Running Time

- Time for setting up of a priority queue of n symbols is O(n).
- Each operation on priority queue takes O($\log n$) time.
- So, due to for loop, the time is O($n \log n$).

Optimal Substructure Property

- Let x and y be pair of symbols with lowest frequencies.
- Consider symbols $C' = C - \{x, y\} \cup \{z\}$.
- Let T' be the tree for an optimal prefix code of C'.
- Construct a new tree T from T' as follow:
  - Make z by an internal node, and
  - Make x and y as the left and the right children of z.

Optimal Substructure Property

Now,

$$ABL(T') = \sum_{c \in C'} f(c) d_{T'}(c)$$

From construction of T, $d_T(c) = d_{T'}(c)$, for $c \in C' - \{z\}$, and:

$$d_T(x) = d_T(y) = d_{T'}(z) + 1$$

Therefore,

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= f(z)(d_{T'}(z) + 1) \end{aligned}$$

Optimal Substructure Property

So,

$$ABL(T) = ABL(T') + f(x) + f(y), \text{ or}$$
$$ABL(T') = ABL(T) - f(x) - f(y)$$

If T is not optimal, then we can replace with optimal tree T'' having lower cost, i.e.,

$$ABL(T'') < ABL(T).$$

It is proved earlier that x and y would be siblings in T''.

Optimal Substructure Property

Now construct a new tree T''' by replacing x, y in T'' with leaf z such that $f(x) + f(y) = f(z)$.
Then T''' represent a prefix code tree for C'. Furthermore,

$$\begin{aligned} ABL(T''') &= ABL(T'') - f(x) - f(y) \\ &< ABL(T) - f(x) - f(y) \\ &= ABL(T') \end{aligned}$$

It contradicts the fact that T' is optimal for C'.
Hence, T is the tree representing optimal prefix code for C.

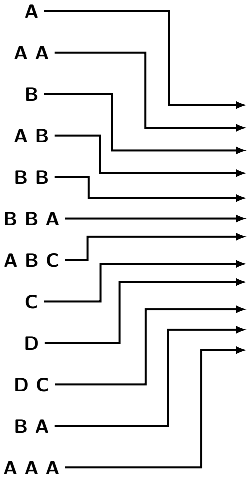
Lempel-Ziv Compression

- Provides a variable-to-fixed length coding scheme.
- Consists of an encoder and decoder.
- Encoder incrementally parses the text into distinct unseen phrases.
- A dictionary is built as the words keep coming for encoding.
- The idea behind the compression is that normally words occur repetitively.
  - A substring that have been seen already more likely to appear again than a substring not seen.

Incremental Parsing

- Uses the concept of incremental parsing of the text.
- Consider the string: AAABABBBBBAABCCDDCBAAAA
- First codeword is a null character (Λ), it is inserted into slot 0 (its index) of dictionary.
- Incoming string is incrementally parsed first character A is not present in the dictionary.
- A prefix is Λ, so codeword for A is $<$index(Λ), A$>$, i.e., $<$0, A$>$.
- Codeword $<$0, A$>$ is inserted in slot 1.

Incremental Parsing



Index	String	Codeword
1	A	(0, A)
2	AA	(1, A)
3	B	(0, B)
4	AB	(1, B)
5	BB	(3, B)
6	BBA	(5, A)
7	ABC	(4, C)
8	C	(0, C)
9	D	(0, D)
10	DC	(9, C)
11	BA	(3, A)
12	AAA	(2, A)

A|AA|B|AB|BB|BBA|ABC|C|D|DC|BA|AAA

Amount of Compression

- Total number of bits = # of symbols * 8 = 23 * 8 = 184 bits
- Index i requires $\log i$ bits for $i > 1$.
- In fact, requirement will be determined by the significant bits in i.
  - For example, codeword (0, A), (1, A), (0, B), (1, B), (0, C), (0, D) reach require 1 index bit.
  - (3, B) (3, A), (2, A) each require 2 index bits.
  - (5, A) and (4, C) each require 3 index bits.
  - (9, C) requires 4 index bits.
- So, minimum total = 9 * 6 + 3 * 10 + 2 * 11 + 12 = 118 bits.
- And the compressed message would be: 0A1A0B1B11B101A100C0C0D1001C10A

Implementation Aspects

- For the purpose of implementation, preferably the code should be of fixed bit length or it should be possible to determine the length, if and when it changes.
- Incremental way of growing dictionary allows to control the width of the index part.
  - The expansion of width must follow a rule in creation of the dictionary.
  - Both the coder and the decoder should agree on it.
  - The encoder usually runs ahead of decoder.
  - Width is increased to $p + 1$ bits when the next empty slot in table is at the position 2^p (which requires $p + 1$ bits) for the new word.

Implementation Aspect

- So at the point of 2nd entry in our example, expand the code from 9 to 10 bits.
- At the point of 4th entry expand the code from 10 to 11 bits
- At the point of 8th entry expand the code from 11 to 12 bits
- This way the message will be coded as
  0A 01A 00B 001B 011B 101A 100C 0000C 0000D 1001C 0011A 0010A
- It requires 1*9 + 2*10 + 4*11 + 5*12 = 133 bits.
- In general, 2^k entries of length $(k+2)$+8 bits.

Encoder Algorithm

```
dictionary = null;
phrase w = null;
while (true) {
    wait for the next symbol v;
    if (w.v in dictionary)
        w = w.v;
    else {
        encode <index(w), v>;
        add w.v to dictionary;
        w = null;
    }
}
```