# Самостоятельная работа 3. АЛГОРИТМЫ СЖАТИЯ ДАННЫХ. ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ

**Цель**: ознакомиться с принципами кодтрования информации для ее хранения и передачи; изучить основные виды и алгоритмы сжатия данных и научиться решать задачи сжатия данных по методу Хаффмана и LZW; получить представление о помехоустойчивом кодировании, рассмотреть коды Хэмминга, корректирующие 1 ошибку.

**Методические указания к выполнению работы.**

Все задания снабжены теоретическими сведениями и примерами решения задач.Этого, по-моему, достаточно для успешного выполнения задания. Дополнительные сведения можно получить из источников, указанных в списке литературы. Внимательно ознакомьтесь с теорией.

Всего в работе 3 задания. Задания выполняется в тетради, на титульном листе указываете ФИО студента, группу, вариант и текст задания. Вариант выбирается по вашему номеру в списке группы.

Критерии оценивания. За каждое задание максимум вы можете получить 3 балла (максимум в сумме 9 баллов) = 2 балла за задачу + 1 балл за ответы на контрольные вопросы. 2 балла – задача решена верно и правильно оформлена, 1 балл – в принципе задача решена, но имеются ошибки или неаккуратное оформление, 0 баллов – задача не решена или решена неверно.

Задание должно быть сдано в указанные сроки. При задержке сдачи работы более 2 недель оценка автоматически снижается на 2% за каждую неделю задержки.

# Теоретические сведения

[1. Cжатие данных 1](#_Toc142802088)

[1.1 Метод Хафмана. Оптимальное префиксное кодирование 3](#_Toc142802089)

[1.2 Алгоритм LZW 6](#_Toc142802090)

[2. ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ 14](#_Toc142802091)

[2,1. Помехоустойчивые коды и их основные параметры 14](#_Toc142802092)

[2.2 Код Хэмминга. Пример работы алгоритма 15](#_Toc142802093)

[Задание 1. Код Хаффмана 6](#_Toc142802414)

[Задание 2. Сжатие алгоритмом LZW 13](#_Toc142802415)

[Задание 3. Обнаружить и исправить ошибку в принятых данных с помощью кода Хэмминга (33 балла) 19](#_Toc142802416)

## 1. Cжатие данных

Основоположником науки о сжатии информации принято считать Клода Шеннона. Его теорема об оптимальном кодировании показывает, к чему нужно стремиться при кодировании информации и насколько та или иная информация при этом сожмется. Кроме того, им были проведены опыты по эмпирической оценке избыточности английского текста. Шеннон предлагал людям угадывать следующую букву и оценивал вероятность правильного угадывания. На основе ряда опытов он пришел к выводу, что количество информации в английском тексте колеблется в пределах 0,6 – 1,3 бита на символ. Несмотря на то, что результаты исследований Шеннона были по-настоящему востребованы лишь десятилетия спустя, трудно переоценить их значение.

1. [Код Хаффмана](#задание1)  (*34 балла*)
2. [Сжатие алгоритмом LZW](#задание2) (33 балла)
3. [Обнаружить и исправить ошибку в принятых данных с помощью кода Хэмминга](#Задание3) (33 балла)

**Сжатие данных** – *это процесс, обеспечивающий уменьшение объема данных путем сокращения их избыточности.*

Сжатие данных связано с компактным расположением порций данных стандартного размера. Сжатие данных можно разделить на два основных типа:

*Сжатие без потерь (полностью обратимое)* – это метод сжатия данных, при котором ранее закодированная порция данных восстанавливается после их распаковки полностью без внесения изменений. Для каждого типа данных, как правило, существуют свои оптимальные алгоритмы сжатия без потерь.

*Сжатие с потерями* – это метод сжатия данных, при котором для обеспечения максимальной степени сжатия исходного массива данных часть содержащихся в нем данных отбрасывается. Для текстовых, числовых и табличных данных использование программ, реализующих подобные методы сжатия, является неприемлемыми. В основном такие алгоритмы применяются для сжатия аудио- и видеоданных, статических изображений.

**Алгоритм сжатия данных (алгоритм архивации)** – это алгоритм, который устраняет избыточность записи данных.

Введем ряд определений, которые будут использоваться далее в изложении материала.

**Алфавит кода** – множество всех символов входного потока. При сжатии англоязычных текстов обычно используют множество из 128 ASCII кодов. При сжатии изображений множество значений пикселя может содержать 2, 16, 256 или другое количество элементов.

**Кодовый символ** – наименьшая единица данных, подлежащая сжатию. Обычно символ – это 1 байт, но он может быть битом, тритом {0,1,2}, или чем-либо еще.

**Кодовое слово** – это последовательность кодовых символов из алфавита кода. Если все слова имеют одинаковую длину (число символов), то такой код называется *равномерным (фиксированной длины)*, а если же допускаются слова разной длины, то – *неравномерным (переменной длины)*.

Код – полное множество слов.

**Токен** – единица данных, записываемая в сжатый поток некоторым алгоритмом сжатия. Токен состоит из нескольких полей фиксированной или переменной длины.

**Фраза** – фрагмент данных, помещаемый в словарь для дальнейшего использования в сжатии.

**Кодирование** – процесс сжатия данных.

**Декодирование** – обратный кодированию процесс, при котором осуществляется восстановление данных.

**Отношение сжатия** – одна из наиболее часто используемых величин для обозначения эффективности метода сжатия.

|  |  |
| --- | --- |
| \textit{Отношение сжатия} = \frac{\textit{размер выходного потока}}{\textit{размер входного потока}} | (1) |

Значение 0,6 означает, что данные занимают 60% от первоначального объема. Значения больше 1 означают, что выходной поток больше входного (отрицательное сжатие, или расширение).

**Коэффициент сжатия** – величина, обратная отношению сжатия.

|  |  |
| --- | --- |
| \textit{Коэффициент сжатия} = \frac{\textit{размер входного потока}}{\textit{размер выходного потока}} | (2) |

Значения больше 1 обозначают сжатие, а значения меньше 1 – расширение.

**Средняя длина кодового слова** – это величина, которая вычисляется как взвешенная вероятностями сумма длин всех кодовых слов.

|  |  |
| --- | --- |
| *L*cp=*p*1*L*1+*p*2*L*2+...+*p*n*L*n, | (3) |

где *p*1, *p*2, …, *p*n  – вероятности кодовых слов;

*L*1,*L*2,...,*L*n – длины кодовых слов.

Существуют два основных способа проведения сжатия.

***Статистические методы*** – методы сжатия, присваивающие коды переменной длины символам входного потока, причем более короткие коды присваиваются символам или группам символам, имеющим большую вероятность появления во входном потоке. Лучшие статистические методы применяют кодирование Хаффмана.

***Словарное сжатие***– это методы сжатия, хранящие фрагменты данных в "словаре" (некоторая структура данных). Если строка новых данных, поступающих на вход, идентична какому-либо фрагменту, уже находящемуся в словаре, в выходной поток помещается указатель на этот фрагмент. Лучшие словарные методы применяют метод Зива-Лемпела.

Рассмотрим несколько известных алгоритмов сжатия данных более подробно.

### 1.1 Метод Хафмана. Оптимальное префиксное кодирование

Этот алгоритм кодирования информации был предложен Д.А. Хаффманом в 1952 году. **Хаффмановское кодирование (сжатие)** – это широко используемый метод сжатия, присваивающий символам алфавита коды переменной длины, основываясь на вероятностях появления этих символов.

Идея алгоритма состоит в следующем: зная вероятности вхождения символов в исходный текст, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью присваиваются более короткие коды. Таким образом, в этом методе при сжатии данных каждому символу присваивается оптимальный префиксный код, основанный на вероятности его появления в тексте.

**Префиксный код** – это код, в котором никакое кодовое слово не является префиксом любого другого кодового слова. Эти коды имеют переменную длину.

**Оптимальный префиксный код** – это префиксный код, имеющий минимальную среднюю длину.

***Алгоритм Хаффмана*** можно разделить на два этапа.

Определение вероятности появления символов в исходном тексте.

Первоначально необходимо прочитать исходный текст полностью и подсчитать вероятности появления символов в нем (иногда подсчитывают, сколько раз встречается каждый символ). Если при этом учитываются все 256 символов, то не будет разницы в сжатии текстового или файла иного формата.

Нахождение оптимального префиксного кода.

Далее находятся два символа *a* и *b* с наименьшими вероятностями появления и заменяются одним фиктивным символом *x*, который имеет вероятность появления, равную сумме вероятностей появления символов *a* и *b*. Затем, используя эту процедуру рекурсивно, находится оптимальный префиксный код для меньшего множества символов (где символы *a* и *b* заменены одним символом *x*). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 или 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов.

*Пример 1*. Вероятности появления символов алфавита A={*a*,*b*,*c*,*d*,*e*} равны:

*pa*=0,3; *pb*=0,15; *pc*=0,2; *pd*=0,1; *pe*=0,25.

Наименьшие вероятности имеют *d* и *b*, заменяем их символом *x*∈{*d*,*b*}, вероятность *px*=0,1+0,15=0,25. Далее на множестве символов {*x*, *c*, *e*, *a*} снова находим пару с наименьшей вероятностью, это будет *c* и *e*, заменим ее символом *y*, вероятность появления которого *py*=*pc*+*pe*=0,2+0,22=0,42. Наконец, на множестве символов {*x*,*y*,*a*} находим пару с наименьшей вероятностью появления, это – символы *x* и *a*, заменяем их символом *z*, *pz*=*px*+*pa*=0,58. Результат показан на рисунке 1.

Осталось 2 символа: *y* и *z*, условимся символу с меньшей вероятностью приписывать префикс 0, а с большей 1. Код символа *y* начинается с 0, а символа *z* – с 1. Символ *y* заменяет символы *c* и *e*, первая цифра кода у обоих 0, вторая у символа *c* – 0, у символа *e* – 1. Коды для символа *z* и всех подчиненных символов: *x*, *d, b*,*a* строим аналогично. Получили следующую таблицу кодировки символов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | *d* | *b* | *c* | *e* | *a* |
| Вероятность | 0,1 | 0,15 | 0,2 | 0,25 | 0,3 |
| Код | 01 | 101 | 00 | 100 | 11 |

0

1

*pz*=0,58

*kz*=1…

*px*=0,25

*kx*=10…

*pa*=0,3

*ka*=11

*pd*=0,1

*kd*=100

*pb*=0,15

*kb*=101

*p*c=0,2

*kc*=00

*pe*=0,25

*ke*=01

0

0

0

1

1

*py*=0,42

*ky*=0…

0

0

1

Рисунок 1 – Построение кода Хаффмана

Сравним средний объем информации по Шеннону и среднюю длину кода сообщения:

*I* =

=0,33\*2+0,15\*3+0,2\*2+0,1\*3+0,22\*2=2,37

Видим, что в среднем объем данных близок к количеству информации в сообщении, но все-таки немного больше.

Сравним среднюю длину сообщения (в битах) при кодировании сообщения кодами постоянной длины и кодами Хафмана.

Для кодирования 5 символов алфавита достаточно ⎤log25⎡=3 бита. Длина сообщения, приходящаяся на 1 символ передаваемых данных постоянна и равна 3.

Для неравномерных кодов видим, что с вероятностью p2=0, 33+0,2+0,1=0,63 длина кода составит 2 бита, и с вероятностью p3=0,15+0,22=0,37 длина кода будет 3 бита. Соответственно средняя длина кода для передачи одного символа равна: 0,63\*2+0,37\*3=2,37 бита.

Коэффициент сжатия=3/2,37=1,27

Поскольку при построении дерева кода Хаффмана может возникнуть некоторый произвол, для выбора оптимального варианта кода используют дисперсию. Дисперсия показывает, насколько сильно отклоняются длины индивидуальных кодов от их средней величины. Лучшим будет код с наименьшей дисперсией.

Дисперсия рассчитывается по формуле:

Для полученного кода дисперсия будет равна

*D* = 0,33\*(2-2,37)2+0,15\*(3-2,37)2+0,2\*(2-2,37)2+0,1\*(3-2,37)2+0,22\*2-2,37)2≈1,2.

Для уменьшения дисперсии кода существует правило: *когда на дереве имеется более двух узлов с наименьшей вероятностью, следует объединять символы с наибольшей и наименьшей вероятностью*; это сокращает общую дисперсию кода.

Коды Хаффмана имеют уникальный префикс, что и позволяет однозначно их декодировать, несмотря на их переменную длину.

Закодируем текст *abbcdaae*, код сообщения: 1110110100100111101.

Начинаем раскодировать: Первая цифра 1, с 1 начинаются коды символов *a*, *b*, *d*. Смотрим вторую цифру, это 1. Коду 11 подходит только буква *a*.

Следующая цифра 1, аналогично, возможны буквы *a*, *b*, *d*. Следующая цифра 0. Коду 10.. удовлетворяют символы *b*, *d*. Следующая цифра 1 однозначно указывает на символ *b*, имеющий код 101.

Получаем строку *ab*…. Убедитесь самостоятельно, что расшифровка однозначна и дает исходную строку.

Сжатие получается за счет того, что символам, которые имеют большую вероятность, ставятся в соответствие коды меньшей длины.

Алгоритм Хаффмана универсальный, его можно применять для сжатия данных любых типов, но он малоэффективен для файлов маленьких размеров (за счет необходимости сохранения словаря). В настоящее время данный метод практически не применяется в чистом виде, обычно используется как один из этапов сжатия в более сложных схемах. Это единственный алгоритм, который не увеличивает размер исходных данных в худшем случае (если не считать необходимости хранить таблицу перекодировки вместе с файлом).

1. Код Хаффмана
2. В качестве исходной строки текста выбрать «Фамилия Имя Отчество» студента (с пробелами).
3. Сформировать алфавит фразы, посчитать количество вхождений символов и их вероятности появления.
4. Отсортировать алфавит в порядке убывания вероятности появления символов.
5. Построить дерево кодирования (см. рис. 1).
6. Упорядочить построенное дерево слева-направо (при необходимости). Присвоить ветвям коды. Определить коды символов (см. рис. 1).
7. Закодировать исходную строку.
8. Рассчитать коэффициенты сжатия относительно кодировки ASCII и относительно равномерного кода.
9. Рассчитать среднюю длину полученного кода и его дисперсию.

**Контрольные вопросы**

1. Порядок работы алгоритма Хаффмана.
2. Построение оптимального кодового дерева.
3. Средняя длина кода и ее расчет.
4. Дисперсия кода и ее расчет.
5. Является ли код Хаффмана оптимальным кодированием?
6. Является ли код Хаффмана алфавитным кодированием?
7. Как упорядочены вероятности при кодировании с помощью кода Хаффмана?
8. Что такое оптимальная длина кодирования?
9. Что такое «энтропия»?
10. Каковы вероятности в случае максимальной энтропии?
11. В каком случае количество информации равно нулю?
12. В каком случае кодирования сообщения избыточность кода будет сколь угодно близкой к нулю?

### 1.2 Алгоритм LZW

**Алгори́тм** Ле́мпеля — Зи́ва — Ве́лча — это универсальный **алгоритм** сжатия данных без потерь, созданный Авраамом Лемпелем, Яаковом Зивом и Терри Велчем. Он был опубликован Велчем в 1984 году.

#### Применение

Опубликование алгоритма LZW произвело большое впечатление на всех специалистов по сжатию информации. За этим последовало большое количество программ и приложений с различными вариантами этого метода.

Этот метод позволяет достичь одну из наилучших степеней сжатия среди других существующих методов сжатия графических данных, при полном отсутствии потерь или искажений в исходных файлах. В настоящее время используется в файлах формата TIFF, PDF, GIF, PostScript и других, а также отчасти во многих популярных программах сжатия данных (ZIP, ARJ, LHA).

#### Описание

Процесс сжатия выглядит следующим образом: последовательно считываются символы входного потока, и происходит проверка, существует ли в созданной таблице строк такая строка. Если такая строка существует, считывается следующий символ, а если строка не существует, в поток заносится код для предыдущей найденной строки, строка заносится в таблицу, а поиск начинается снова.

Например, если сжимают байтовые данные (текст), то строк в таблице окажется 256 (от "0"до "255"). Если используется 10-битный код, то под коды для строк остаются значения в диапазоне от 256 до 1023. Новые строки формируют таблицу последовательно, т. е. можно считать индекс строки ее кодом.

Для декодирования на вход подается только закодированный текст, поскольку алгоритм LZW может воссоздать соответствующую таблицу преобразования непосредственно по закодированному тексту. Алгоритм генерирует однозначно декодируемый код за счет того, что каждый раз, когда генерируется новый код, новая строка добавляется в таблицу строк. LZW постоянно проверяет, является ли строка уже известной, и, если так, выводит существующий код без генерации нового. Таким образом, каждая строка будет храниться в единственном экземпляре и иметь свой уникальный номер. Следовательно, при декодировании во время получения нового кода генерируется новая строка, а при получении уже известного, строка извлекается из словаря.

#### Алгоритм LZW

#### Кодирование

Начало.

**Шаг 1.** Все возможные символы заносятся в словарь. Во входную фразу Xзаносится первый символ сообщения.

**Шаг 2.** Считать очередной символ Yиз сообщения.

**Шаг 3.** Если Y– это символ конца сообщения, то выдать код для X, иначе:

Если фраза XYуже имеется в словаре, то присвоить входной фразе значение XYи перейти к **Шагу 2** ,

Иначе выдать код для входной фразы X, добавить XYв словарь и присвоить входной фразе значение Y. Перейти к **Шагу 2.**

Конец.

#### Декодирование

Начало.

**Шаг 1.** Все возможные символы заносятся в словарь. Во входную фразу Xзаносится первый код декодируемого сообщения.

**Шаг 2.** Считать очередной код Yиз сообщения.

**Шаг 3.** Если Y– это конец сообщения, то выдать символ, соответствующий коду X, иначе:

Если фразы под кодом XYнет в словаре, вывести фразу, соответствующую коду X, а фразу с кодом XYзанести в словарь.

Иначе присвоить входной фразе код XYи перейти к **Шагу 2** .

Конец.

#### Пример

Рассмотрим пример сжатия и декодирования сообщения. Сначала создадим начальный словарь единичных символов. В стандартной кодировке ASCII имеется 256 различных символов, поэтому, для того, чтобы все они были корректно закодированы (если нам неизвестно, какие символы будут присутствовать в исходном файле, а какие — нет), начальный размер кода будет равен 8 битам. Если нам заранее известно, что в исходном файле будет меньшее количество различных символов, то вполне разумно уменьшить количество бит. Чтобы инициализировать таблицу, мы установим соответствие кода 0соответствующему символу с битовым кодом 00000000, тогда 1 соответствует символу с кодом 00000001, и т.д., до кода 255.

Больше в таблице не будет других кодов, обладающих этим свойством.  
По мере роста словаря, размер групп должен расти, с тем чтобы учесть новые элементы. 8-битные группы дают 256 возможных комбинации бит, поэтому, когда в словаре появится 256-е слово, алгоритм должен перейти к 9-битным группам. При появлении 512-ого слова произойдет переход к 10-битным группам, что дает возможность запоминать уже 1024слова и т.д.

В нашем примере алгоритму заранее известно о том, что будет использоваться всего 5 различных символов, следовательно, для их хранения будет использоваться минимальное количество бит, позволяющее нам их запомнить, то есть 3 (8 различных комбинаций).

Таблица 1

Исходная кодовая таблица

|  |  |  |
| --- | --- | --- |
| Символ | Битовый код | Код |
| *a* | 000 | 0 |
| *b* | 001 | 1 |
| *c* | 010 | 2 |
| *d* | 011 | 3 |
| *e* | 100 | 4 |

Эту начальную кодировку символов необходимо передать вначале сообщения, чтобы его можно было раскодировать.

#### Кодирование

Пусть мы сжимаем последовательность abacabadabacabae.

**Шаг 1:** Тогда, согласно изложенному выше алгоритму, мы добавим к изначально пустой строке a и проверим, есть ли строка aв таблице. Поскольку мы при инициализации занесли в таблицу все строки из одного символа, то строка a есть в таблице.

**Шаг 2:** Далее мы читаем следующий символ b из входного потока и проверяем, есть ли строка ab в таблице. Такой строки в таблице пока нет.

Добавляем в таблицу ab (код 5). В выходной поток выдаем 000 – код символа *a*;

**Шаг 3:** ba – нет. В таблицу: ba (код 6). В поток: 001 – код *b*;

**Шаг 4:** ac **–** нет. В таблицу: ac(код 7). В поток: 000 – код *a*;

**Шаг 5:** ca – нет. В таблицу: ca (код 8). В поток: 010 – код *c*; Начиная с этого момента в поток вместо 3-битовых кодов выводим 4-х битовые

**Шаг 6:** ab – есть в таблице; aba— нет. В таблицу: aba(код 9). В поток: 0101 - *ab*;

**Шаг 7:** ad – нет. В таблицу: ad (код 10). В поток: 0000 – код *a*;

**Шаг 8:** da – нет. В таблицу: da(код 11). В поток: 0011 – код *d*;

**Шаг 9:** aba – есть в таблице; abac— нет. В таблицу: abac(код 12). В поток: 1001 – *aba*;

**Шаг 10:** ca – есть в таблице; cab – нет. В таблицу: cab (код 13). В поток: 1000 – *ca*;

**Шаг 11:** ba– есть в таблице; bae— нет. В таблицу: bae (код 14). В поток: 0110 – *ba*;

обратите внимание, что, начиная с шага 6 коды передаются 4-мя битами вместо 3-х.

**Шаг 12:** И, наконец последняя строка e, за ней идет конец сообщения, поэтому мы просто выводим в поток 0100 – *e*.

Таблица 2

Процесс кодирования сообщения

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Текущая строка | Текущий символ | Следующий символ | Вывод (поток) | | | Словарь | |
| Символ | Код | Биты |
| *ab* | *a* | *b* | *a* | 0 | 000 | 5: | *ab* |
| ba | b | a | *b* | 1 | 001 | 6: | *ba* |
| ac | a | c | *a* | 0 | 000 | 7: | *ac* |
| ca | c | a | *c* | 2 | 010 | 8: | *ca* |
| ab | a | b |  | - | - | - | *-* |
| aba | b | a | *ab* | 5 | 0101 | 9: | *aba* |
| ad | a | d | *a* | 0 | 0000 | 10: | *ad* |
| da | d | a | *d* | 3 | 0011 | 11: | *da* |
| ab | a | b |  | - | - | - | *-* |
| aba | b | a |  | - | - | - | *-* |
| abac | a | c | *aba* | 9 | 1001 | 12: | *abac* |
| ca | c | a |  | - | - | - | *-* |
| cab | a | b | *ca* | 8 | 1000 | 13: | *cab* |
| ba | b | a |  | - | - | - | *-* |
| bae | a | e | *ba* | 6 | 0110 | 14: | *bae* |
| e | e | - | *e* | 4 | 0100 | - | *-* |

Итак, мы получаем закодированное сообщение 0 1 0 2 5 0 3 9 8 6 4 и его битовый эквивалент 000 001 010 0101 0000 0011 1001 1000 0110 0100. Его объем – 40 бит.

Каждый символ исходного сообщения был закодирован группой из трех бит, сообщение содержало 16 символов, следовательно, длина исходного сообщения составляла 3 \cdot 16 = 48бит.

Закодированное же сообщение так же сначала кодировалось трехбитными группами, а при появлении в словаре восьмого слова — четырехбитными, итого длина сообщения составила 4 \cdot 3 + 7 \cdot 4 = 40 бит, что на 8 бит короче исходного.

Коэффициент сжатия = 48/40=1,2

#### Декодирование

Особенность LZW заключается в том, что для декомпрессии нам не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только входным потоком кодов.

Теперь представим, что мы получили закодированное сообщение, приведённое выше, и нам нужно его декодировать. Прежде всего нам нужно знать начальный словарь, а последующие записи словаря мы можем реконструировать уже на ходу, поскольку они являются просто конкатенацией предыдущих записей. Кроме того, в процессе кодировании и декодировании коды в словарь добавляются во время обработки одного и того же символа, т.е. это происходит “синхронно”.

Таблица 3

Декодирование сообщения

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Данные | | На выходе | Новая запись | | | |
| Биты | Код | Полная | | Частичная | |
| 000 | 0 | *a* | - | - | 5: | *a*? |
| 001 | 1 | *b* | 5: | *ab* | 6: | *b*? |
| 000 | 0 | *a* | 6: | *ba* | 7: | *a*? |
| 010 | 2 | *c* | 7: | *ac* | 8: | *c*? |
| 0101 | 5 | *ab* | 8: | *ca* | 9: | *ab*? |
| 0000 | 0 | *a* | 9: | *aba* | 10: | *a*? |
| 0011 | 3 | *d* | 10: | *ad* | 11: | *d*? |
| 1001 | 9 | *aba* | 11: | *da* | 12: | *aba*? |
| 1000 | 8 | *ca* | 12: | *abac* | 13: | *ca*? |
| 0110 | 6 | *ba* | 13: | *cab* | 14: | *ba*? |
| 0100 | 4 | *e* | 14: | *bae* | - | - |

#### Примечание

Для повышения степени сжатия изображений данным методом часто используется одна “хитрость” реализации этого алгоритма. Некоторые файлы, подвергаемые сжатию с помощью LZW, имеют часто встречающиеся цепочки одинаковых символов, например aaaaaaaaaaaaa...или 303030… и т. п. Их непосредственное сжатие будет генерировать выходной код 005000600007.... Спрашивается, можно ли в этом частном случае повысить степень сжатия?

Оказывается, это возможно, если оговорить некоторые действия:

Мы знаем, что для каждого кода надо добавлять в таблицу строку, состоящую из уже присутствующей там строки и символа, с которого начинается следующая строка в потоке.

Пусть словарь состоит из слов : a, b, c, d, e. Будем кодировать стоку aaaaaaaaaa

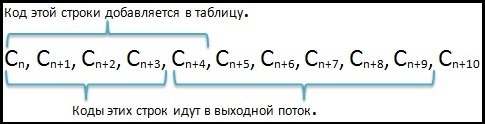
Итак, кодировщик заносит первую a в строку, ищет и находит a

в словаре под номером \langle0\rangle. Добавляет в строку следующую a, находит, что aaнет в словаре. Тогда он добавляет запись \langle5\rangle: aaв словарь и выводит метку \langle0\rangle(a) в выходной поток.

Далее строка инициализируется второй a, то есть принимает вид a?вводится третья a, строка вновь равна aa, которая теперь имеется в словаре.

Если появляется четвертая a, то строка aa?равна aaa, которой нет в словаре. Словарь пополняется этой строкой, а на выход идет метка \langle5\rangle(aa).

После этого строка инициализируется третьей a, и т.д. и т.п. Дальнейший процесс вполне ясен.

[](http://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:LZW-img.jpg)

|  |  |
| --- | --- |
| Слово | Номер в словаре |
| *a* | \langle0\rangle |
| *b* | \langle1\rangle |
| *c* | \langle2\rangle |
| *d* | \langle3\rangle |
| *e* | \langle4\rangle |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Текущая строка | Текущий символ | Следующий символ | Вывод | | Словарь | |
| Код | Биты |
| *aa* | *a* | *a* | 0 | 000 | 5: | *aa* |
| *aa* | *a* | *a* | - | - | - | *-* |
| *aaa* | *a* | *a* | 5 | 101 | 6: | *aaa* |
| *a* | *a* | *a* | - | - | - | *-* |
| *aa* | *a* | *a* | - | - | - | *-* |
| *aaa* | *a* | *a* | - | - | - | *-* |
| *aaaa* | *a* | *a* | 6 | 110 | 7: | *aaaa* |
| *a* | *a* | *a* | - | - | - | *-* |
| *aa* | *a* | *a* | - | - | - | *-* |
| *aaa* | *a* | *a* | - | - | - | *-* |
| *aaaa* | *a* | *a* | 7 | 111 | 8: | *aaaaa* |

В результате на выходе получаем последовательность 0567. При кодировании использовались только трехбитные группы. Длина закодированного сообщения составила 4 \cdot 3 = 12бит, что на 7 \cdot 3  - 12 = 9 бит короче кодирования стандартным методом LZW. Можно показать, что такая последовательность будет корректно восстановлена. Декодировщик сначала читает первый код – это \langle0\rangle, которому соответствует символ a. Затем читает код \langle5\rangle, но этого кода в его таблице нет. Но мы уже знаем, что такая ситуация возможна только в том случае, когда добавляемый символ равен первому символу только что считанной последовательности, то есть a. Поэтому он добавит в свою таблицу строку aa с кодом \langle5\rangle, а в выходной поток поместит aa. И так может быть раскодирована вся цепочка кодов.

Мало того, описанное выше правило кодирования мы можем применять в общем случае не только к подряд идущим одинаковым символам, но и к последовательностям, у которых очередной добавляемый символ равен первому символу цепочки.

#### Преимущества алгоритма LZW

Алгоритм является однопроходным.

Для декомпрессии не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.

#### Недостатки алгоритма LZW

Алгоритм не проводит анализ входных данных.

1. Сжатие алгоритмом LZW
2. По алгоритму LZW выполнить сжатие сообщения, которое задано в табл. 4. (таблица кодировки – ASCII). Процесс сжатия оформить в виде табл. 1 и 2. В таблице 1 показать только коды используемых символов.
3. Декодировать сжатое в п. 1 сообщение. Процесс декодирования оформить в виде табл. 3.
4. Посчитать коэффициент сжатия

Таблица 4

Варианты заданий

|  |  |  |  |
| --- | --- | --- | --- |
| № | Текст | № | Текст |
| 1 | ФЯ1241234ЙЦЙФЯ | 17 | HDREEREDREF |
| 2 | WASH2ASZXCWAS | 18 | KJYUJKJYUJKY |
| 3 | RTEY421421TEITY | 19 | BGHTBGHTBGH |
| 4 | FGWQ767FGWQRE | 20 | JYRTYTYRTYT |
| 5 | EDTTW3366ETTED | 21 | REDEDRETYRE |
| 6 | OIRT67767OIRT63 | 22 | GRBYGBYGTR |
| 7 | HJWEFV44WEFJW | 23 | АПРКПРКЕНТПР |
| 8 | KJDFFKDFF563332 | 24 | ОЛНЕОНЕГРНЕО |
| 9 | IERRHWERRHWRE | 25 | РАОПВАОРАОПВ |
| 10 | UYIIERIIERWDFG | 26 | ПРЕДЛЕДПРЕДЛ |
| 11 | IFDEFGDSEFGDK | 27 | ВКАПКАПВАПК |
| 12 | DEVVDEVTYDF | 28 | ЛЕКЫЕКВЕКЫВ |
| 13 | KGJVFJVFHTGN | 29 | МИНТВВАМИТВ |
| 14 | PODFRRRTRRB | 30 | ЛГОУВАОУВЛГ |
| 15 | FNGNNHTRHTR | 31 | ДЫЦАТДЦАТДЦ |
| 16 | DFFGFFFGFFFG | 32 | 23843384723338 |

**Контрольные вопросы**

1. В чем состоит принцип LZW-сжатия?
2. Какие форматы файлов наиболее эффективно архивировать с помощью LZW-сжатия?
3. В чем особенность процесса распаковки при использовании алгоритма LZW?
4. Какими достоинствами и недостатками обладает алгоритм LZW?
5. Что называется эффективным кодированием или сжатием сообщения?
6. Назовите причины избыточности сообщений.
7. Дайте определение коэффициенту сжатия?
8. Чему равен максимальный коэффициент сжатия в случае сжатия без потерь информации?
9. Назовите известные алгоритмы сжатия сообщений без потерь информации.
10. Опишите процесс сжатия сообщения по алгоритму LZW.
11. Опишите процесс декодирования сообщения по алгоритму LZW.

## 2. ПОМЕХОУСТОЙЧИВОЕ КОДИРОВАНИЕ

### 2,1. Помехоустойчивые коды и их основные параметры

Проблема повышения верности обусловлена не соответствием между требованиями, предъявляемыми при передаче данных и качеством реальных каналов связи. В сетях передачи данных требуется обеспечить верность не хуже 10-6 - 10-9, а при использовании реальных каналов связи и простого (первичного) кода указанная верность не превышает 10-2 - 10-5.   
Одним из путей решения задачи повышения верности в настоящее время является использование специальных процедур, основанных на применении помехоустойчивых (корректирующих) кодов.

#### Принцип построения помехоустойчивых кодов

Простые коды характеризуются тем, что для передачи информации используются все кодовые слова (комбинации), количество которых равно N=qn (q - основание кода, а n - длина кода). В общем случае они могут отличаться друг от друга одним символом (элементом). Поэтому даже один ошибочно принятый символ приводит к замене одного кодового слова другим и, следовательно, к неправильному приему сообщения в целом.   
*Помехоустойчивыми называются коды, позволяющие обнаруживать и (или) исправлять ошибки в кодовых словах, которые возникают при передаче по* каналам связи. Эти коды строятся таким образом, что для передачи сообщения используется лишь часть кодовых слов, которые отличаются друг от друга более чем в одном символе. Эти кодовые слова называются разрешенными. Все остальные кодовые слова не используются и относятся к числу запрещенных.   
Применение помехоустойчивых кодов для повышения верности передачи данных связанно с решением задач кодирования и декодирования.   
Задача кодирования заключается в получении при передаче для каждой k - элементной комбинации из множества qk соответствующего ей кодового слова длиною n из множества qn.

### 2.2 Код Хэмминга. Пример работы алгоритма

Прежде всего стоит сказать, что такое Код Хэмминга и для чего он, собственно, нужен.

Коды Хэмминга — наиболее известные и, вероятно, первые из самоконтролирующихся и самокорректирующихся кодов. Построены они применительно к двоичной системе счисления.

Другими словами, это алгоритм, который позволяет закодировать какое-либо информационное сообщение определённым образом и после передачи (например, по сети) определить появилась ли какая-то ошибка в этом сообщении (к примеру, из-за помех) и, при возможности, восстановить это сообщение. Рассмотрим самый простой алгоритм Хемминга, который может исправлять лишь одну ошибку.

Также стоит отметить, что существуют более совершенные модификации данного алгоритма, которые позволяют обнаруживать (и если возможно исправлять) большее количество ошибок.

Сразу стоит сказать, что Код Хэмминга состоит из двух частей. Первая часть кодирует исходное сообщение, вставляя в него в определённых местах контрольные биты (вычисленные особым образом). Вторая часть получает входящее сообщение и заново вычисляет контрольные биты (по тому же алгоритму, что и первая часть). Если все вновь вычисленные контрольные биты совпадают с полученными, то сообщение получено без ошибок. В противном случае, выводится сообщение об ошибке и при возможности ошибка исправляется.

#### Как это работает.

Для того, чтобы понять работу данного алгоритма, рассмотрим пример.

#### Подготовка

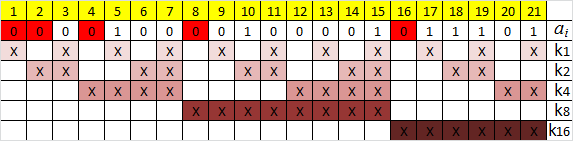
Допустим, у нас есть сообщение «habr», которое необходимо передать без ошибок. Для этого сначала нужно наше сообщение закодировать при помощи кода Хэмминга. Нам необходимо представить его в бинарном виде.  
  


На этом этапе стоит определиться с, так называемой, длиной информационного слова, то есть длиной строки из нулей и единиц, которые мы будем кодировать. Допустим, у нас длина слова будет равна 16. Таким образом, нам необходимо разделить наше исходное сообщение («habr») на блоки по 16 бит, которые мы будем потом кодировать отдельно друг от друга. Так как один символ занимает в памяти 8 бит, то в одно кодируемое слово помещается ровно два ASCII символа. Итак, мы получили две бинарные строки по 16 бит:   
  
https://habrastorage.org/storage2/7e3/8f2/0fe/7e38f20fe4ac07f834f5a68ec775a829.pngи https://habrastorage.org/storage2/917/51d/2a9/91751d2a972767500f78b7700c9e21dc.png

После этого процесс кодирования распараллеливается, и две части сообщения («ha» и «br») кодируются независимо друг от друга. Рассмотрим, как это делается на примере первой части.  
Прежде всего, необходимо вставить контрольные биты. Они вставляются в строго определённых местах — это позиции с номерами, равными степеням двойки. В нашем случае (при длине информационного слова в 16 бит) это будут позиции 1, 2, 4, 8, 16. Соответственно, у нас получилось 5 контрольных бит (выделены красным цветом):  
  
Было:  
https://habrastorage.org/storage2/7e3/8f2/0fe/7e38f20fe4ac07f834f5a68ec775a829.png  
  
Стало:  
https://habrastorage.org/storage2/14f/13a/b50/14f13ab5014ce60d2a6a04275d144331.png  
  
Таким образом, длина всего сообщения увеличилась на 5 бит. До вычисления самих контрольных бит, мы присвоили им значение «0».

#### Вычисление контрольных бит.

Теперь необходимо вычислить значение каждого контрольного бита. Значение каждого контрольного бита зависит от значений информационных бит (как неожиданно), но не от всех, а только от тех, которые этот контрольных бит контролирует. Для того, чтобы понять, за какие биты отвечает каждых контрольный бит необходимо понять очень простую закономерность: контрольный бит с номером N контролирует все последующие N бит через каждые N бит, начиная с позиции N. Не очень понятно, но по картинке, думаю, станет яснее:



Здесь знаком «X» обозначены те биты, которые контролирует контрольный бит, номер которого справа. То есть, к примеру, бит номер 12 контролируется битами с номерами 4 и 8. Ясно, что чтобы узнать какими битами контролируется бит с номером N надо просто разложить N по степеням двойки.

Но как же вычислить значение каждого контрольного бита? Делается это очень просто: берём каждый контрольный бит и смотрим сколько среди контролируемых им битов единиц, получаем некоторое целое число и, если оно чётное, то ставим ноль, в противном случае ставим единицу. Вот и всё! Можно конечно и наоборот, если число чётное, то ставим единицу, в противном случае, ставим 0. Главное, чтобы в «кодирующей» и «декодирующей» частях алгоритм был одинаков. (Мы будем применять первый вариант).

В нашем примере:

*a*1= (*a*3+*a*5+*a*7+*a*9+*a*11+*a*13+*a*15+*a*17+*a*19+*a*21) mod 2 = (0+0+1+0+0+0+0+1+1+1+1) mod 2 = 5 mod 2 = **1**;

контрольная сумма k1

k1=(*a*1+*a*3+*a*5+*a*7+*a*9+*a*11+*a*13+*a*15+*a*17+*a*19+*a*21) mod 2 = 6 mod 2 = 0;

*a*2=(*a*3+*a*6+*a*7+*a*10+*a*11+*a*114+*a*15+*a*18+*a*19) mod 2 = 4 mod 2 = **0**;

контрольная сумма k2

k2=(*a*2+*a*3+*a*6+*a*7+*a*10+*a*11+*a*114+*a*15+*a*18+*a*19) mod 2 = 4 mod 2 = 0;

*a*4= (*a*5+ *a*6+*a*7+*a*12+*a*13+*a*14+*a*15+*a*20+*a*21) mod 2 = 3 mod 2 = **1**;

контрольная сумма k4

k4= (*a*4+*a*5+ *a*6+*a*7+*a*12+*a*13+*a*14+*a*15+*a*20+*a*21) mod 2 = 4 mod 2 = 0;

*a*8 = (*a*9+ *a*10+*a*11+*a*12+*a*13+*a*14+*a*15) mod 2= 2 mod 2 = **0**;

контрольная сумма k8

k8 = (*a*8+*a*9+ *a*10+*a*11+*a*12+*a*13+*a*14+*a*15) mod 2= 2 mod 2 = 0;

*a*16 = (*a*17+*a*18+*a*19+*a*20+*a*21) mod 2 = 4 mod 2 = **0**;

контрольная сумма k16

k16=(*a*16+*a*17+*a*18+*a*19+*a*20+*a*21) mod 2 = 4 mod 2 = 0;

здесь операция x mod 2 означает целочисленный остаток от деления х/2. Обратите внимание, что все контрольные суммы равны нулю: k1=k2=k4=k8=k16=0.

Высчитав контрольные биты для нашего информационного слова получаем следующее:   
https://habrastorage.org/storage2/719/e5c/30a/719e5c30a55f74f58db64960cbe01191.png  
и для второй части:   
https://habrastorage.org/storage2/f4c/b23/f09/f4cb23f09724cc292f3de4b4730d9650.png  
  
Вот и всё! Первая часть алгоритма завершена.

#### Декодирование и исправление ошибок.

Теперь, допустим, мы получили закодированное первой частью алгоритма сообщение, но оно пришло к нам с ошибкой. К примеру, 11-ый бит передался неправильно:

Исходное сообщение: Полученное сообщение:  
https://habrastorage.org/storage2/719/e5c/30a/719e5c30a55f74f58db64960cbe01191.png https://habrastorage.org/storage2/e14/9f0/22e/e149f022e390deae045650c6e17c03ba.png

Вся вторая часть алгоритма заключается в том, что необходимо заново вычислить все контрольные суммы (так же, как и в первой части) и сравнить их с 0. Так, посчитав контрольные биты полученного сообщения с неправильным 11-ым битом мы получим такую картину:

k1= *a*1+*a*3+*a*5+*a*7+*a*9+*a*11+*a*13+*a*15+*a*17+*a*19+*a*21) mod 2 = (1+0+1+0+0+1+0+1+1+1+1) mod 2 = 7 mod 2 = 1; (ошибка)

k2=(*a*2+*a*3+*a*6+*a*7+*a*10+*a*11+*a*114+*a*15+*a*18+*a*19) mod 2 = 5 mod 2 = 1; (ошибка)

k4= (*a*4+*a*5+ *a*6+*a*7+*a*12+*a*13+*a*14+*a*15+*a*20+*a*21) mod 2 = 3 mod 2 = 0;

k8= (*a*8+*a*9+ *a*10+*a*11+*a*12+*a*13+*a*14+*a*15) mod 2= 3 mod 2 = 1; (ошибка)

k16=(*a*16+*a*17+*a*18+*a*19+*a*20+*a*21) mod 2 = 4 mod 2 = 0;

Как мы видим, контрольные биты под номерами: 1, 2, 8 не совпадают с 0. Теперь просто сложив номера позиций неправильных контрольных бит (1 + 2 + 8 = 11) мы получаем позицию ошибочного бита. Теперь просто инвертировав его и отбросив контрольные биты, мы получим исходное сообщение в первозданном виде! Абсолютно аналогично поступаем со второй частью сообщения.

#### Обнаружение 2-х ошибок.

Рассмотренный выше алгоритм позволяет исправлять только одну ошибку. Рассмотрим, что будет, если в сообщении будет сразу 2 ошибки (11-ый и 7-ой биты переданы неправильно):

Исходное сообщение: Полученное сообщение:  
https://habrastorage.org/storage2/719/e5c/30a/719e5c30a55f74f58db64960cbe01191.png 

Проверим контрольные биты полученного сообщения:

k1= *a*1+*a*3+*a*5+*a*7+*a*9+*a*11+*a*13+*a*15+*a*17+*a*19+*a*21) mod 2 = (1+0+1+1+0+1+0+1+1+1+1) mod 2 = 8 mod 2 = 0;

k2=(*a*2+*a*3+*a*6+*a*7+*a*10+*a*11+*a*114+*a*15+*a*18+*a*19) mod 2 = 6 mod 2 = 0;

k4= (*a*4+*a*5+ *a*6+*a*7+*a*12+*a*13+*a*14+*a*15+*a*20+*a*21) mod 2 = 5 mod 2 = 1; (ошибка)

k8= (*a*8+*a*9+ *a*10+*a*11+*a*12+*a*13+*a*14+*a*15) mod 2= 3 mod 2 = 1; (ошибка)

k16=(*a*16+*a*17+*a*18+*a*19+*a*20+*a*21) mod 2 = 4 mod 2 = 0;

Неверны контрольные биты k4 и k8. Сложим номера позиций неправильных контрольных битов (4 + 8 = 12), но в 12 бите нет ошибки. Что произошло?

7-ой бит контролируется разрядами (1, 2, 4); 11-ый бит – разрядами (1, 2, 8). В контрольных разрядах 1 и 2 сложились обе ошибки, а так как вычисление контрольных сумм выполняется по модулю 2, то ошибка не зафиксировалась.

Чтобы зафиксировать (но не исправить) появление 2-х ошибок, добавим еще один контрольный бит *a*0=, который контролирует все разряды. Вычислив контрольную сумму k0=, можно обнаружить 1 или 2 ошибки и исправить 1:

1. если все контрольные суммы k0=k1=k2=k4=k8=k16=0 – ошибки нет;
2. если k0=1, то обнаружена одна ошибка и ее положение в коде определяется контрольными битами k1, k2, k4, k8, k16;
3. если k0=0, а какие-либо из контрольных сумм k1, k2, k4, k8, k16 отличны от нуля, то фиксируется двойная ошибка, которая не может быть скорректирована. В этом случае сообщение нужно передать заново.

#### Заключение.

В данном примере, мы взяли длину информационного сообщения именно 16 бит, так как полагали, что она наиболее оптимальна для рассмотрения примера (не слишком длинная и не слишком короткая), но конечно же длину можно взять любую. Только стоит учитывать, что в данной простой версии алгоритма на одно информационное слово можно исправить только одну ошибку.

Можно построить коды, исправляющие 2, 3 и более ошибок (см. список литературы)

1. Обнаружить и исправить ошибку в принятых данных с помощью кода Хэмминга (33 балла)

Расчетным путем определите, в каком разряде принятых данных произошло искажение. Исходные данные для разных вариантов приведены в таблице 4. Исправьте ошибку, если она есть. Процесс вычисления искаженного бита следует подробно описать в отчете.

Таблица 4

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Вар | *k1* | *k2* | *b1* | *k4* | *b2* | *b3* | *b4* | *k8* | *b5* | *b6* | *b7* | *b8* |
| бит | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 3 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 6 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 8 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 11 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 12 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 13 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 14 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 15 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 17 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 18 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 19 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 20 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 21 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 22 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 23 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 25 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 26 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 27 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 28 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 29 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 30 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 31 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 32 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

**Контрольные вопросы**

1. Можно ли исправить возникшую ошибку в передаваемых данных, если для помехоустойчивого кодирования используется лишь единственный бит паритета?
2. Запишите формулы для формирования контрольных битов с помощью кода Хэмминга.
3. Для чего используется QR код?
4. Как рассчитывается слово синдрома в коде Хэмминга?
5. К чему приведет искажение контрольного бита при использовании кода Хэмминга?
6. Нужно ли исправлять контрольные биты?
7. Почему коды БЧХ называют избыточными?
8. Как проверить, правильно ли сформирован код БЧХ на передающей стороне?
9. Что такое вес остатка?

**Список литературы**

* 1. Баранова Е.К. Эффективное кодирование и защита информации: текст лекций для студентов специальности 510200. М.: МГУЛ, 2002.
  2. Вернер М Основы кодирования: учебник для вузов. М.: Техносфера, 2006.
  3. Крушный В.В. Основы теории информации и кодирования. Снежинск: СГФТА, 2005.
  4. Морелос-Сарагоса Р. Искусство помехоустойчивого кодирования. Методы, алгоритмы, применение. М.: Техносфера, 2006.
  5. Филонеяков А.И., Самсонов Б.Б., Кречет т.н., Плохов Е.М. Теория информации и кодирования М.: ФЕНИКС, 2002.
  6. Хохлов Г.И. Основы теории информации: учеб. пособие. М.: Академия, 2008.
  7. Шеннон К. Математическая теория связи. Работы по теории информации и кибернетике: пер. с англ.; под ред. Р.Л. Добрушина и О.Б. Лупанова. М.: ИЛ, 1963.