# Алгоритмы LZ77, LZSS, LZ78, LZW

В этом разделе рассматрвиаются алгоритмы семейства LZ, которыt в различных модификациях используется в программах-архиваторах. (По инициалиам их авторов Lempel, Ziv, Welch, Storer, Szimanski).

Суть алгоритма состоит в обнаружении во входном потоке повторяющихся цепочек байтов, составлении таблицы (словаря) таких обнаруженных цепочек и выдаче в выходной поток кодов (смещений в буфере или номеров строк словаря), соответствующих обнаруженной цепочке. Таким образом, если исходный файл имеет много повторяющихся последовательностей байтов, то такие последовательности будут заменены на некоторые ее числовой код. Важной особенностью алгоритмов является то, что распаковщик, анализируя сжатые данные в процессе работы, может построить копию исходной таблицы или буфера и, следовательно, ее не надо передавать вместе с сжатыми данными.

Опишем общую схему этих алгоритмов.

**LZ77**

На входной последовательности определяются два смежных участка — окно и буфер с заданной максимальной длиной. Окно предствалят собой “просмотренные и закодированные” символы последовательности, буфер — это текущая часть последовательнойсти для анализа. В окне ищется подстрока максимальной длины, совпадающая с фрагментом из начала буфера. Для такой строки выдается код

(смещение, длина, символ)

где смещение — смещение начала подстроки в окне относительно конца окна,

длина — количество символов найденной подстроки,

символ — символ, следующий в буфере за обнаруженным совпадающим фрагментом.

Если фрагмент не обнаружен (он односимвольный), то просто выводится код с этим символом и длиной 0 и смещением 0.

После этого окно и буфер сдвигаются вперед по последовательности на длину фрагмента + 1. (В первые моменты окно не сдвигается, а растет до заданного размера).

Декодирование происходит очевидным образом так как декодер аналогично строит просмотренную часть и имеет ссылки на фрагменты в ней.

При кодировании может использоваться идея от RLE — периодическая подпоследовательнойсть (длина $>$ смещения).

**Пример.** a b bbc ba bab cbac acb...

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

**LZSS**

Та же идея, что и LZ77, но другой код:

(смещение, длина) для найденных фрагментов

(0, символ) для дносимвольных

**Пример.** a b bb c b abab cba c a cb...

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

**LZ78**

Строится словарь. Код представляет собой номер слова в словаре и (следующий) символ. Словарь заполняется строками, которые соответствуют выведенному коду. Слово с номером 0 — пустое.

**Пример.** a b bb c bab ab cb ac acb...

словарь

| | слово | вход | код |
|---|---|---|---|
| 0 | | | |
| 1 | a | a | 0,a |
| 2 | b | b | 0,b |
| 3 | bb | b b | 2,b |
| 4 | c | c | 0,c |
| 5 | ba | b a | 2,a |
| 6 | bab | ba b | 5,b |
| 7 | ab | a b | 1,b |
| 8 | cb | c b | 4,b |
| 9 | ac | a c | 1,c |
| 10 | acb | ac b | 9,b |

**LZW**

*Кодирование.* Создается таблица, способная вместить достаточно большое количество строк. Первые 256 строк этой таблицы инициализируются всеми возможными односимвольными цепочками (т.е. соответствуют символам с кодами от 0 до 255). Дальнейший алгоритм выглядит так:

```
s = <пустая цепочка>;
while (есть символы во входном потоке) {
   c = NextSym();           // взять очередной символ
   if ( InTable (s+c) ) {   // цепочка s+c уже есть в таблице
      s = s+c;              // удлиняем цепочку прочитанным символом
   } else {                 // цепочки s+c нет в таблице
      OutCode (s);          // вывод кода, соответствующего цепочке s
      AddString (s+c);      // добавляем новую цепочку s+c в таблицу
      s = c;                // готовимся к обнаружению следующей цепочки
   }
}
OutCode (s);                // вывод кода для оставшейся цепочки s
```

Здесь знак “+” обозначает конкатенацию цепочек. Алгоритм кодирования каждый раз пытается найти в таблице наиболее длинную цепочку, соответствующую читаемой последовательности символов. Если это в какой-то момент не удается, то накопленная к этому времени цепочка заносится в таблицу. В какой-то момент может наступить переполнение таблицы. В этом случае кодировщик выводит в выходной поток специальный код очистки и таблица цепочек инициализируется заново. Обычно в реальных алгоритмах применяется таблица с 4096 входами для цепочек, при этом число 256 является кодом очистки (обозначим его CLC), а число 257 — кодом конца информации (EOI), эти строки таблицы не используется для цепочек. Заметим, что в этом случае для каждого выходного кода достаточно 12 бит. Поэтому при выводе целесообразно упаковывать каждые два кода в три байта. Для упрощения логики декодера обычно первым кодом сжатых данных является CLC, что сразу вызывает инициализацию таблицы цепочек.

**Пример.** abbbcbababcbacacb... алфавит {a,b,c,d}

словарь

| | слово | вход | код |
|---|---|---|---|
| 0 | a | | |
| 1 | b | | |
| 2 | c | | |
| 3 | d | | |
| 4 | ab | a | 0 |
| 5 | bb | b | 1 |
| 6 | bbc | bb | 5 |
| 7 | cb | c | 2 |
| 8 | ba | b | 1 |
| 9 | aba | ab | 4 |
| 10 | abc | ab | 4 |
| 11 | cba | cb | 7 |
| 12 | ac | a | 0 |
| 13 | ca | c | 2 |
| 14 | acb | ac | 12 |

*Декодирование.* Распаковка сжатых данных основывается на построении идентичной таблицы цепочек. Инициализация таблицы выполняется так же как и при кодировании.

```
while ( (k=NextCode()) != EOI ) {  // пока не конец информации
   if ( k == CLC ) {               // k есть код очистки
      InitTable();                 // заново инициализируем таблицу
      k = NextCode();              // читаем следующий код
      if ( k == EOI ) break;
      OutString(k);                // выводим цепочку для кода k
      old = k;                     // запоминаем текущий код
   } else {
      if ( InTable(k) )  {    // в таблице есть строка для кода k
         OutString(k);        // выводим цепочку для кода k
         AddString( String(old)+Char(k) );  // формируем и добавляем новую цепочку
         old = k;
      } else {                // в таблице нет строки для кода k
         s = String(old)+Char(old);  // формируем цепочку
         OutString(s);               // выводим цепочку
         AddString(s);               // и добавляем ее в таблицу
         old = k;
      }
   }
}
```

В этом алгоритме функция `String(i)` возвращает строку из таблицы, соответствующую коду `i`, а функция `Char(i)` возвращает только первый символ такой строки. Заметим, что функция `NextCode()` должна извлекать 12-битные коды из входного потока байтов.

Степень сжатия этого алгоритма оценить непросто. Но здесь применим все тот же вывод: сжатие будет хорошим, если входной поток байтов обладает свойствами

повторяемости отдельных фрагментов. С другой стороны, можно предложить такую последовательность входных байтов, что почти все они будут кодироваться односимвольными цепочками и алгоритм даст даже некоторый проигрыш в размере результата.

Заметим, что кодировщик и декодировщик алгоритма LZW заполяют таблицу цепочек по одинаковым правилам. Если кодировщик обработал $m$ входных байтов, а декодировщик раскодировал $m$ выходных байтов, то они имеют одинаковые состояния таблиц цепочек. Этот факт позволяет еще немного улучшить степень сжатия алгоритма.

Действительно, пока в таблице менее 512 цепочек, кодировщих может выдавать на выход 9-битные коды. Когда число зарегистрированных цепочек превысит 511 но еще будет меньше 1024, кодировщик может перейти на 10-битные коды. В диапазоне от 1024 до 2048 декодировщик выдает 11-битные коды, и только потом — 12-битные. Декодировщик сначала извлекает из входного потока 9-битные коды и аналогично переходит к более длинным кодам по мере разрастания своей таблицы цепочек.

В заключение заметим, что рассмотренные примерs алгоритмов сжатия используют различные подходы к определению информационной избыточности и различные методы для ее сокращения. Поэтому с практической точки зрения оправдано последовательное применение нескольких различных алгоритмов сжатия. Так, например, выходная кодовая последовательность алгоритма LZ может содержать много одинаковых кодов для некоторой часто встречающейся цепочки входных символов. В этом случае можно сжать сам код некоторым алгоритмом, ориентированным на анализ частотных характеристик появления отдельных байтов (Хаффмена либо арифметического кодирования). Для больших черно-белых изображений можно сначала использовать алгоритм RLE, а затем какой-либо другой алгорим.