В. О. Дженжер, Л. В. Денисова
Оренбургский государственный педагогический университет, Оренбург, Россия

РЕАЛИЗАЦИЯ КОДА ХЕММИНГА НА PASCALABC.NET ПРИ ИЗУЧЕНИИ ТЕОРЕТИЧЕСКИХ ОСНОВ ИНФОРМАТИКИ*

Аннотация
Теоретические основы информатики — классический раздел дискретной математики, преподаваемый студентам различных информационных, математических и технических направлений. Изложение материала в основном ведется с использованием матричной алгебры. В настоящей статье описывается методика преподавания темы помехоустойчивого кодирования студентам педвуза в рамках курса теоретических основ информатики и рассмотрен алгоритм получения кода Хемминга через табличный шаблон. Предлагается оригинальная реализация рассмотренного алгоритма на PascalABC.NET. При написании кода используются современные приемы и техники: динамические массивы, срезы, безопасные срезы, условная (тернарная) операция, цикл foreach, методы последовательностей, лямбда-выражения, кортежи, документирующие комментарии и др. Для работы с двоичными числами используются функции модуля School, входящего в официальную поставку компилятора PascalABC.NET. Описанная методика апробирована в преподавании студентам 4-го курса педагогического направления, профиль «Математика и информатика», Оренбургского государственного педагогического университета и показала свою применимость. Кроме того, возможно использование предлагаемого подхода для обучения школьников на профильном уровне или во внеурочной работе.

Ключевые слова: теоретические основы информатики, помехоустойчивое кодирование, код Хемминга, программирование, PascalABC.NET.
DOI: 10.32517/2221-1993-2021-20-9-27-36

1. Введение

Теоретические основы информатики — классический раздел дискретной математики, преподаваемый студентам различных информационных, математических и технических направлений. Изложение материала в основном ведется с использованием матричной алгебры. При этом, как правило, от студентов требуется реализация некоторых из изучаемых алгоритмов на практике.

Эта же тема присутствует и в школьных учебниках информатики для базового и профильного/углубленного уровня обучения (см., например, [1, 3, 6, 8]), а поэтому включена и в образовательную программу бакалавра педагогического направления физико-математического профиля. Очевидно, что преподавать эту дисциплину будущим учителям информатики нужно несколько иначе, чем студентам-программистам. Помимо чисто теоретического и фундаментального компонента следует иметь в виду возможность применения этого знания именно в школьной практике учителя-информатика. Под этим мы понимаем умение реализовать изучаемый алгоритм наиболее простыми способами и средствами, доступными заинтересованному старшекласснику.

Рассмотрим этот подход на примере изучения кода Хемминга.

* Материалы к статье можно скачать на сайте ИНФО: http://infojournal.ru/journals/school/school_09-2021/

Контактная информация

Дженжер Вадим Олегович, канд. физ.-мат. наук, доцент, зав. кафедрой информатики, физики и методики преподавания информатики и физики, Оренбургский государственный педагогический университет, Оренбург, Россия; *адрес:* 460014, Россия, г. Оренбург, ул. Советская, д. 19; *e-mail:* vdjenjer@yandex.ru

Денисова Людмила Викторовна, канд. пед. наук, доцент кафедры информатики, физики и методики преподавания информатики и физики, Оренбургский государственный педагогический университет, Оренбург, Россия; *адрес:* 460014, Россия, г. Оренбург, ул. Советская, д. 19; *e-mail:* lv-denisova@yandex.ru

V. O. Dzhenzher, L. V. Denisova
Orenburg State Pedagogical University, Orenburg, Russia

IMPLEMENTATION OF THE HAMMING CODE ON PASCALABC.NET WHILE STUDYING THE THEORETICAL FOUNDATIONS OF INFORMATICS

Abstract
Theoretical Foundations of Informatics is a classic branch of discrete mathematics taught to students in various information, mathematical and technical fields. The presentation of the material is mainly carried out using matrix algebra. This article describes a methodology for teaching the topic of error correcting coding to pedagogical university students as part of the course on theoretical foundations of informatics and considers an algorithm for obtaining a Hamming code through a tabular template. An original implementation of the considered algorithm in PascalABC.NET is proposed. When writing code, modern techniques are used: dynamic arrays, slices, safe slices, conditional (ternary) operation, foreach loop, sequence methods, lambda expressions, tuples, documenting comments, etc. The functions of the School module from the official delivery of the PascalABC.NET compiler are used to work with binary numbers. The described methodology has been tested in teaching 4-year students of the pedagogical direction, the profile "Mathematics and Informatics" of the Orenburg State Pedagogical University and has shown its applicability. In addition, it is possible to use the proposed approach for teaching schoolchildren at a specialized level or in extracurricular work.

***Keywords*:** theoretical foundations of informatics, error correcting coding, Hamming code, programming, PascalABC.NET.

2. Помехоустойчивое кодирование. Основные сведения

Как известно, в канале связи неизбежны шумы, а поэтому часть информации, передаваемой по таким каналам, может быть утеряна или передана с ошибкой. Помимо физико-технических способов борьбы с шумами в каналах связи существуют программные методы, заключающиеся в специальных способах кодирования передаваемой информации. Такие методы называются *помехоустойчивыми*.

Если в системе возможно повторение сигнала, часто бывает достаточно лишь обнаружить ошибку, после чего повторить передачу еще и еще. Число повторений зависит от принятой модели ошибок. Например, если после третьей попытки принятое сообщение оказалось неприемлемым, то происходит переход к следующей задаче/сообщению. При этом обнаружить ошибку невозможно, если любой принятый символ может служить сообщением. Как показано в [10], ошибки можно обнаружить только в том случае, если на сообщение наложены некоторые ограничения. Очевидно стремление найти наиболее простые, а значит, легко вычислимые ограничения. Например, можно использовать *проверку четности*, когда вместе с сообщением передается дополнительный бит, содержащий 0, если общее число единиц в двоичном коде исходного сообщения четно, и содержащий 1 — в противном случае. В приемнике (декодере) производится подсчет числа единиц, и нечетное их число *во всех позициях, включая дополнительный бит,* указывает на появление по крайней мере одной ошибки. Заметим, что двукратные ошибки этим кодом обнаружить невозможно, но возможно обнаружение нечетного числа ошибок. Так, при четном количестве ошибок в последнем бите будет записан 0 (что будет воспринято как верное сообщение), а при нечетном — 1 (передача повторяется). Проверка четности — это, по сути, сложение по модулю 2, т. е. суммирование всех битов, а затем взятие остатка при делении на 2. Эта операция аналогична логической XOR.

Таким образом, для построения помехоустойчивого кода к битам, содержащим информацию (информационные биты), добавляются проверочные. В результате коды становятся избыточными. В качестве количественной меры принимается *относительная избыточность помехоустойчивого кода,* которая показывает, какая часть переданной кодовой комбинации не содержит первичной информации [9]. Если принять за k количество информационных битов, m — количество проверочных битов, то избыточность кода рассчитывается как m/k. Например, если при передаче трех битов к ним добавить один проверочный бит, то избыточность составит $1/3 \approx 0{,}33$. Понятно, что за избыточность кода нужно как-то «расплачиваться»: либо увеличивать пропускную способность канала связи, либо увеличивать длительность передачи. Поэтому одной из задач теории помехоустойчивого кодирования является *построение кода с наименьшей избыточностью*.

Другая задача вытекает из простого соображения: если компьютер может обнаружить ошибку, то почему бы ему ее и не исправить. С того момента, как Ричард Хемминг задался таким вопросом, и началась история развития *самокорректирующихся,* или *помехоустойчивых, кодов*.

В общем случае помехоустойчивые коды имеют маркировку (n, k) по числу передаваемых и числу информационных битов. В (n, k)-кодах с возможностью обнаружения и исправления *одной* ошибки число проверочных битов m и *максимальное* число информационных разрядов k связаны формулой, которую приводим по [2]: $k = 2^m - m - 1$. Так, если в код добавить три проверочных бита, то максимальное число передаваемых информационных битов будет равно четырем, общее число битов станет равно семи. Избыточность кода (7, 4) составит $3/4 = 0{,}75$. Как показано в [9], выгоднее передавать более длинные цепочки информационных битов, поскольку m изменяется не пропорционально k. Однако с ростом k повышается вероятность появления ошибок высокой кратности.

Подробное описание помехоустойчивых кодов, включая их классификацию, характеристики и оценку помехоустойчивости можно найти, например, в [2, 4, 7].

3. Код Хемминга — код с исправлением ошибки

Коды, исправляющие ошибки, особенно востребованы в системах, в которых повторение передачи невозможно. Например, при длительной передаче или передаче на большие расстояния (с Марса на Землю) может случиться так, что к моменту обнаружения ошибки исходная информация может быть уже утеряна. Здесь мы ограничимся рассмотрением исторически первого самокорректирующегося кода — *кода Хемминга*.

Ричард Хемминг построил код, использующий *несколько независимых* проверок на четность. Если записать число 0 для каждой выполненной проверки и 1 — для невыполненной и расположить эти проверки в определенном порядке, то они образуют двоичное число, которое Хемминг назвал *синдромом ошибки*. «Синдром ошибки — это признак, по которому ошибка передачи может быть локализована» [9, с. 167]. Другими словами, синдром позволяет обнаружить позицию ошибочного бита. При этом независимость проверок означает, что никакая сумма двух проверок не совпадает ни с какой другой проверкой.

Поясним последнее утверждение на примере. Рассмотрим часть некоторого сообщения: 011001010. Пусть даны три проверки на четность в некоторых позициях:

- проверка № 1 представляет собой сумму нулей и единиц в позициях 1, 2, 5, 6;
- проверка № 2 — сумма нулей и единиц в позициях 5, 6, 8, 9;
- проверка № 3 — сумма нулей и единиц в позициях 1, 2, 8, 9.

Выстроим проверки одну под другой по позициям:

1:	1	2	5	6						1:	0	1	0	1	
2:			5	6	8	9		то же в битах		2:			0	1	1	0
3:	1	2			8	9				3:	0	1			1	0

Заметим, что сумма первой и второй проверок совпадет с третьей (напомним, что суммирование происходит по модулю 2, или как результат операции XOR). Поэтому эти три проверки являются зависимыми.

В 1950 году Хемминг предложил правило построения синдрома из независимых проверок в позициях, являющихся степенями двойки.

Допустим, мы знаем, что номер ошибочного бита равен $11 = 8 + 2 + 1 = 2^3 + 2^1 + 2^0$. Значит, единицы в двоичном представлении числа 11 стоят в 0-й, 1-й и 3-й позициях. Если же *перенумеровать номера позиций начиная с первой* (как это делает сам Хемминг), то это будут 1-я, 2-я и 4-я позиции справа. Тогда, для того чтобы синдром указывал номер ошибочного бита (11), нужно, чтобы:

- в первую проверку входили все биты, у которых в двоичной записи их номера на первой справа позиции записана 1 — биты с номерами: 1, 3, 5, 7, 9, 11, ...;
- во вторую проверку входили все биты, у которых в двоичной записи их номера на второй справа позиции записана 1 — биты с номерами: 2, 3, 6, 7, 10, 11, ..., т. е. те числа, в развернутую запись которых (в двоичной системе счисления) входит $2^1 = 2$;
- в третью проверку входили все биты, у которых в двоичной записи их номера на третьей справа позиции записана 1 — биты с номерами: 4, 5, 6, 7, 12, 13, 14, 15, ..., т. е. те числа, в развернутую запись которых (в двоичной системе счисления) входит $2^2 = 4$;
- и т. д.

Заметим, что в первую проверку вошли биты начиная с 1-го по одному через один; во вторую — начиная с бита номер 2 по два через два (2, 3 входят, 4, 5 — пропущены, 6, 7 снова входят, и т. д.); в следующую проверку вошли биты начиная с 4-го по четыре через четыре; и т. д. Это становится хорошо заметно, если выписать четырехразрядные двоичные числа и их десятичные аналоги (посмотрите на столбики нулей и единиц поразрядно на рисунке 1).

Тогда разумно назвать эти проверки по номеру первого входящего бита — 1, 2, 4, 8, ... и вставить их в передаваемое сообщение в позиции с соответствующим номером. То есть позиции, равные степеням двойки, хранят проверочные биты, а все остальные биты являются информационными. Все эти рассуждения следуют из единственности представления любого числа в (традиционной) позиционной системе счисления, в частности, в двоичном виде.

Если теперь повернуть рисунок 1 на 90° против часовой стрелки, то получим так называемый *шаблон*, дающий наглядное представление процесса вычисления синдрома.

На рисунке 2 показан шаблон длиной 21 бит. Для большей наглядности после поворота единицы заменены на крестики. Важно: *биты нумеруются начиная с единицы и слева направо* (в отличие от обычной нумерации с нуля и справа налево). Для кодирования 21 числа необходимы пять битов, следовательно, шаблон содержит пять проверочных битов с номерами 1, 2, 4, 8, 16. Оставшиеся 16 бит являются информационными.

0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

Рис. 1. Чередование одинаковых битов в двоичном представлении (для второго справа разряда)

Заметим, что *проверочные и информационные биты являются равноправными* в этом способе кодирования, т. е. проверочные биты исправляются так же, как информационные.

Итак, имеем следующий **алгоритм получения кода Хемминга:**

- Представить сообщение в двоичном виде.
- Вставить в позиции, равные степеням двойки, дополнительные проверочные биты. В результате общая длина сообщения увеличится.
- Вычислить значения проверочных битов по шаблону.

Декодер, получив сообщение, должен заново вычислить значения проверочных битов. Затем происходит процедура вычисления синдрома: вновь вычисленные проверочные биты сравниваются с проверочными битами, пришедшими вместе с сообщением. Если соответствую-

№ бита		**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20	21
проверка	1	×		×		×		×		×		×		×		×		×		×		×
	2		×	×			×	×			×	×			×	×			×	×		
	4				×	×	×	×					×	×	×	×					×	×
	8								×	×	×	×	×	×	×	×						
	16																×	×	×	×	×	×

Рис. 2. Шаблон кода Хемминга с пятью проверками

щие биты совпадают, то в синдром на заданную позицию записывается нуль, иначе — единица. Если в результате синдром равен нулю, то ошибки в передаче не было. В противном случае мы полагаем, что была одна ошибка в бите, номер которого — десятичное представление синдрома.

4. Пример работы алгоритма

Пусть требуется закодировать и передать по каналу связи слово *Sign*. Представим каждый символ заданного слова в двоичном виде в соответствии с таблицей ASCII (см. табл.). В результате получим строку из 8 × 4 = 32 символов: 01010011011010010110011101101110.

Таблица

ASCII-коды

Символ	S	i	g	n
Dec	83	105	103	110
Bin	01010011	01101001	01100111	01101110

Обычно сообщение передается порциями, например, по 16 бит (одно слово). Поэтому разобьем строку на две по 16 бит каждая:

Si = ‘0101001101101001’;

gn = ‘0110011101101110’.

Будем работать с каждой строкой отдельно.

В строку *Si* = ‘0101001101101001’ необходимо вставить проверочные биты в позиции, равные степеням двойки. Возникает вопрос: сколько дополнительных битов потребуется? Так как $16 = 2^4$, то потребуется 4 дополнительных бита в позициях 1, 2, 4 и 8. Но так как после вставки каждого дополнительного бита происходит сдвиг остальных символов вправо, общая длина строки увеличивается и становится равна 16 + 4 = 20 бит. Значит, в позицию с номером 16 тоже добавляется проверочный бит (всего 5 бит), и окончательная длина строки становится равна 16 + 5 = 21 символу. Таким образом, для строки длиной N понадобится $\log_2 N + 1$ проверочных битов (для случая, когда сообщение будет передоваться порциями не по 16 бит, а, например, по 32 или 64 бита). Для наглядности представим эти новые биты в виде символа ‘*’. Получаем строку: **0*101*0011011*01001.

Теперь необходимо вычислить значения проверочных битов. Для этого воспользуемся шаблоном из рисунка 2, добавив в него строку с проверочными и информационными битами (рис. 3).

Просуммируем биты над крестиками первой проверки (все нечетные биты). При этом ‘*’ считаем за 0 (для проверки четности важны только единицы). Единицы содержатся в битах с номерами 5, 7, 11, 15, 21 — всего 5 штук. Так как 5 — нечетное, то *в первый бит запишем 1*. Далее суммируем число единиц в строке с битами над крестиками второй проверки — это 5 бит с номерами 7, 11, 14, 15, 18 (в остальных битах этой проверки содержатся нули). Так как 5 — нечетное, то *во второй бит также запишем 1*. Аналогично поступаем с проверками под номерами:

- 4 — 6 единиц, четное, *в четвертый бит записываем 0*;
- 8 — 4 единицы, четное, *в восьмой бит записываем 0*;
- 16 — 2 единицы, *в шестнадцатый бит записываем 0*.

В итоге получаем новую строку 11001010001101100**1**001 (полученные значения проверочных битов выделены серым), которую и отправляем.

Допустим теперь, что строка, которую получил приемник на другом конце канала связи, содержит ***одну*** ошибку, например, в бите номер 19. То есть в 19-м бите вместо нуля пришла единица — 11001010001101100**1**101 (испорченный бит выделен полужирным).

Как нам узнать об этом? Обнулим все проверочные биты в полученной строке и вычислим их заново (рис. 4).

Сравним проверочные биты из приемника с теми, которые вычислил декодер. Наблюдаются отличия в проверочных битах под номерами 1, 2 и 16. Если теперь сложить эти три числа, то получим номер «испорченного» бита. После этого нужно:

- инвертировать испорченный бит;
- удалить из строки проверочные биты;
- разделить строку на две по 8 бит;
- подставить вместо ASCII-кода каждой из полученных битовых последовательностей соответствующий символ.

Аналогично поступают со второй частью сообщения: *gn* = ‘0110011101101110’. Для тренировки предлагаем провести эти процедуры самостоятельно.

Замечания. Длину передаваемого за один раз сообщения можно выбрать любой. То есть можно передавать порции по 8 бит, а можно и по 32 бита. Очевидно, что избыточность в первом случае выше (4 / 8 = 0.5),

№ бита		**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20	21
Биты		*	*	0	*	1	0	1	*	0	0	1	1	0	1	1	*	0	1	0	0	1
проверка	1	×		×		×		×		×		×		×		×		×		×		×
	2		×	×			×	×			×	×			×	×			×	×		
	4				×	×	×	×					×	×	×	×					×	×
	8								×	×	×	×	×	×	×	×						
	16																×	×	×	×	×	×

Рис. 3. Шаблон для первой части кодируемого сообщения до вычисления значений проверочных бит

№ бита	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18	19	20	21
Получено приёмником	**1**	**1**	0	**0**	1	0	1	**0**	0	0	1	1	0	1	1	**0**	0	1	1	0	1
Обнуление проверок	**0**	**0**	0	**0**	1	0	1	**0**	0	0	1	1	0	0	1	**0**	0	1	1	0	1
Вычислено декодером	**0**	**0**	0	**0**	1	0	1	**0**	0	0	1	1	0	1	1	**1**	0	1	1	0	1

Рис. 4. Вычисление проверочных битов декодером

а во втором — меньше (5 / 32 ≈ 0.16). Но в этой версии кода за одну передачу можно обнаружить и исправить только одну ошибку. Поэтому длина в 16 бит является приемлемой.

5. Компьютерная реализация алгоритма

Приводимый ниже код — учебный. Поэтому основная его ценность состоит в наглядности, а не в эффективности или универсальности. Работая над программой, студент не только лучше разбирается в алгоритме, но и разрабатывает довольно крупный проект, требующий умений проектировать код. Программа написана на языке PascalABC.NET.

В программе используется модуль *School*, из которого мы берем функции перевода между двоичной и десятичной системами счисления, чтобы не загромождать текст очевидными алгоритмами. Полная структура программы приведена в листинге 1.

Рассмотрим код в стиле проектирования «сверху вниз», начиная с самых главных подпрограмм и постепенно переходя к вспомогательным.

```
uses School;

var
  Mask := |'10' * 10 + '1',
           '0' + '1100' * 5,
           '0' * 3 + '11110000' * 2 + '11',
           '0' * 7 + '1' * 8 + '0' * 6,
           '0' * 15 + '1' * 6|;
///Вставка битов четности в 16-битное двоичное число
function AddParityBits(s: string; c: char := '0'): string;
///Вычисление количества единиц по j-ой строке маски
function CountOnes(s: string; j: integer): integer;
///Вычисление значений битов четности
function CountParityBits(s: string): string;
///Кодирование Хемминга для двух символов
function HCode16(s: string): string;
///Кодирование Хемминга для произвольной строки
function HammingCode(s: string): array of string;
///Обнуление битов четности
function ResetParityBits(s: string; c: char := '0'): string;
///Вычисление синдрома по полученному сообщению
function CountSyndrome(s, t: string): integer;
///Удаление битов четности
function DelParityBits(s: string): string;
///Преобразование двоичного кода символа в символ
function BinToChar(s: string) := Chr(Dec(s, 2));
///Декодирование двух символов
function HDecode16(s: string): string;
///Декодирование Хемминга для произвольной строки
function HammingDecode(a: array of string): string;

begin
  var s := 'Sign'; // кодируемое сообщение
  var a := HammingCode(s); // получение кода Хемминга для сообщения
  a[0][3] := '1'; // Внесение ошибки в третий бит
  HammingDecode(a).PrintLn; // Раскодирование с исправлением ошибки и печать
end.
```

Листинг 1. Структура программы для кодирования Хемминга

```
1.   function HammingCode(s: string): array of string;
2.   begin
3.     if s.Length.IsOdd then
4.       s += ' ';
5.     var a := s.Batch(2).Select(x -> x.JoinToString).ToArray;
6.     for var i := 0 to a.High do
7.       a[i] := HCode16(a[i]);
8.     result := a;
9.   end;
```

Листинг 2. Функция для кодирования произвольной строки

```
1.   function HCode16(s: string): string;
2.   begin
3.     Assert(s.Length = 2, 'Длина входной строки должна быть равна 2');
4.     var b := '';
5.     foreach var c in s do
6.     begin
7.       var t := Bin(c.Code);
8.       t := '0' * (8 - t.Length) + t;
9.       b += t;
10.    end;
11.    var a := AddParityBits(b);
12.    result := CountParityBits(a);
13.  end;
```

Листинг 3. Основная функция для кодирования двухсимвольной строки

Для непосредственной работы предназначены две функции: *HammingCode* и *HammingDecode*. Первая производит кодирование по Хеммингу с возможностью исправления одной ошибки. Вторая декодирует сообщение, подготовленное первой функцией. В случае, если при передаче была ошибка в одном бите, декодер исправляет ее и возвращает раскодированное сообщение.

Функция *HammingCode* (листинг 2) принимает на вход произвольную строку и возвращает массив из нескольких строк. В каждой ячейке выходного массива записаны очередные два символа, закодированные по алгоритму Хемминга и представленные в виде строки. Если длина строки нечетная, то в ее конец добавляется пробел. Это нужно, так как основной алгоритм кодирования обрабатывает по два символа. Затем разбиваем строку на серии по два символа (*Batch*(2)), символы объединяем в строку (*JoinToString*) и получившуюся последовательность превращаем в массив (*ToArray*). Здесь используются методы для работы с последовательностями*.

Таким образом, в переменной *a* теперь строковый массив, каждым элементом которого является строка из двух символов. В строках 6–7 каждая такая двухсимвольная строка передается на вход основной функции кодирования *HCode16* (листинг 3), после чего возвращает закодированное сообщение.

В строке 3 листинга 3 мы проверяем, что получили на вход именно двухсимвольную строку; в противном случае выводим сообщение об ошибке. В строках 5–10 мы превращаем двухсимвольную строку в строку двоичных кодов этих символов. Каждый из кодов при необходимости дополняется незначащими нулями слева до длины 8 бит (строка 8). Функция *Bin* из модуля *School* превращает десятичное целое число в двоичное, представленное в виде строки. Метод *Code* возвращает код символа (аналогично *Ord*(*c*)). В строке 11 вставляем на нужные позиции в 16-битную строку биты четности. В строке 12 вычисляем эти биты и возвращаем результат в виде строки длиной 16 (исходных битов) + 5 (битов четности) = 21 символ.

Функция *AddParityBits* вставляет в 16-элементную строку дополнительные символы на позиции 1, 2, 4, 8, 16 (при этом длина строки увеличивается до 21). Функция устроена следующим образом (листинг 4).

Функция *AddParityBits* принимает на вход кодируемую строку и необязательный параметр, задающий символ, вставляемый в строку (по умолчанию нуль). Сам код, как мы думаем, в особенном анализе не нуждается. Заметим только, что в строках 8 и 11 использованы безопасные срезы `s?[from:k]`, где знак вопроса как раз обеспечивает безопасность: в случае, если параметры среза выходят за допустимые границы, они возвращаются в нужный диапазон, так что ошибки выхода за границу строки не происходит. Вот пример работы этой функции:

```
var s := '1234567890123456';
AddParityBits(s, '*').Print// Вывод:
**1*234*5678901*23456
```

Звездочки помогают увидеть позиции, в которые вставляются дополнительные символы. В реальном алгоритме на эти места записываются нули.

Теперь нужно вычислить значения вставленных битов четности. Это делает функция *CountParityBits* (листинг 5).

* О новых возможностях языка PascalABC.NET можно узнать, например, из книги А. В. Осипова [5] и его же онлайн-курса на Stepik [11].

```
1.   function AddParityBits(s: string; c: char := '0'): string;
2.   begin
3.     var k := 2;
4.     var from := 1;
5.     var t := c * 2;
6.     while (k < s.Length) do
7.     begin
8.       t := t + s?[from:k] + c;
9.           (from, k) := (k, k * 2 + from);
10.    end;
11.    t := t + s?[from:k];
12.    result := t;
13.  end;
```

Листинг 4. Функция для вставки битов четности в кодируемую строку

```
1.   function CountParityBits(s: string): string;
2.   begin
3.     result := s;
4.     var j := 1;
5.     for var i := 0 to mask.High do
6.     begin
7.       if odd(CountOnes(s, i)) then
8.         result[j] := '1'
9.       else
10.        result[j] := '0';
11.      j *= 2;
12.    end;
13.  end;
```

Листинг 5. Функция для вычисления битов четности

Функция *CountParityBits* должна вписать на место нулей, вставленных на предыдущем шаге, реальные значения четности, вычисляемые по нескольким шаблонам. Все эти шаблоны задаются в программе в виде массива строк (см. переменную *Mask* в листинге 1). Переменная *Mask* имеет такой вид:

```
10101010101010101010
01100101001100110011
00011110000111100011
00000001111111100000
00000000000000111111
```

Каждая *i*–я строка — это шаблон для вычисления *i*–го бита четности (сравните с рисунком 2). Другими словами, чтобы вычислить первый бит четности, нужно сложить по модулю 2 те биты исходной строки, которые стоят на позициях, отмеченных в первой строке шаблона единицами. Для вычисления второго бита смотрим на вторую строку маски, для четвертого бита — на третью строку и т. д. Собственно подсчетом суммы занимается функция *CountOnes*, которая получает на вход кодируемую строку со вставленными нулевыми битами четности и номер строки в массиве *Mask*, в соответствии с которым нужно проводить суммирование. Код функции *CountOnes* приведен в листинге 6.

Функция *CountOnes* устроена просто. В строке 3 кода из массива шаблонов извлекается нужная маска (в данном случае с индексом *j*). Затем проходим по всей строке (*Select*); выбираем очередной элемент *c* (с индексом *i*). Параллельно с этим выбирается соответствующий элемент из маски *m*[*i* + 1]. Вспомним, что нумерация элементов строк начинается с единицы, а нумерация элементов последовательности — с нуля. Это значит, что символы строки *m* нумеруются с единицы, а в методе *Select* первое значение переменной *i* равно нулю. Поэтому при обращении к элементу маски мы используем индекс *i* + 1. Поскольку и в масках, и в строке содержатся лишь символы, изображающие нуль и единицу, мы превращаем их в настоящие цифры методом *ToDigit*. Применяя к соответствующим цифрам операцию and, мы накладываем маску на строку. При этом элементы маски, равные нулю, «обнулят» соответствующие элементы

```
1.   function CountOnes(s: string; j: integer): integer;
2.   begin
3.     var m := Mask[j];
4.     result := s.Select((c, i)-> (c.ToDigit and m[i + 1].ToDigit)).Sum;
5.   end;
```

Листинг 6. Функция для вычисления бита четности по одному шаблону

```
1.   function HammingDecode(a: array of string): string;
2.   begin
3.     result := '';
4.     foreach var s in a do
5.       result += HDecode16(s);
6.   end;
```

Листинг 7. Функция для декодирования произвольной строки

```
1.   function HDecode16(s: string): string;
2.   begin
3.     var t := CountParityBits(ResetParityBits(s));
4.     var Syndrome := CountSyndrome(s, t);
5.     if Syndrome > 0 then
6.       t[Syndrome] := if t[Syndrome] = '0' then '1' else '0';
7.     t := DelParityBits(t);
8.     var a := t[:9];
9.     var b := t[9:];
10.    result := BinToChar(a) + BinToChar(b);
11.  end;
```

Листинг 8. Основная функция для декодирования двухсимвольной строки

строки, а равные единице — не изменят их. В конце для получившейся последовательности вызывается метод *Sum*, который вычисляет сумму элементов получившейся последовательности из нулей и единиц; очевидно, она совпадает с количеством единиц. Это число и является результатом работы функции.

Таким образом, мы полностью описали процедуру кодирования Хемминга.

Рассмотрим теперь декодирование полученного сообщения с возможным исправлением одной ошибки.

Как уже упоминалось, основная функция *HammingDecode* декодирует сообщение произвольной длины. Код функции приведен в листинге 7.

Функция *HammingDecode* принимает на вход массив строк, причем каждая строка — это два закодированных символа исходного сообщения. Результатом работы функции является строка символов, которая, как нам хотелось бы, совпадает с исходной. Алгоритм декодирования следующий. Для каждой строки из поступившего на вход массива мы вызываем основную функцию декодирования *HDecode16*, которая возвращает строку из двух символов. Эту строку мы накапливаем в переменной *result* и возвращаем. Таким образом, главная часть алгоритма лежит в функции *HDecode16* (листинг 8).

Вначале мы сбрасываем (обнуляем) биты четности в исходной строке (функция *ResetParityBits*), а затем сразу же перевычисляем их (функция *CountParityBits*). Результат *t* будет совпадать с входной строкой *s*, которую мы получили по каналу связи, если строка *s* прибыла без ошибок (напомним, что в нашем алгоритме предусмотрено исправление только одной ошибки). Если же строки *s* и *t* различны, то передаваемая строка была повреждена в канале связи. Для определения факта и места ошибки мы вычисляем синдром — функция *CountSyndrome*, которая возвращает десятичный номер позиции, в которой произошла ошибка, либо нуль, если ошибки не было или она была в бите четности. Если ошибка была (*Syndrome* > 0), то в *t* мы инвертируем бит, на который указывает синдром. Для этого удобно использовать условную операцию:

```
x := if a > 0 then 1 else 0;
```

Здесь *x* станет равен либо 1, либо 0 в зависимости от истинности сравнения ($a > 0$). Условную операцию нельзя путать с условным оператором. Больше всего она похожа на функцию ЕСЛИ в электронных таблицах. Условную операцию можно записать и в виде тернарной операции Си-подобных языков:

```
x := a > 0? 1 : 0;
```

Это короче, но, на наш взгляд, чуть сложнее прочитать.

Далее в строке 7 из строки *t* удаляются биты четности (функция *DelParityBits*). В строках 8 и 9 мы разрезаем получившуюся 16-символьную строку на две по 8 символов. Затем при помощи функции *BinToChar* мы превращаем строку, изображающую двоичное число, в символ с соответствующим кодом. Склеенные в строку символы возвращаются как результат работы функции *HDecode16*.

Рассмотрим еще не описанные функции. Первая из встречающихся — *ResetParityBits* (листинг 9).

Здесь мы меняем переменную *k* по правилу 1, 2, 4, 8, 16 и заменяем соответствующие биты заданными символами *c* (по умолчанию нулем).

Функция *CountSyndrome* приведена в листинге 10.

Проходим по всем битам, номера которых совпадают со степенями двойки, и для каждого такого бита вычисляем сумму по модулю 2 у обеих входных строк, а результаты суммируем. В строке 11 разворачиваем синдром, превращая его в запись двоичного числа, а затем при помощи функции *Dec* из модуля *School* переводим его в десятичную систему счисления.

Функция *DelParityBits* (листинг 11) физически удаляет из строки биты четности, когда они уже выполнили свою роль и больше не нужны.

```
1.   function ResetParityBits(s: string; c: char := '0'): string;
2.   begin
3.     var k := 1;
4.     for var i := 1 to s.Length do
5.       if i = k then
6.       begin
7.         k *= 2;
8.         s[i] := c
9.       end;
10.    result := s;
11.  end;
```

Листинг 9. Функция для обнуления битов четности

```
1.   function CountSyndrome(s, t: string): integer;
2.   begin
3.     var Syndrome := '';
4.     var k := 1;
5.     for var i := 1 to s.Length do
6.       if i = k then
7.       begin
8.         k *= 2;
9.         Syndrome += t[i].ToDigit xor s[i].ToDigit;
10.      end;
11.    result := Dec(Syndrome[::-1], 2);
12.  end;
```

Листинг 10. Вычисление синдрома для строки, полученной по каналу связи

```
1.   function DelParityBits(s: string): string;
2.   begin
3.     var k := 1;
4.     result := '';
5.     for var i := 1 to s.Length do
6.       if i = k then
7.         k *= 2
8.       else
9.         result += s[i];
10.  end;
```

Листинг 11. Удаление из строки битов четности

```
1.   function BinToChar(s: string) := Chr(Dec(s, 2))
```

Листинг 12. Преобразование двоичного кода символа в символ

Код очень похож на предыдущий. Проходим по строке, и если номер символа не является степенью двойки, то добавляем символ к результату.

Последнюю функцию *BinToChar* запишем в короткой форме (листинг 12).

Здесь мы берем 8-символьную строку, представляющую собой двоичную запись кода символа, превращаем ее в десятичное число (*Dec*) и затем в символ с соответствующим кодом (*Chr*).

Вернемся к рассмотрению главной программы (листинг 1). Мы кодируем сообщение ‘Sign’ при помощи функции *HammingCode*, после чего будто бы передаем его по каналу связи. В канале связи сообщение подвергается воздействию помех, в результате чего третий бит левого символа превращается из 0 в 1. Затем мы вызываем функцию *HammingDecode*, которая получает на вход закодированное сообщение, находит и исправляет ошибку. Результат выводится на экран, и это строка ‘Sign’.

6. Заключение

Ричард Хемминг является классиком информатики, а его работы в области кодирования можно смело отнести к фундаментальным. Изучение идей Хемминга позволяет не только лучше освоить теорию кодирования, но и попрактиковаться в программировании не самых элементарных алгоритмов и в реализации проектов более крупных, чем обычная учебная задача.

Использование современного стиля (методы последовательностей LINQ, кортежи, срезы и пр.) уменьшает объем кода и повышает его читаемость. Вместе с тем написание таких программ доступно не только студентам,

но и продвинутым школьникам профильных классов, занимающимся проектной деятельностью по информатике. Совместная работа над проектом в группе дает возможность приобрести навыки коллективной работы. Здесь очень полезной может оказаться система контроля версий (например, GitHub.com). Работу в таких системах можно рассматривать как серьезный шаг в направлении промышленного программирования.

Список использованных источников

1. *Босова Л. Л., Босова А. Ю., Аквилянов Н. А., Куклина И. Д., Мирончик Е. А.* Информатика. 10–11 классы. Базовый уровень: методическое пособие. М.: БИНОМ. Лаборатория знаний, 2020. 470, [10] с.

2. *Вернер М.* Основы кодирования: учебник для вузов. М.: Техносфера, 2004. 288 с.

3. *Калинин И. А., Самылкина Н. Н.* Информатика. 10 класс. Углубленный уровень: учебник. М.: БИНОМ. Лаборатория знаний, 2019. 256 с.

4. *Морелос-Сарагоса Р.* Искусство помехоустойчивого кодирования. Методы, алгоритмы, применение. М.: Техносфера, 2005. 320 с.

5. *Осипов А.* PascalABC.NET. Введение в современное программирование. http://pascalabc.net/downloads/OsipovBook/ModernProgr.pdf

6. *Поляков К. Ю., Еремин Е. А.* Информатика. 11 класс. Базовый и углубленный уровни: учебник: в 2 ч. Ч. 1. М.: БИНОМ. Лаборатория знаний, 2021. 304 с.

7. *Прокис Дж.* Цифровая связь: пер. с англ. / под ред. Д. Д. Кловского. М.: Радио и связь, 2000. 800 с.

8. *Семакин И. Г., Шеина Т. И., Шестакова Л. В.* Информатика. Углубленный уровень: учебник для 10 класса: в 2 ч. Ч. 1. М.: БИНОМ. Лаборатория знаний, 2014. 184 с.

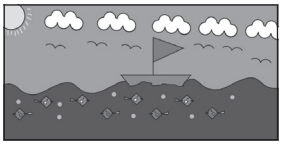
9. *Стариченко Б. Е.* Теоретические основы информатики: учебник для вузов. 3-е изд., перераб. и доп. М.: Горячая линия — Телеком, 2016. 400 с.

10. *Хэмминг Р. В.* Теория кодирования и теория информации: пер. с англ. М.: Радио и связь, 1983. 176 с.

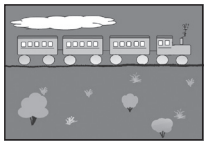
11. PascalABC.NET: современный код. https://stepik.org/course/91781/

КОНКУРСЫ

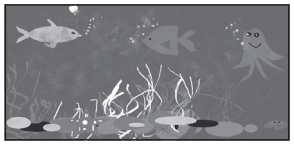
В оформлении обложки данного номера журнала «Информатика в школе» использованы работы следующих авторов — победителей конкурса цифровых изображений и фотографий:



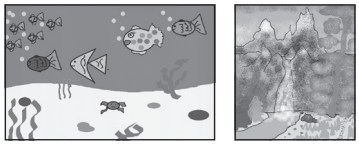
Григорий Санников,
учащийся объединения «Информатика» Центра технического творчества, ученик средней общеобразовательной школы № 60, г. Киров



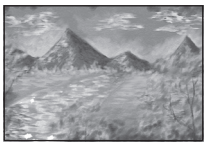
Александр Ширяев,
учащийся объединения «Информатика» Центра технического творчества, ученик средней общеобразовательной школы № 52, г. Киров



Валерия Воронцова,
учащаяся объединения «Информатика» Центра технического творчества, ученица средней общеобразовательной школы № 30, г. Киров

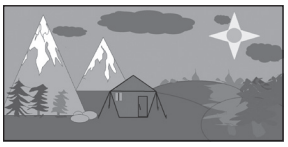


Алина Сысолятина,
учащаяся объединения «Информатика» Центра технического творчества, ученица средней общеобразовательной школы № 30, г. Киров

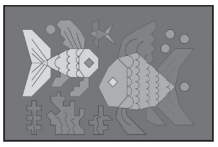


Анна Макарова,
ученица средней общеобразовательной школы № 25, г. Сергиев Посад, Московская область

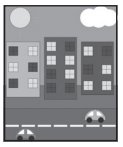
Виктория Четыркина,
ученица гимназии № 39, г. Петропавловск-Камчатский



Иван Шарыбкин,
ученик гимназии № 39, г. Петропавловск-Камчатский



Дмитрий Манухин,
ученик Наро-Фоминской средней общеобразовательной школы № 1, г. Наро-Фоминск, Московская область



Егор Игнатов,
ученик Наро-Фоминской средней общеобразовательной школы № 6 с углубленным изучением отдельных предметов, г. Наро-Фоминск, Московская область

Уважаемые авторы и читатели!

Издательство «Образование и Информатика» объявляет о проведении конкурса цифровых изображений и фотографий, которые будут размещены на обложке журнала «Информатика в школе» в первом полугодии 2022 года.

Подробности — на сайте издательства «Образование и Информатика»: http://www.infojournal.ru/