as a trie) from file.
2. Read compressed message from file and expand using trie.

Huffman code: Build best prefix-free code - I

- Count frequency for each character in the input string.
- Start with one node corresponding to each character `i` with weight equal to the frequency `freq[i]`.
- Repeat until single trie formed:
  - select any two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with a new root node of weight `freq[i]` +`freq[j]`

Sample Input: A B R A C A D A B R A !

char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

Huffman code: Build best prefix-free code - II

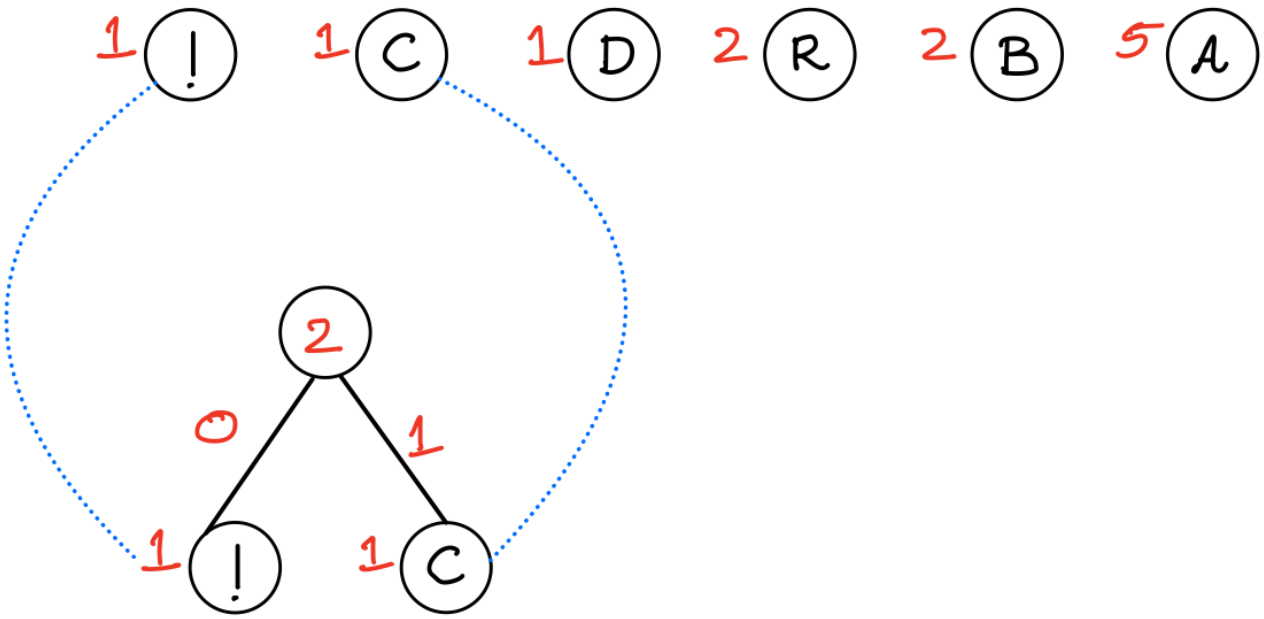
- Count frequency for each character in the input string.
- Start with one node corresponding to each character `i` with weight equal to the frequency `freq[i]`.
- Repeat until single trie formed:
  - select any two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with a new root node of weight `freq[i]` +`freq[j]`

Huffman code: Build best prefix-free code - III

- Count frequency for each character in the input string.
- Start with one node corresponding to each character `i` with weight equal to the frequency `freq[i]`.
- Repeat until single trie formed:
  - select any two tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with a new root node of weight `freq[i]` +`freq[j]`

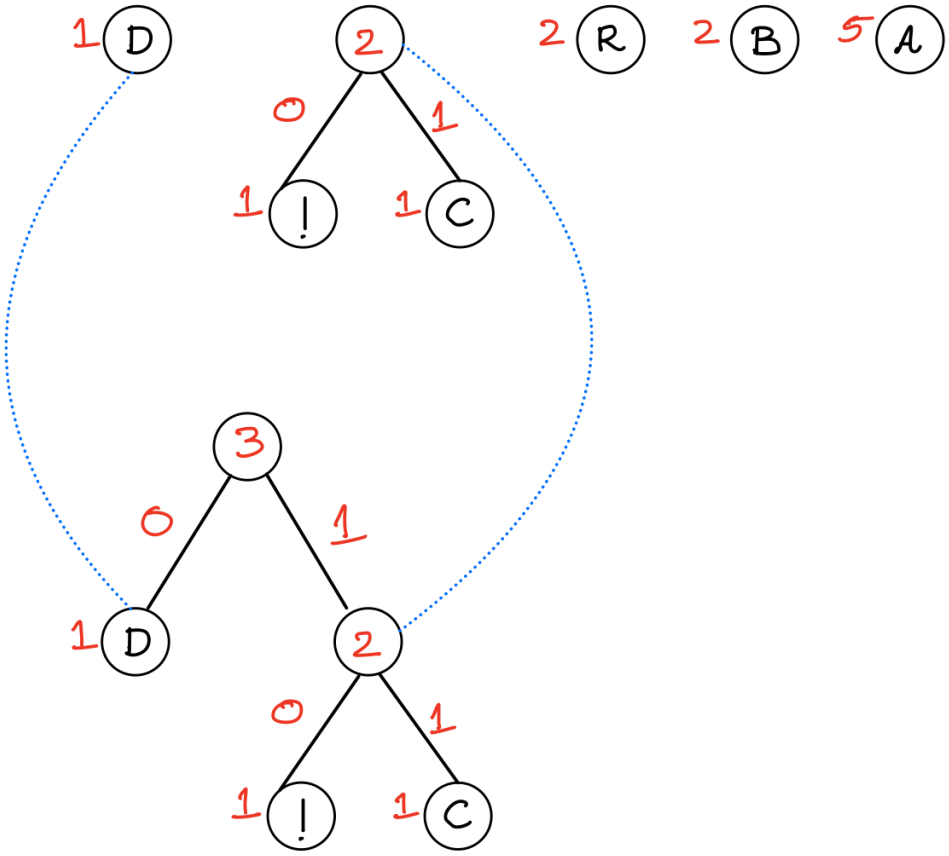
Huffman code: Build best prefix-free code - IV

Merge the two min. weight tries into into single trie with a new root node of weight `freq[i]` + `freq[j]`



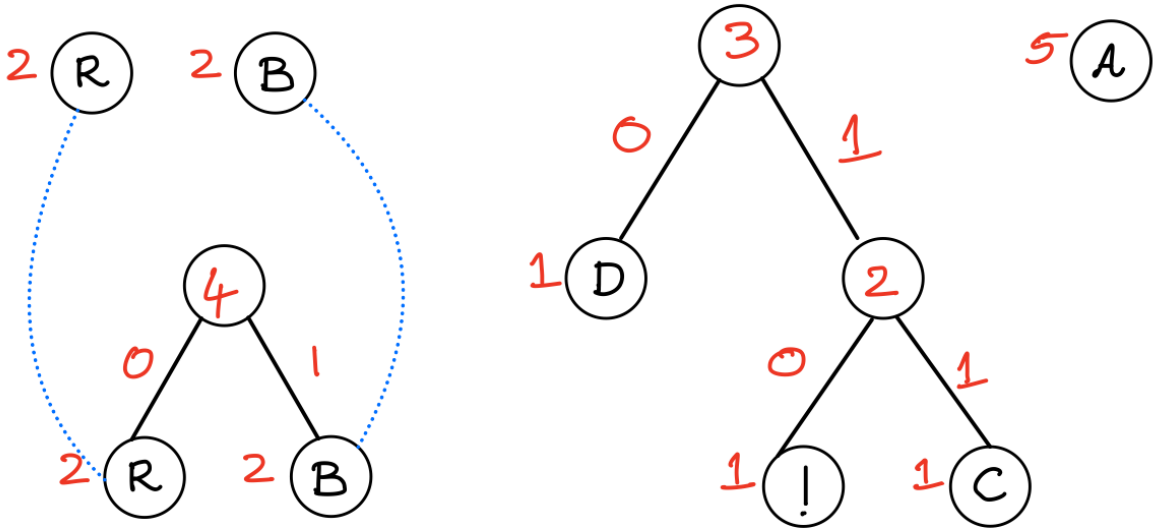
Huffman code: Build best prefix-free code - V

Merge min. weight tries into into single tries.



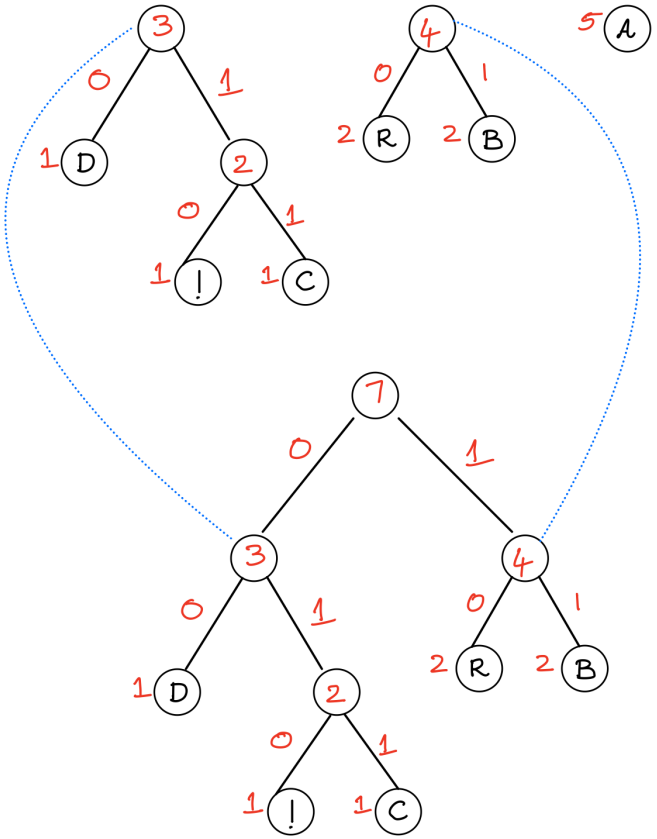
Huffman code: Build best prefix-free code - VI

Merge min. weight tries into into single tries.



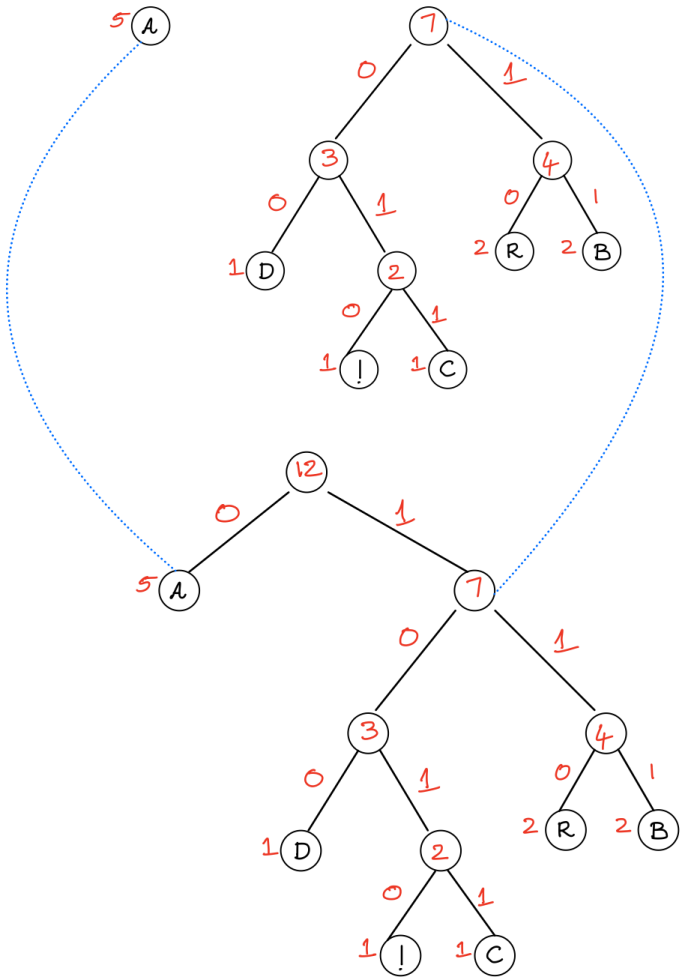
Huffman code: Build best prefix-free code - VIII

Merge min. weight tries into into single tries.



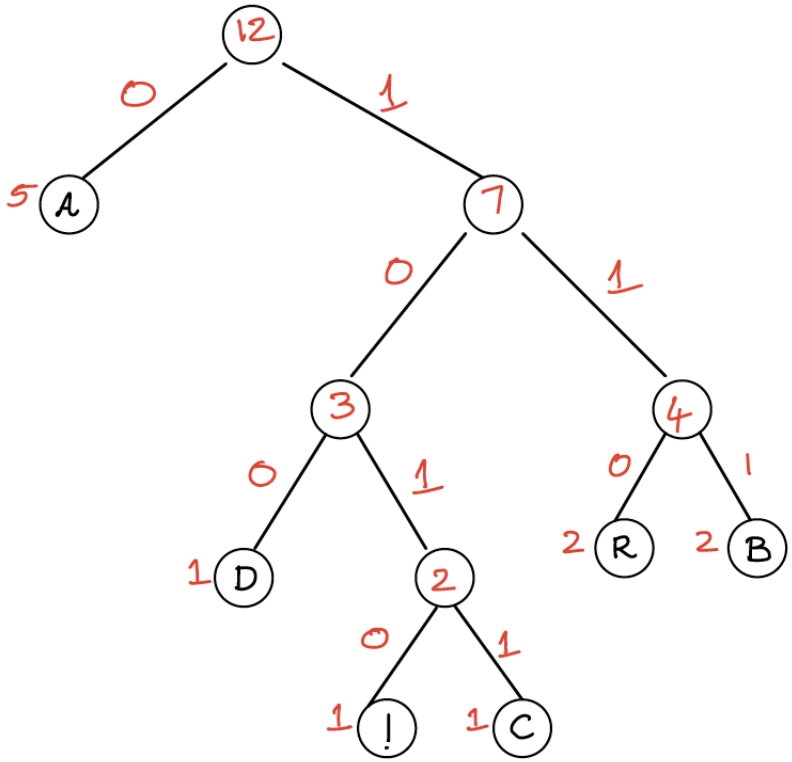
Huffman code: Build best prefix-free code - X

Merge min. weight tries into into single tries.



Huffman Encoding

Optimal encoding for the message: A B R A C A D A B R A !



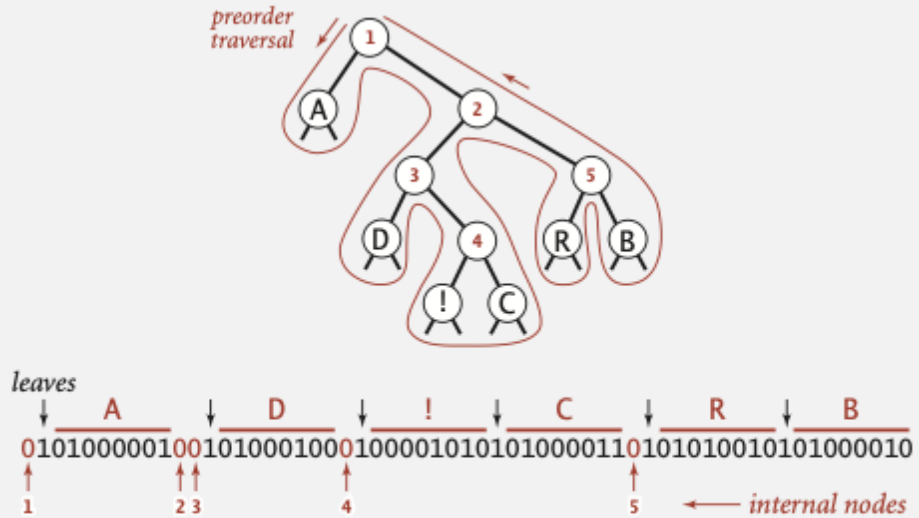
Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code (no prefix-free code uses fewer bits).

Prefix-free codes: how to write/transmit?

Q. How to write the trie; that is, generate the trie encoding?

A. Write preorder traversal of trie.
- Mark leaf with 1 bit, followed by the 8-bit ASCII code of the character in the leaf.
- Mark internal nodes with 0 bit.

Note. If message is long, overhead of transmitting trie is small.



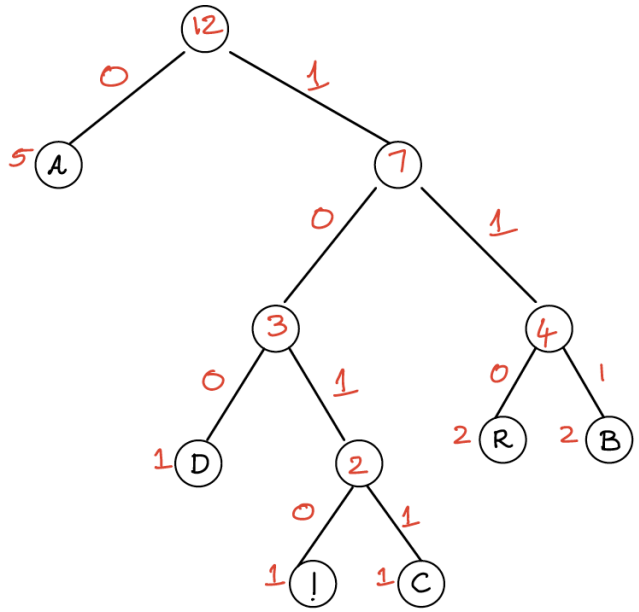
Using preorder traversal to encode a trie as a bitstream

The preorder traversal of the trie in the above figure is:
`1 A 2 3 D 4 ! C 5 R B`.
Substituting, all the internal nodes with bit 0 and the characters with bit 1 followed by its 8-bit ASCII code, we get the encoding shown above in the figure.

Huffman code: Compressed string

Encoding of the characters in the input string: ABRACADABRA!

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Single trie built from Huffman encoding strategy.

Compressed bit string:

$\underbrace{0}_{A}\underbrace{1\ 1}_{B}\underbrace{1\ 1\ 0}_{R}\underbrace{0}_{A}\underbrace{1\ 0\ 1\ 1}_{C}\underbrace{0}_{A}\underbrace{1\ 0\ 0}_{D}\underbrace{0}_{A}\underbrace{1\ 1}_{B}\underbrace{1\ 1\ 0}_{R}\underbrace{0}_{A}\underbrace{1\ 0\ 1\ 0}_{!}$ – 28 bits

Note that the above is ONLY the compressed string not the complete encoded string.

Full data compression scheme

- Note that savings achieved by Huffman encoding is not the total savings achieved.
- This is due to the fact that compressed bitstream cannot be decoded without the trie.
- Hence, we must account for the cost of including the trie in the compressed output, along with the compressed bit string.
- encoded string (as a bitstream) = trie encoding + compressed string.
- For long inputs, this cost is relatively small, but in order for us to have a full data-compression scheme, we must write the trie onto a bitstream when compressing and read it back when expanding.

Huffman Encoding: Expansion

Expansion.

- Read prefix-free code (as a trie) from file.
- Read compressed message from file and expand using trie.

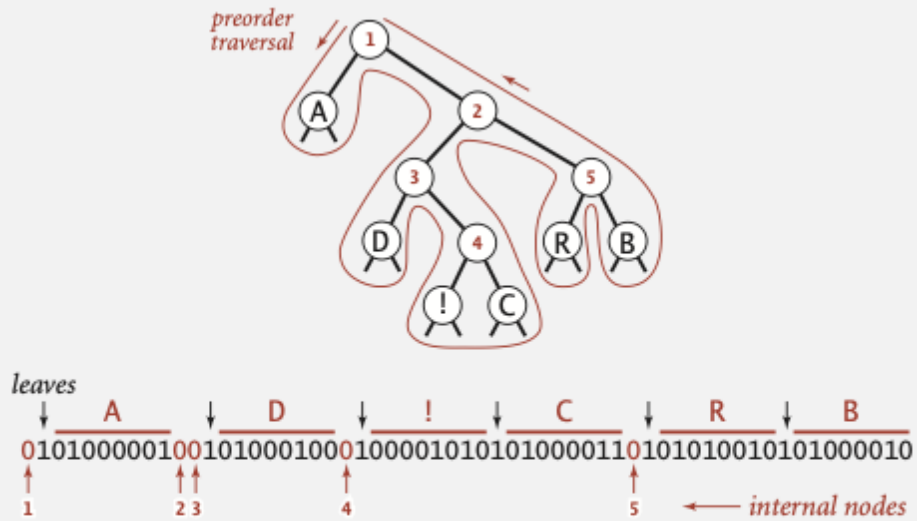
Prefix-free codes: how to read?

Q. How to read in the trie encoding?

A. Reconstruct trie from the bit string by preorder traversal of trie.
- Read a single bit to learn which type of node comes next:

- If the bit is 1, then it is a leaf node, so read the next character and create the corresponding leaf node in the trie.

- If the bit is 0, then it is an internal node, so create the corresponding internal node in the trie, and then (recursively) build its left and right subtrees.



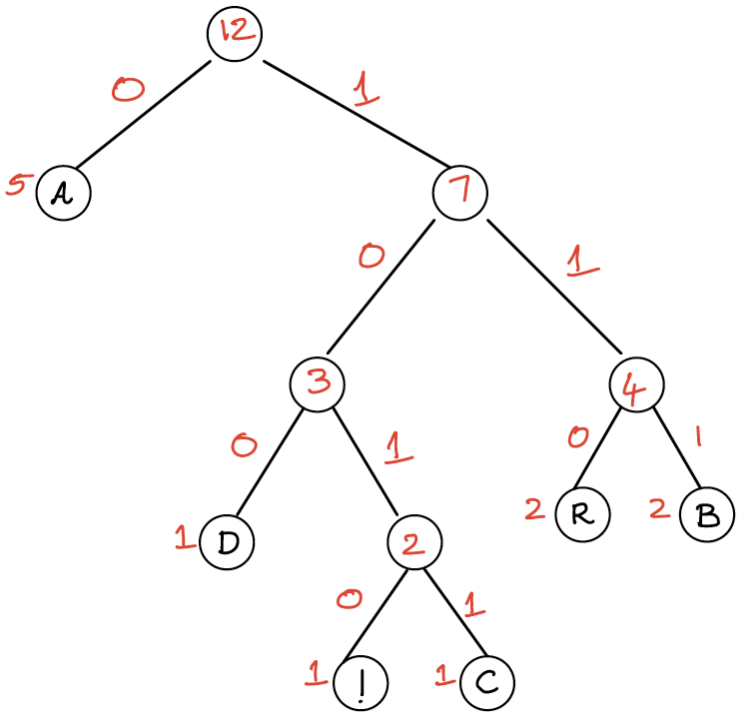
Using preorder traversal to encode a trie as a bitstream

Read compressed message

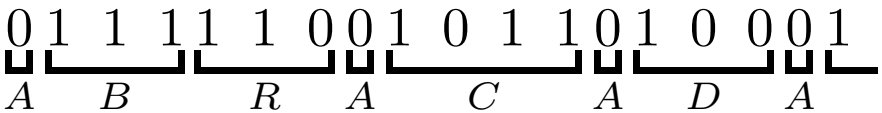
Compressed bit string:

0 1 1 1 1 1 0 0 1 0 1 1 0 1 0 0 0 1 1

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.



blueCompressed bit string:



Lempel-Ziv-Welch (LZW) Compression

- One of the most widely used compression methods because it is easy to implement and works well for a variety of file types.
- It maintains a symbol table of fixed-length codewords for variable-length patterns/strings in the input.
- It does not require encoding of the symbol table!

LZW Algorithm for Compression

The algorithm performs the following steps as long as there are unscanned input characters:

1. Find the longest string s in the symbol table that is a prefix of the unscanned input.
2. Write the fixed length codeword associated with s in the output.
3. Scan one character past s in the input. Let this character be c. c is termed as the lookahead character.
4. Associate the next available codeword value with $s + c = sc$ (c appended/concatenated to s) in the symbol table, and add it to the symbol table.

LZW Compression Algorithm Example - I

Input string: **A B R A C A D A B R A B R A B R A**.

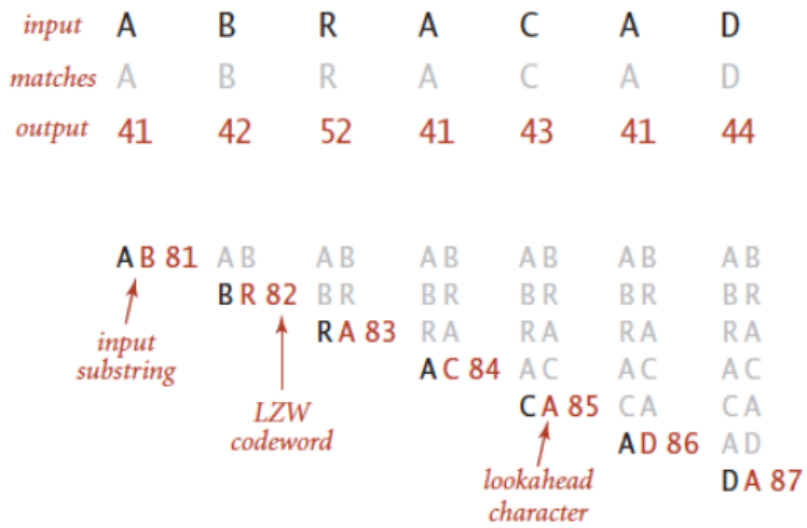
- For this example the input string consists of 7-bit ASCII characters.
- The encoded string; that is, the output consists of 8-bit codewords.
- The symbol table is initialized with 128 possible single character string keys and associate them with 8-bit codewords obtained by prepending 0 to the 7-bit value defining each character.
- For economy and clarity, hexadecimal is used to refer to codeword values, so 41 is the codeword for ASCII A, 52 for R, and so forth.
- Codeword 80 is reserved to signify end of file.
- The rest of the codeword values (81 through FF) are assigned to various substrings of the input that we encounter, by starting at 81 and incrementing the value for each new key added.
- For the moment, we simply stop adding entries to the symbol table when we run out of codeword values (after assigning the value FF to some string).

LZW Compression Algorithm Example - II

Input string: **A B R A C A D A B R A B R A B R A**.

Unscanned characters are marked red, and scanned ones are marked green.

- The symbol table is initialized with 128 possible single character string keys and associate them with 8-bit codewords obtained by prepending 0 to the 7-bit value defining each character.
- For the first seven characters, the longest prefix match is just one character, so we output the codeword associated with these characters, and build the symbol table further based on the lookahead characters. by associating the codewords from 81 through 87 to two character strings.

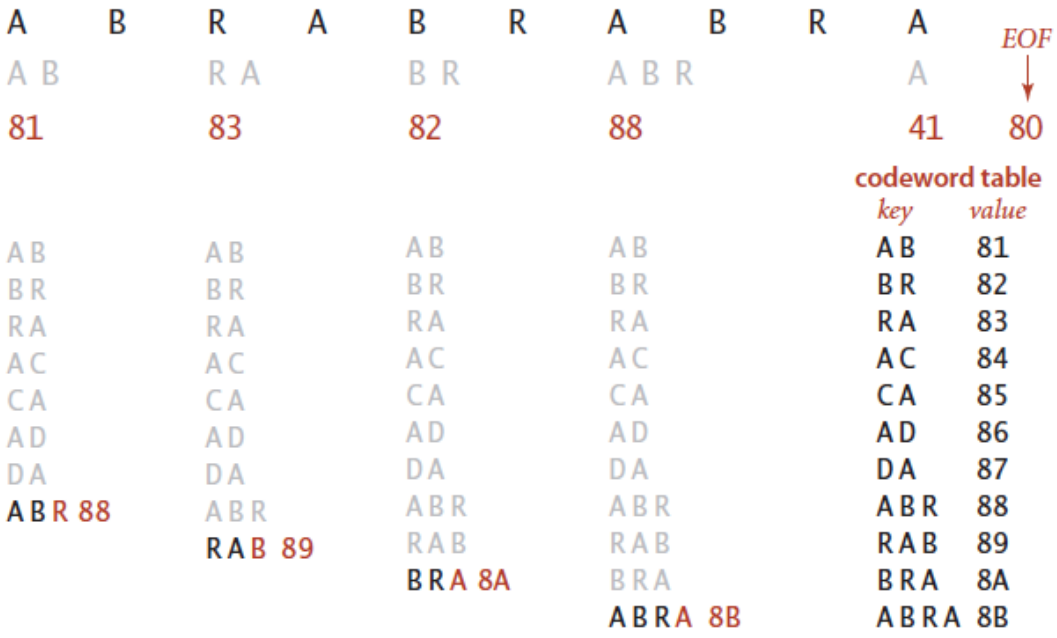


LZW Compression Algorithm Example - III

Input string: **A B R A C A D A B R A B R A B R A**.

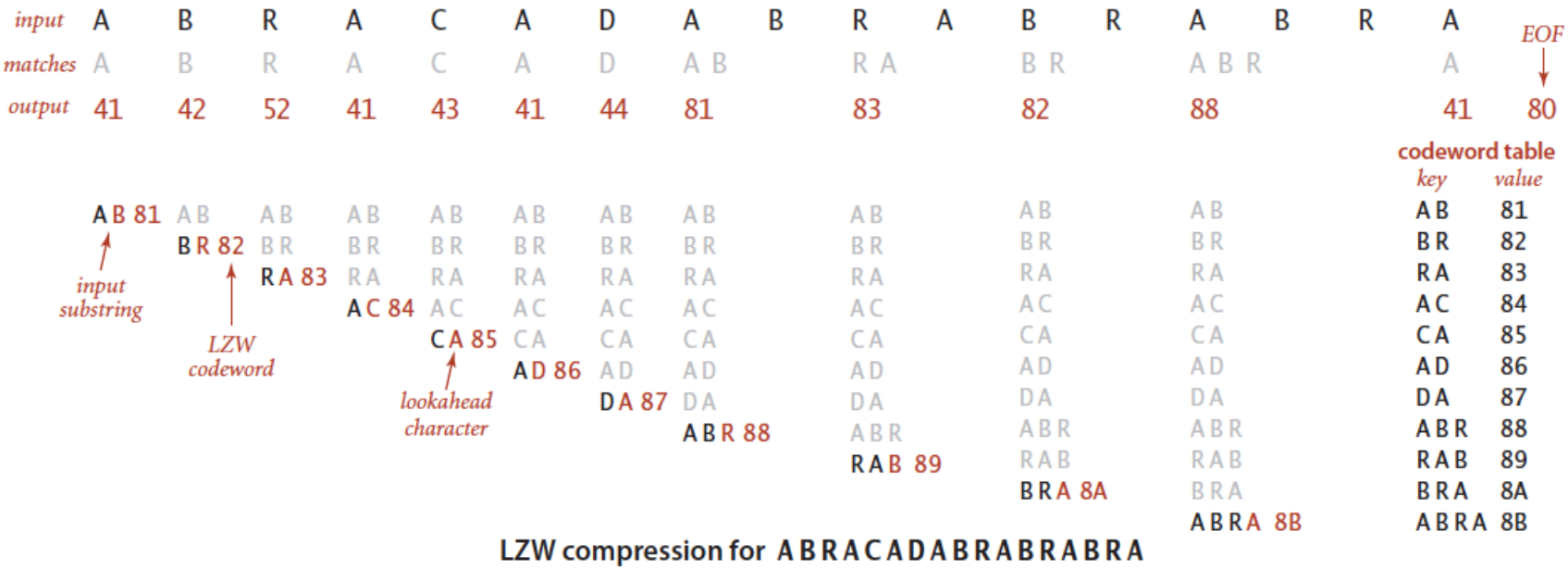
Unscanned characters are marked red, and scanned ones are marked green.

- Then we find prefix matches with AB (so we output 81 and add ABR to table), RA (so we output 83 and add RAC to the table), BR (so we output 82 and add BRA to the table), and ABR (so we output 88 and add ABRA to the table), leaving the last A (so we output its codeword, 41).
- Finally we output 80 (code for end of the file), signifying the end of the input.



LZW Compression Algorithm Example: all put together

Input string: **A B R A C A D A B R A B R A B R A**.



LZW compression for A B R A C A D A B R A B R A B R A

LZW Compression Algorithm Example: all put together

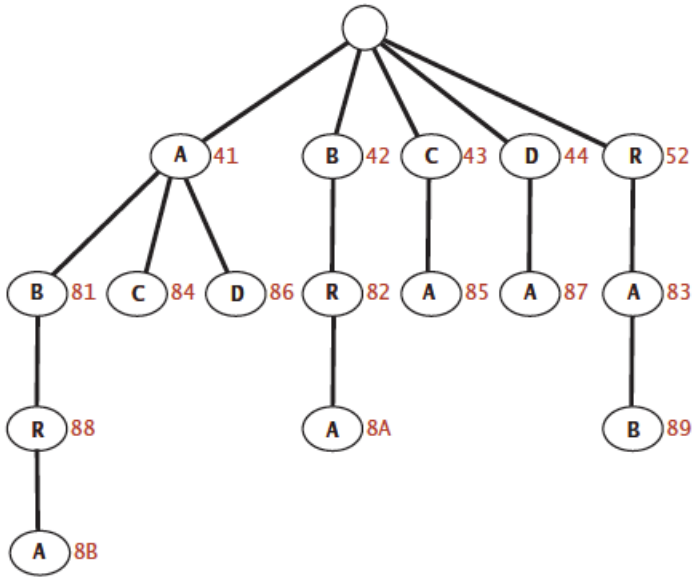
- The input is 17 ASCII characters of 7 bits each for a total of 119 bits.
- The output is 12 codewords of 8 bits each for a total of 96 bits - a compression ratio of $96/119 \approx 80$ percent even for this tiny example.

LZW Compression symbol table representation

Q. How to represent the LZW compression symbol table?

A. A trie to support the longest prefix match.

- To find a longest prefix match, we traverse the trie from the root, matching node labels with input characters.
- To add a new codeword, we connect a new node labeled with the next codeword and the lookahead character to the node where the search terminated.



LZW Expansion Algorithm

The expansion algorithm reads the codeword string, one codeword at a time, and performs the following steps until it reads the codeword 80 (end of file):

1. Read a codeword X from the input.
2. Write the string value (val), associated with the X to the output.
3. Set S to the value associated with Y (the next codeword after X) in the symbol table.
4. Associate the next unassigned codeword a value equal to val + c in the symbol table, where c is the first character of S.

LZW Expansion Algorithm Example - I

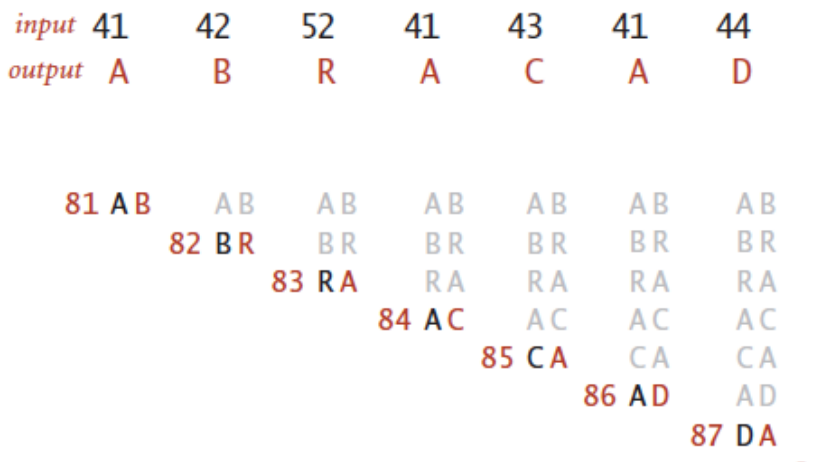
- The input for LZW expansion in our example is a sequence of 8-bit codewords, and the output is a string of 7-bit ASCII characters.
- To implement expansion, we maintain a symbol table that associates strings of characters with codeword values (the inverse of the table used for compression).
- We fill the table entries from 00 to 7F with one-character strings, one for each ASCII character, set the first unassigned codeword value to 81 (reserving 80 for end of file).

LZW Expansion Algorithm Example - II

Input string: **41 42 52 41 43 41 44 81 83 82 88 41 80**.

Unscanned characters are marked red, and scanned ones are marked green.

- The symbol table/inverse codeword table is initialized with entries from 00 to 7F (8-bit codewords) with one-character strings (7-bit ASCII characters).
- For the first seven characters, the longest prefix match is just one character, so we output the codeword associated with these characters.
- In the process, we build the symbol table further based on the lookahead characters by associating the codewords from 81 through 87 to two character strings as indicated below.

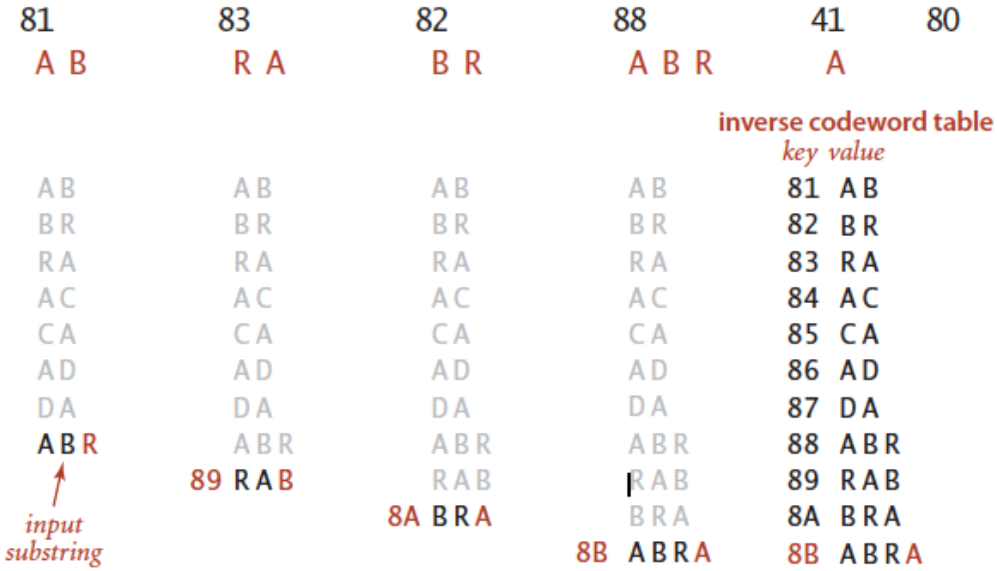


LZW Expansion Algorithm Example - III

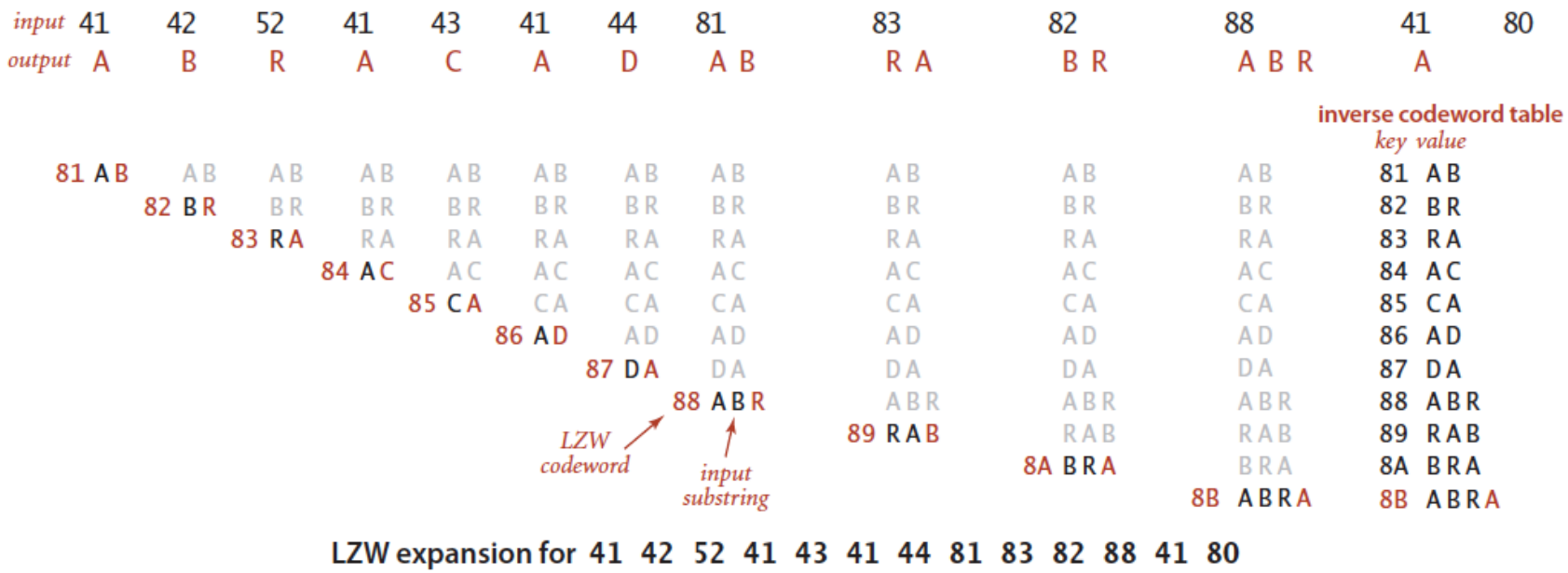
Input string: **41 42 52 41 43 41 44 81 83 82 88 41 80**.

Unscanned characters are marked red, and scanned ones are marked green.

- Then we read 81 (so we write AB and add ABR to the table), 83 (so we write RA and add RAB to the table), 82 (so we write BR and add BRA to the table), and 88 (so we write ABR and add ABRA to the table), leaving 41. Finally we read the end-of-file character 80 (so we write A).



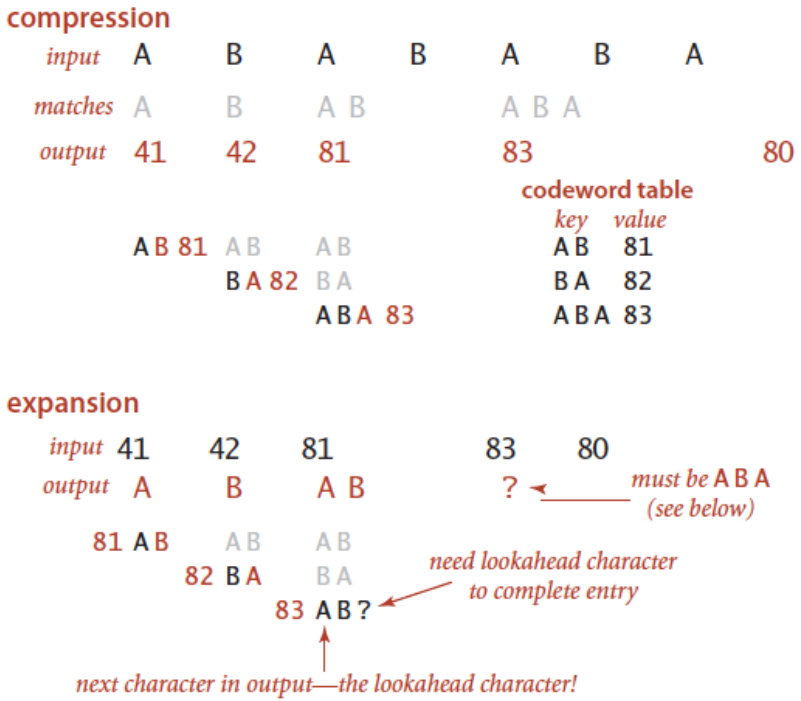
LZW Expansion Algorithm Example: all put together



Q. How to represent LZW expansion code table?

A. An array.

LZW Expansion Algorithm: Tricky Case



- It happens precisely when the codeword is the same as the table entry to be completed.
- To correct it (the lookahead character must be the first character in that table entry, since that will be the next character to be output).
- In this example, this logic tells us that the lookahead character must be A (the first character in ABA). Thus, both the next output string and table entry 83 should be ABA.

The end of course!!

Final exam material and Format

1. The exam will be based on the material covered in the lectures slides.
2. The exam will consist of 10 multiple choice questions and 6 descriptive questions. In particular, at least 3 descriptive questions will be from chapters 4 and 5.
3. For the test you can refer to all lecture slides, your class notes, all tutorials, assignments and the textbook.

Tips for preparing for the exam

1. At least 3 descriptive questions will be from chapters 4 and 5. So make sure you know the material well.
2. Know all the algorithms. The best way to understand an algorithm is to do a dry run (take and example and execute the algorithm step by step).
3. For all algorithms know what kinds of inputs will give you the best/worst running time/no.of comparisons.
4. First go through the lecture slides, then tutorial/mid-term solutions and then assignments. Use the textbook to understand the lecture slides better.
5. Note that KMP algorithm is not from the textbook, make sure you know the one from the lecture slides.