Adaptive Huffman code, Dictionary coders

Coding Technology

Illés Horváth

2025/10/29

Reminder: character encodings

Character codings: prefix codes, code tree for decoding.

Fixed length encoding, Shannon–Fano code, Huffman code.
Huffman is optimal among character codings.

Entropy as a measure of information. Theoretical lower bound.

Arithmetic coding is not a character encoding, but it is asymptotically optimal (also called an entropy coding).

Adaptive Huffman code

Shannon–Fano coding, Huffman coding and Arithmetic coding all use the source distribution. What if this information is not available?

Main idea of Adaptive Huffman coding: instead of using the source distribution, build the code tree based on the number of characters seen so far in the text, then change the tree as the text progresses.

This is a possible algorithm:

1. Initialize the code tree (e.g. 1 occurrence for each character).
2. Read the next character from the text and code it according to the current state of the code tree.
3. Increase the count for the current character by 1.
4. Rebuild the code tree according to the new character counts.
5. Move ahead to the next character and repeat from step 2.

Adaptive Huffman code

When reading a new character, why is it important to code first (using the old code tree), then increase the count for the new character after?

The order of these steps is important for decoding:

1. Initialize the code tree (e.g. 1 occurrence for each character).
2. Decode 1 character according to the current state of the code tree.
3. Increase the count for the decoded character by 1.
4. Rebuild the code tree according to the new character counts.
5. Move ahead in the bit sequence and repeat from step 2.

Adaptive Huffman code

This is essentially the Adaptive Huffman code; however, we will improve it a little bit.

The main issue is that rebuilding the entire code tree after every character is very costly computationally.

If the source characters do actually have an asymptotic frequency (just unknown), then the code tree will typically converge fast and remain unchanged after a while.

So instead of rebuilding the tree in every step, it would be nice to have a simple check whether the code tree needs to be changed at all.

Also, even if the tree has to be changed, maybe we can do some local changes to fix it instead of entirely rebuilding it.

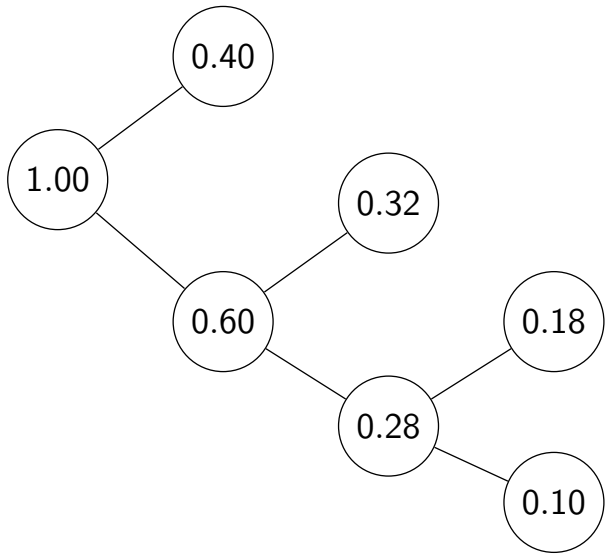
Huffman trees

We want to understand the structure of code trees obtained using Huffman coding better.

During the construction of the tree, probabilities are summed up and assigned to internal nodes of the tree; the tree, along with the probabilities for every node, are collectively referred to as a *Huffman tree*.

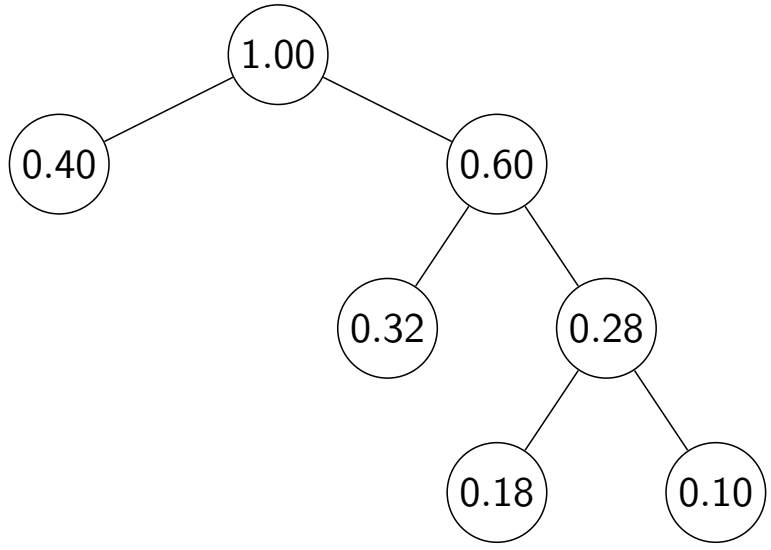
Example.

A	0.40
B	0.32
C	0.18
D	0.10



Sibling property

It will be convenient to depict a Huffman tree from top to bottom instead of left to right:

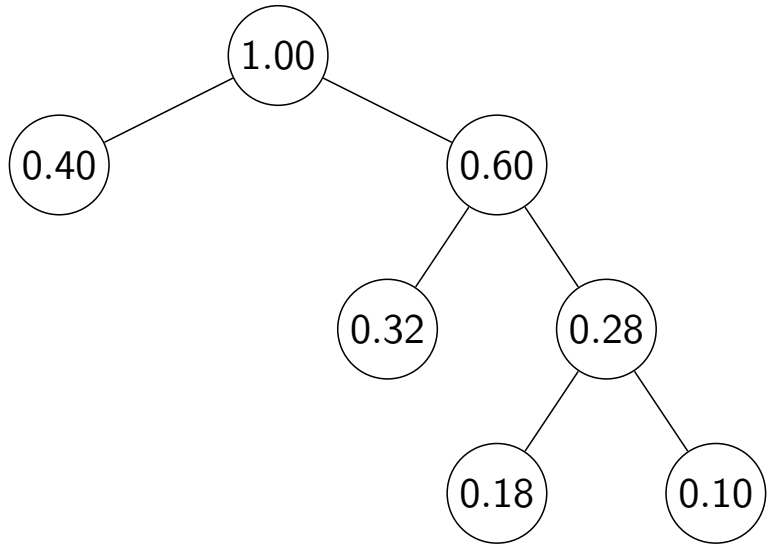


We say that two nodes are *siblings* if they have a common parent node.

A tree has the *sibling property* (sometimes called the *sibling pair property*) if listing the nodes from bottom level to top level, going from left to right within each level, the weights of the nodes are increasing, except possibly for sibling pairs.

Sibling property

It will be convenient to depict a Huffman tree from top to bottom instead of left to right:



Does this tree have the sibling property?

List the nodes from bottom level to top level, left to right within each level (sibling pairs marked with (,)):

$$(0.18, 0.10), (0.32, 0.28), (0.40, 0.60), 1.00$$

Apart from some sibling pairs, the list is increasing, so the sibling property holds for this tree.

Sibling property

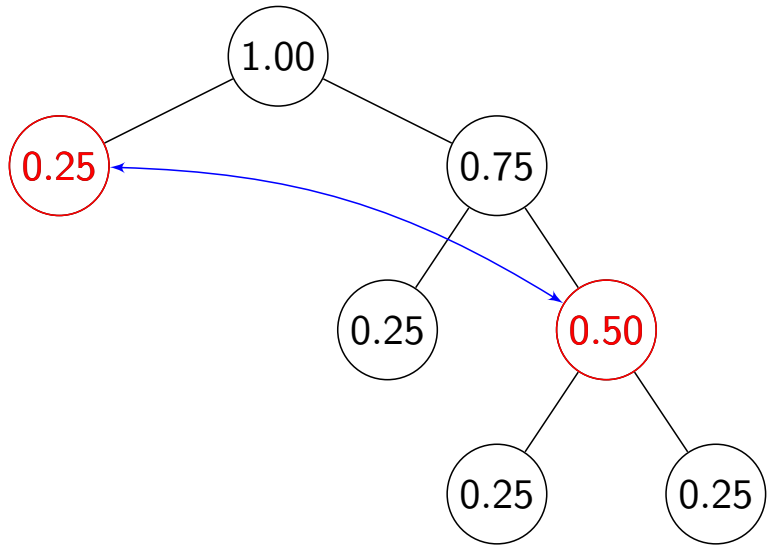
The sibling property can be formulated in several equivalent ways:

- The sibling property holds for a tree if sibling pairs can be listed in an order such that their weights are increasing (non-decreasing, so equality is allowed).
- When listing the nodes according to increasing weight, sibling pairs are consecutive elements of the list, and nodes from a level closer to the root come later in the list.

Despite the name, the sibling property is not so much about sibling pairs; it is more about the tree being balanced (with respect to the weights in the nodes).

Sibling property

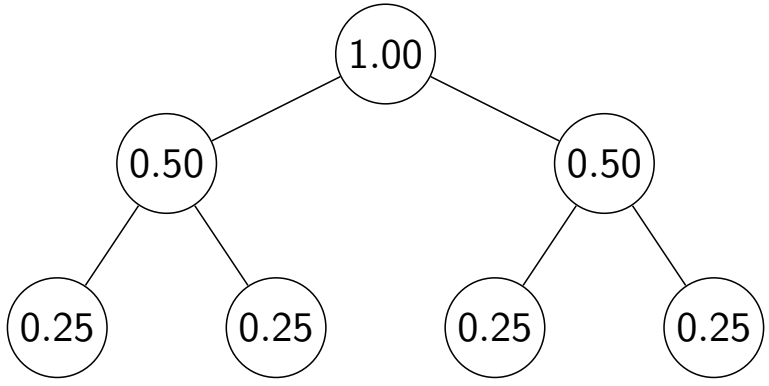
Example. This is a tree that does not have the sibling property.



We can fix it by exchanging two nodes (along with their entire subtrees).

Sibling property

Example. Now the tree does have the sibling property.



Note that after the exchange of two nodes, we also have to recalculate the weights of internal nodes, starting from the leaves (nodes at the edge of the tree).

Adaptive Huffman code

Theorem

Every code tree obtained from a source by using the Huffman code has the sibling property.

Every weighted tree that has the sibling property can be obtained by using the Huffman code for some source.

Proof (sketch). During every step of the algorithm for Huffman coding, the two smallest probabilities are summed, so the corresponding nodes will be siblings, and nodes coming later during the algorithm will have larger weights.

(For each sibling pair, the two nodes can be ordered either way.) □

Adaptive Huffman code

We are going to use the sibling property as the check for whether the Huffman tree needs updating, and exchanging nodes as the local fix to restore the sibling property if necessary.

1. Initialize the code tree (e.g. 1 occurrence for each character).
2. Read the next character from the text and code it according to the current state of the code tree.
3. Add the character to the tree (increase the count for the current character by 1, along with its parents etc. toward the root).
4. Check the sibling property, and if violated, restore it by exchanging two nodes with their entire subtrees.
5. Move ahead to the next character and repeat from step 2.

Note that in this algorithm, the weights in the tree are the number of occurrences instead of probabilities, so they do not sum up to 1 in the root. This causes no issues.

Adaptive Huffman code

The restoration step is the following:

- take the character last added to the tree (sibling property will be violated here), and
- exchange it with the node farthest away (along the tree) with weight 1 less.

These two nodes are going to be exchanged along with their entire subtrees, then the weights are recalculated among all nodes between the root and either of the exchanged nodes.

Sibling pairs can be in either order (increasing or decreasing). Some versions of the algorithm order sibling pairs in every step - that's also fine, it will just make changes to the tree more often.

The exact details of the algorithm are important in order to decode correctly!

Decoding

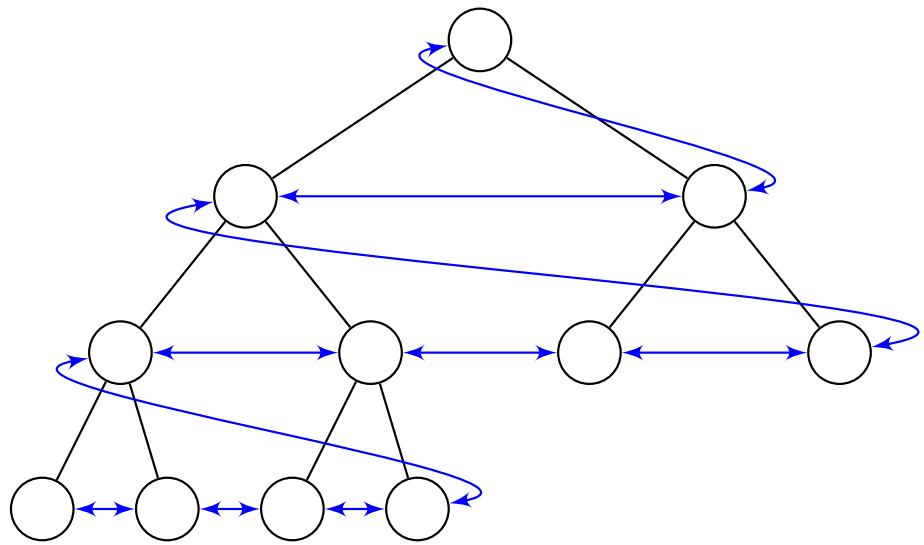
The decompression (decoding) algorithm

1. Initialize the code tree (e.g. 1 occurrence for each character).
2. Decode 1 character according to the current state of the tree.
3. Add the character to the tree.
4. Check the sibling property, and if violated, restore it by exchanging two nodes with their entire subtrees.
5. Move ahead in the text and repeat from step 2.

Steps 1 and 4 have to be executed identically during encoding and decoding.

Data structure

The tree is stored as a binary tree; to make checking the sibling property easy, we add a doubly linked list of the nodes from bottom to top level, from left to right. Sibling property is easy to check along the doubly linked list.

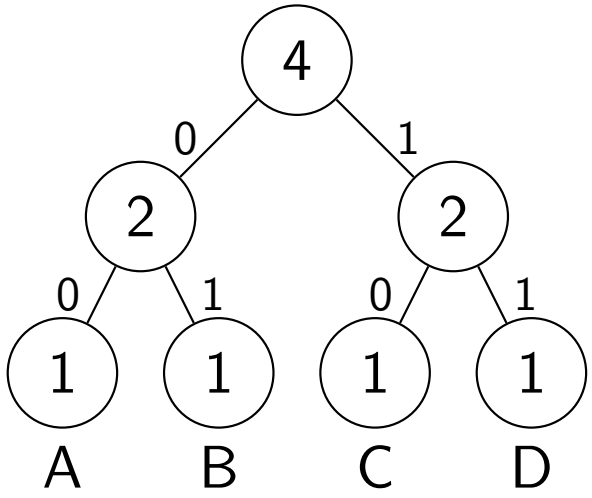


(When the tree changes, both the tree and the doubly linked list need to be maintained.)

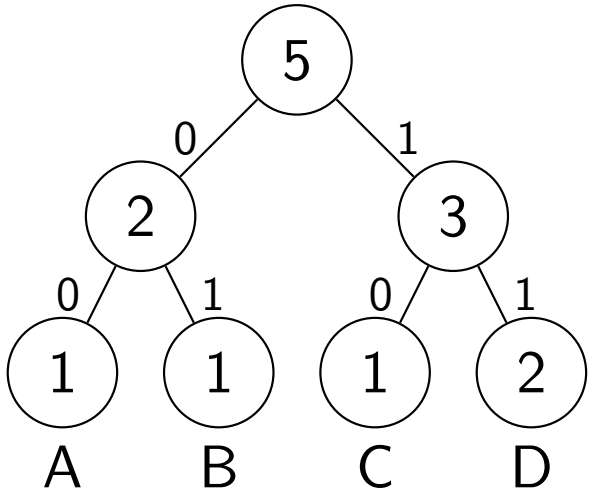
Adaptive Huffman code

Example. For the source alphabet {A,B,C,D}, make an adaptive Huffman code for the message DCDADD.

Initialization.

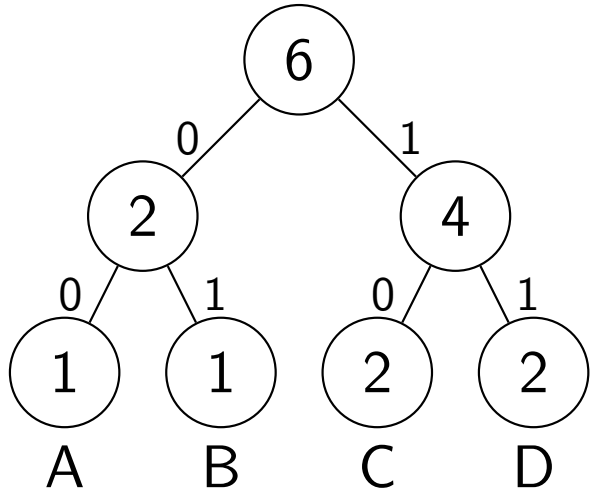


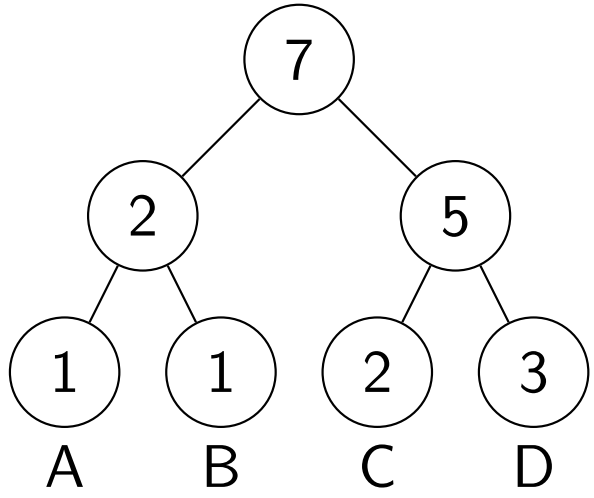
D $\rightarrow$ 11



DC $\rightarrow$ 1110

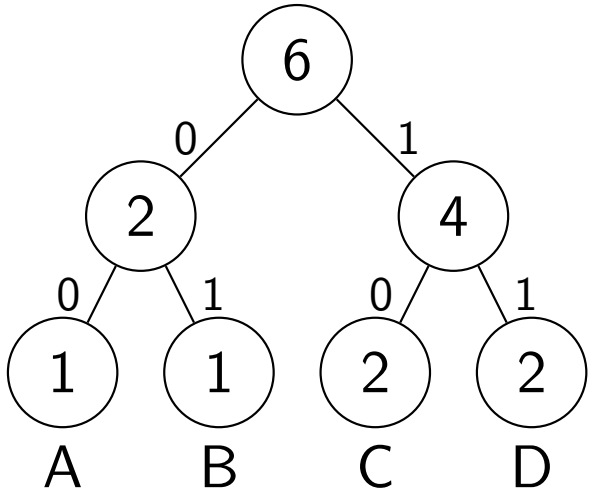
Adaptive Huffman code

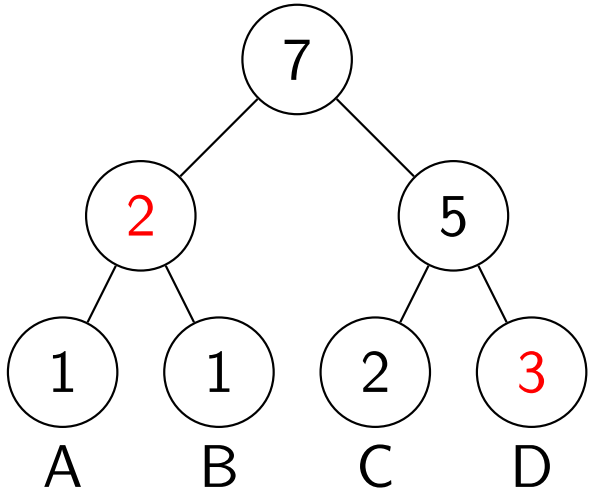




DCD $\rightarrow$ 111011

Adaptive Huffman code

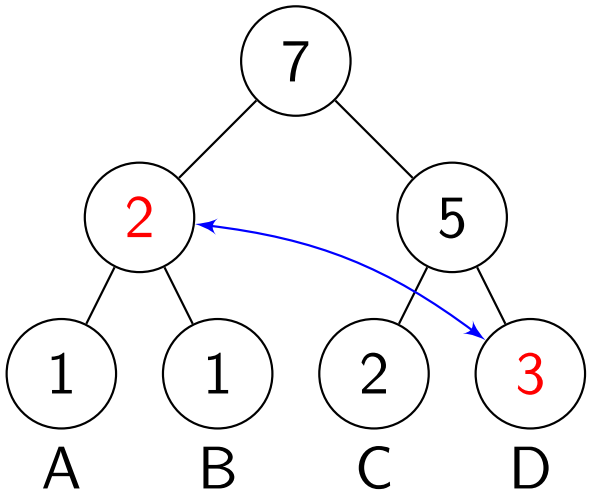


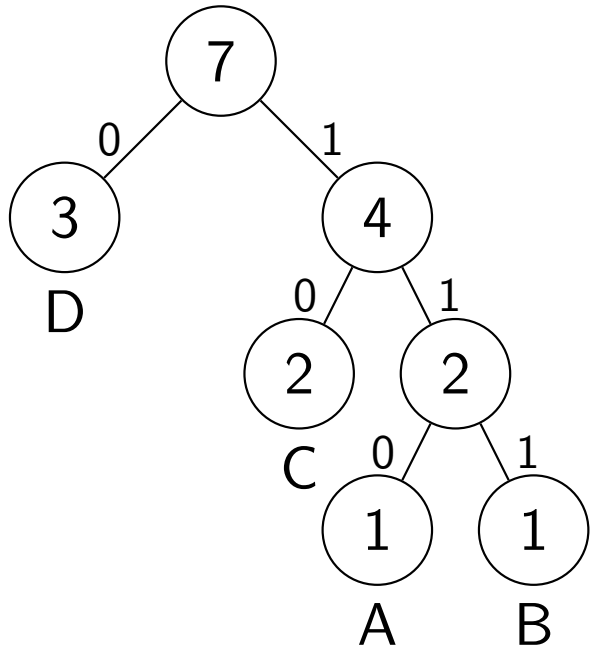


DCD $\rightarrow$ 111011

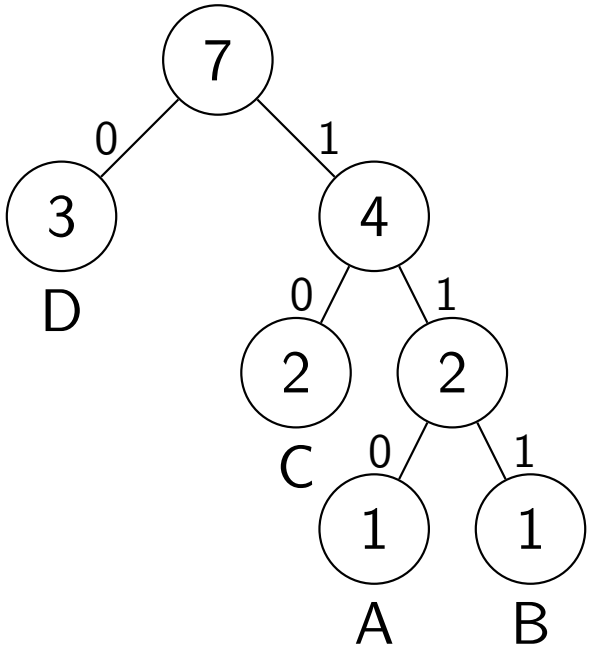
Adaptive Huffman code

Restoration step.

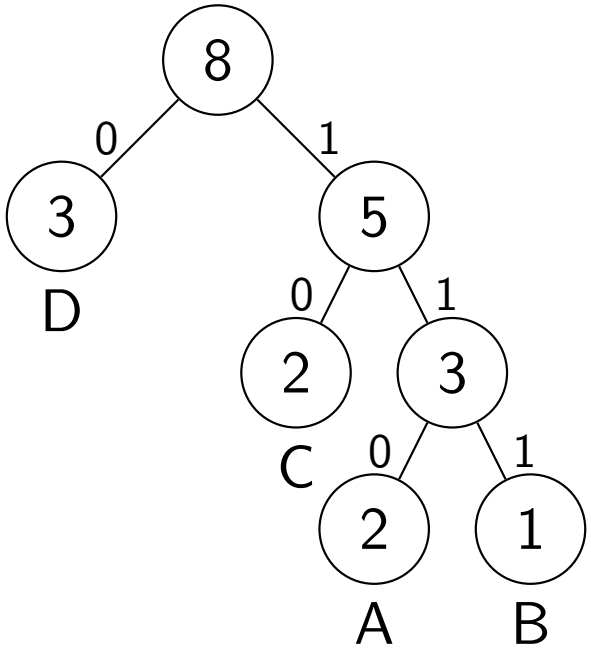




Adaptive Huffman code

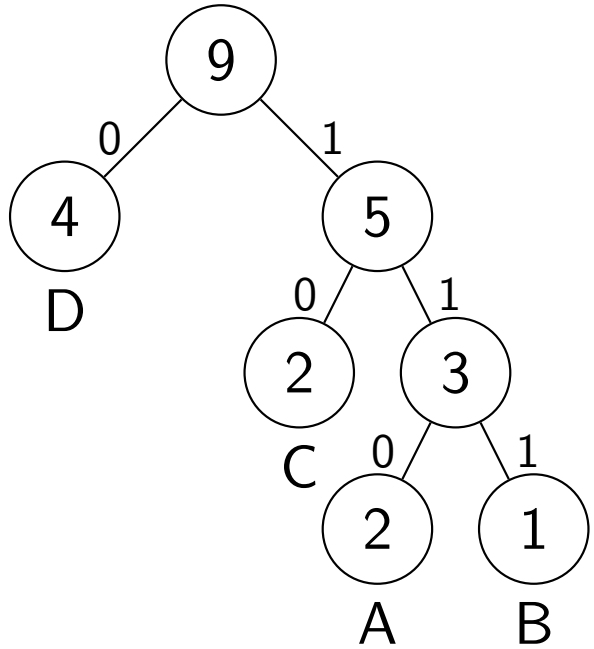


DCDA → 111011110

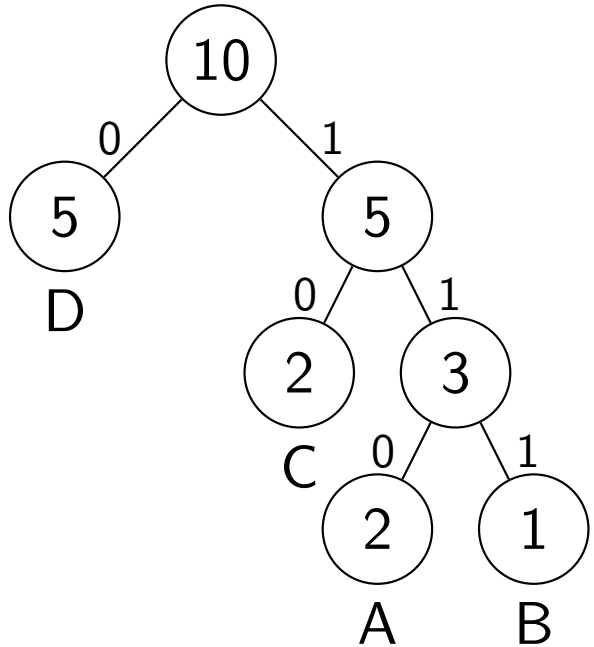


DCDAD → 1110111100

Adaptive Huffman code



DCDADD$\color{red}{D}$ $\rightarrow$ 11101111000



DCDADD $\rightarrow$ 11101111000

Static universal codings

What to do when even the size of the alphabet is unknown?

Universal codes are character encodings that map the positive integers to binary codewords and are prefix codes.

Since character probabilities are not available, the codewords can not depend on the character probability distribution either, hence *static* codeword assignments are used.

Static universal codings are not optimal, but are still useful in certain settings:

- when the size of the alphabet is unknown;
- when the character probability distribution is not known to the receiver;
- for short messages with many different characters.

Elias gamma coding

For the Elias gamma coding, the codeword assigned to a positive integer n is the following: compute n in base 2 binary form, then add one fewer 0's at the beginning than the length of the base 2 number.

n	$n_{②}$	c_n	impl. prob.
1	1	1	$1/2$
2	10	010	$1/8$
3	11	011	$1/8$
4	100	00100	$1/32$
5	101	00101	$1/32$
6	110	00110	$1/32$
7	111	00111	$1/32$
8	1000	0001000	$1/128$
9	1001	0001001	$1/128$
10	1010	0001010	$1/128$
⋮	⋮	⋮	⋮

Static universal coding

Static universal codings have an *implied probability distribution* of the form

$$p_n = 2^{-|c_n|};$$

the coding is actually equivalent to a Huffman coding for the implied probability distribution.

Technically, we cannot use the usual Huffman coding algorithm for an infinite source alphabet, but if the character probabilities are negative integer powers of 2, we know what the codeword lengths should be, and construct a prefix code for those codeword lengths.

The implied probability distribution is specific to each static universal coding; typically p_n is decreasing in n, which means that the resulting code is generally better if the actual character distribution is also decreasing.

Sources with memory

Next we discuss algorithms that look for repeated structures within the source. Such algorithms are typically useful for sources that are not memoryless.

Sources with memory can exhibit very different types of behaviour, depending on how the memory affects the output of the source.

("All happy families are alike; each unhappy family is unhappy in its own way." – Leo Tolstoy, Anna Karenina)

English (or any other language) text is typically structured this way: certain words or expressions are likely to repeat. The repeated expressions might also depend on the topic and type of the text (e.g. scientific, literature etc.)

The entropy of a source with memory is always less than the entropy of a memoryless source with the same character distribution.

Dictionary coders

Instead of pursuing theoretical setups and models for different types of sources with memory, we are going to focus on algorithms.

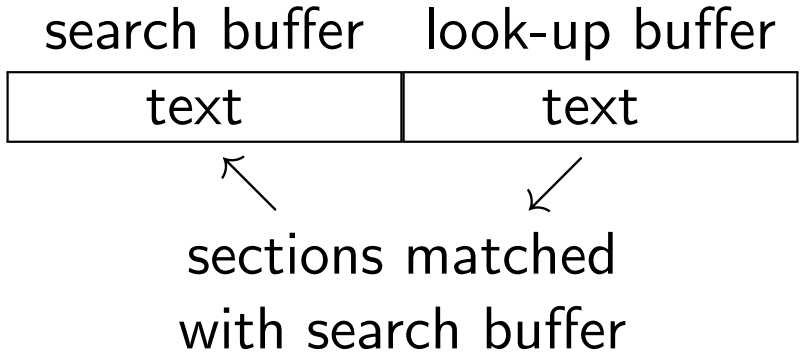
We are going to discuss so-called *dictionary coders*: sections of text are assigned an address, and on later occurrences, are only referenced by that address.

The exact details of how to keep track of repeated sections are different though. Next we are going to discuss two algorithms developed by Lempel and Ziv in the 1970's and 1980's.

We assume the alphabet $\mathcal{X}$ is known and finite, and that's it – the algorithms can be applied without any further information. (Due to this reason, they are sometimes called *universal codings* as well, but they are different from the previous static universal codings!)

LZ77

The LZ77 algorithm maintains two sliding windows:



Parameters:

- h_s: length of the search buffer
- h_ℓ: length of the look-up buffer

Output is a sequence of records containing (p, ℓ, n), where:

- p: position of the beginning of matching section (backwards from the cursor);
- ℓ: length of the matching section
- c: next character after the matching section

LZ77

Example. Compress the source text

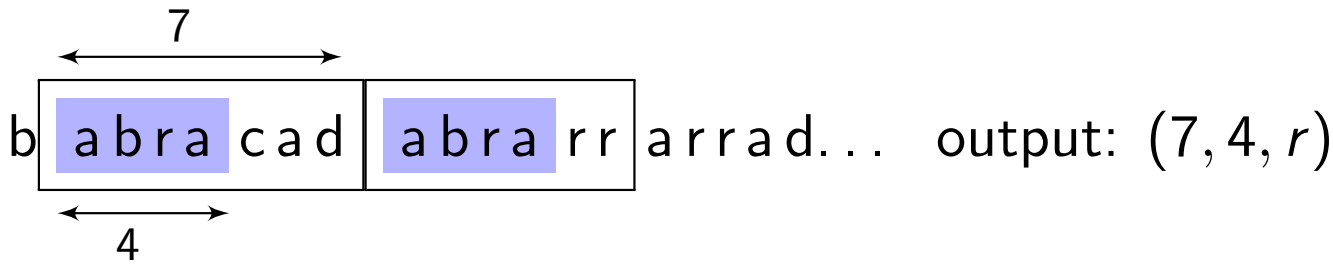
$$\mathsf{b\,a\,b\,r\,a\,c\,a\,d\,a\,b\,r\,a\,r\,r\,a\,r\,r\,a\,d\ldots}$$

with LZ77. Parameter values are $h_s = 7, h_\ell = 6$. Cursor is initially at position 7.

- initial state:

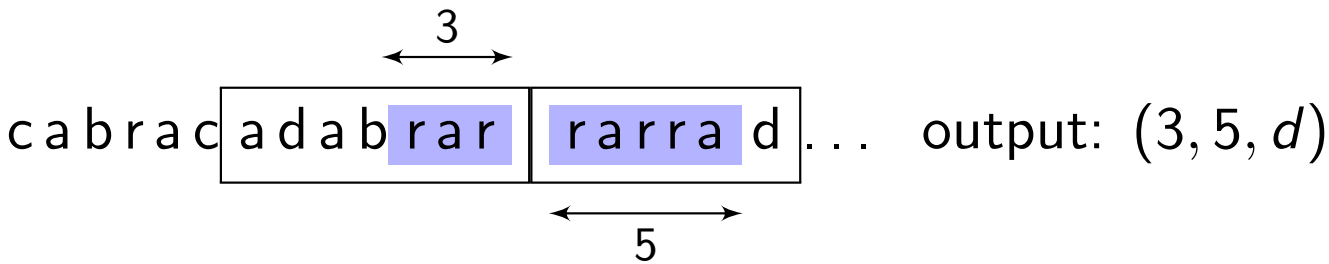
  b a b r a c a | d a b r a r r a r r a d. . . output: $(0, 0, d)$

- next step:



output: $(7, 4, r)$

- next step:



output: $(3, 5, d)$

LZ77

Records are then converted to bit sequences.

Size of a single record (in bits):

$$\lceil \log_2 h_s \rceil + \lceil \log_2 (h_\ell + h_s) \rceil + \lceil \log_2 |\mathcal{X}| \rceil$$

LZ77 is a naive algorithm that will typically not be asymptotically optimal in any setting, but can offer good compression for sources where longer sections of text repeat frequently.

LZ78 algorithm

The LZ78 algorithm also aims to identify sections of text seen before, but this time, we arrange the sections into an ever-expanding dictionary during coding.

We parse the text into sections that are 1 character longer than a section seen before. The output (coding for the new section) is (i, c), where

- i is the address of the old section (1 character shorter than the current section),
- c is the new character (novelty factor).

The address of the current section is incremented by 1 for each new section.

LZ78 algorithm

1	(0,a)
2	(0,b)
3	(1,b)
4	(2,b)
5	(4,a)
6	(5,b)

Example.

a b a b b b b b a b b a b

Coding.

1. "a" is a new section, so the output is (0, a).
2. "b" is a new section, so the output is (0, b).
3. "a" is an old section with address 1; the new section is "ab", the new character is "b", so the output is (1, b).
4. "b" is an old section with address 2; the new section is "bb", and the new character is "b", so the output is (2, b).
5. "bb" is an old section with address 4; the new section is "bba", and the new character is "a", so the output is (4, a).
6. "bba" is an old section with address 5; the new section is "bbab"
   and the new character is "b", so the output is (5, b).

LZ78 algorithm

Each record is converted into a bit sequence.

- for record n, the pointer to the old address is converted into $\lceil \log_2 n \rceil$ bits;
- the novelty factor is converted into $\lceil \log_2 |\mathcal{X}| \rceil$ bits.

address	record	bit sequence
1	(0,a)	0
2	(0,b)	01
3	(1,b)	011
4	(2,b)	101
5	(4,a)	1000
6	(5,b)	1011

a b a b b b b b a b b a b $\quad\rightarrow\quad$ 0010111010001001

LZ78 algorithm

For short messages, LZ78 code tends to make a code longer than the original message.

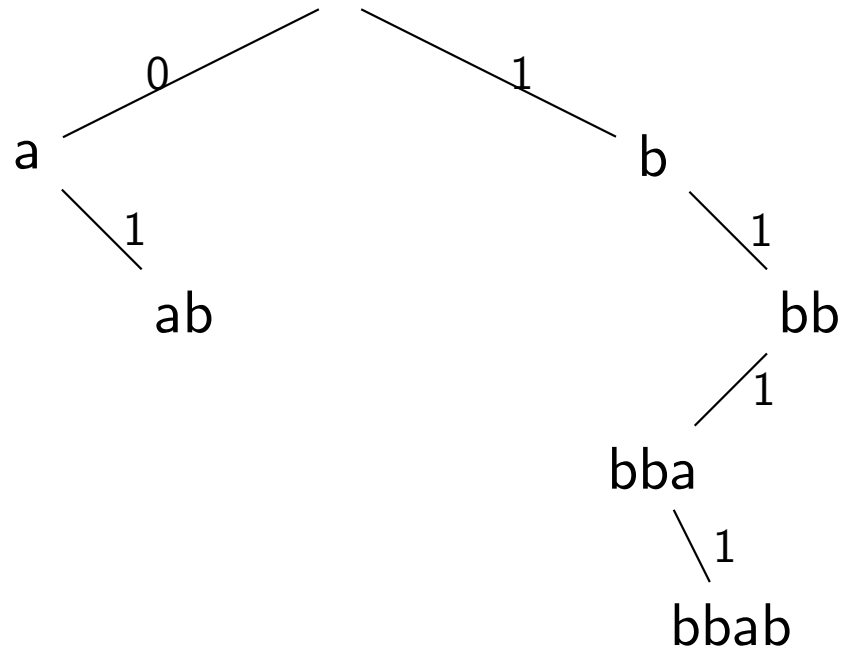
However, LZ78 is asymptotically optimal in many different practical settings.

In practice, asymptotical optimality means that the resulting code might still be longer compared to the entropy of the source, but for very long messages, the difference is negligible.

In general, LZ78 is more efficient for source alphabets of size 2^k for some k integer.

LZ78 algorithm

The dictionary can also be depicted as a code tree. Each node will have at most $|\mathcal{X}|$ children (so the tree is binary only for $|\mathcal{X}| = 2$).



Deflate

"Whenever I want to compress data, I just use zip. So what's with that?"

zip is actually a file format that supports several data compression algorithms, the most widely used of which is Deflate.

High-level description of Deflate. The message is cut into blocks of varying length, and each block uses one of the following three methods:

- no compression (useful for short blocks or blocks that are already compressed);
- compression by LZ77 followed by Huffman coding (code tree included with the compressed block);
- compression by LZ77 followed by a static character coding (no code tree necessary).

The compressor has some freedom in choosing how to cut the message into blocks, and which method to use for each block. (We do not go into more details about these choices.)