МИНОБРНАУКИ РФ

ФГБОУ ВПО Тверской государственный технический университет

Кафедра «Программное обеспечение».

Курсовая работа.

Дисциплина «Алгоритмы и структуры данных».

Тема: «Алгоритмы поиска ошибок при передачи текста по информационному каналу».

Выполнил: студент группы

ПИН 17.05

Иванов Р.В

Проверил:

Мальков А.А

Тверь 2018

Оглавление

[Введение. 2](#_Toc533366130)

[Аналитическая часть. 3](#_Toc533366131)

[История создание кода Хэмминга. 3](#_Toc533366132)

[О кодах исправляющих несимметричные ошибки. 4](#_Toc533366133)

[Реализация кода Хэмминга 8](#_Toc533366134)

[Подготовка. 8](#_Toc533366135)

[Вычисление контрольных бит. 8](#_Toc533366136)

[Декодирование и исправление ошибок. 10](#_Toc533366137)

[Вывод. 10](#_Toc533366138)

[Бит четности. 11](#_Toc533366139)

[Примеры 11](#_Toc533366140)

[Вывод. 11](#_Toc533366141)

[Мой собственный алгоритм или алгоритм контрольной суммы. 12](#_Toc533366142)

[Плюсы и минусы алгоритма. 13](#_Toc533366143)

[Выводы. 13](#_Toc533366144)

[Проектная часть. 13](#_Toc533366145)

[Функционал 13](#_Toc533366146)

[Архитектура приложения 14](#_Toc533366147)

[Диаграмма классов 14](#_Toc533366148)

[Реализация проекта 14](#_Toc533366149)

[Хронология разработки проекта 17](#_Toc533366150)

[Тестирование 19](#_Toc533366151)

[Заключение 20](#_Toc533366152)

[Источники 20](#_Toc533366153)

# Введение.

Цель: Ознакомится с алгоритмами поиска ошибок при передачи текста по информационному каналу и реализовать их.

Задача: Реализовать Windows Form проект на c#.

Реализовано: код Хэмминга, добавление бита четности, собственный алгоритм «Контрольная сумма».

# Аналитическая часть.

## История создание кода Хэмминга.

 Первая работа по теории кодирования была опубликована известным ученым Ричардом Уэсли Хэммингом в 1950 году. Привело его к этой работе следующее: после окончания университета он занимался тем, что сейчас бы назвали численными методами, и работал сначала в Лос-Аламосе над Манхэттенским проектом, то есть над созданием ядерной бомбы, а потом почти сразу, в 1946 году, перешел в *Bell Labs*. Это та великая лаборатория, в которой было сделано, наверное, самое большое количество открытий второй половины XX века. Там он продолжал заниматься численными методами, решать уравнения на первых вычислительных машинах.

Вычислительные машины были огромными, ненадежными, потому что на лампах, и для их охлаждения требовалось большое количество воды, и тем не менее они часто сбоили. Программисты того времени, чтобы бороться со сбоями, делили задачу на этапы, и после каждого этапа программа должна была записать или распечатать результаты. Тогда задачи были простенькие по теперешним понятиям, в основном счет.

Зачем это делалось? Так как эти компьютеры были ненадежными, у них было такое устройство, как сигнал тревоги. Когда работа шла неверно, когда они обнаруживали ошибку, то звучал сигнал тревоги, и работа прекращалась. Соответственно, человек мог с этого места посмотреть назад, какое было первое от этого места правильно посчитанное значение, и продолжать счет дальше. Хэмминг оставлял свою работу на выходные и уходил спокойно отдыхать, а приходя, обнаруживал, что машина работала впустую, потому что в некий момент происходила ошибка, но никого не было, его не было, чтобы остановить. Значит, все его труды пропадали даром. Тогда у него возникла совершенно естественная даже не идея, а вопрос: если машина умеет обнаруживать ошибки, почему бы не научить ее, чтобы она их исправляла?

Он подумал, почему, собственно, проверять все значения? Может, одного проверочного символа недостаточно? Может быть, добавить не один, а два, три, четыре - столько, сколько понадобится, и исправлять?

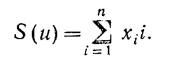
Так и возник, наверное, самый известный код, так называемый (7,4) код Хемминга. 4 означает, что мы будем защищать от ошибок 4 бита информации, а 7 - что для этого мы к 4 информационным битам добавим 3 проверочных. Итого всего получается 7 бит.

## О кодах исправляющих несимметричные ошибки.

На практике нередко встречаются каналы отличающиеся асимметричный характером ошибок, скажем такие, в которых преобладают замещения вида 0 -> 1 (т.е вместо нуля принимается единица), а замещения 1->0 крайне редки

Конечно, и в этом случае можно использовать коды, предназначенный для исправления замещений обоих видов, в частности, рассмотренный нами код Хемминга. Но это било бы слишком расточительно, так как корректирующая способность кода наполовину расходовалась бы тогда вхолостую. Придуманы поэтому коды, приспособленные специально к исправлению несимметричных ошибок.

Исходное соображение здесь очень простое: если в канале невозможны ошибки вида 1 -> 0, то в принятом двоичном слове нет нужды проверять позиции с нулевыми символами – они наверняка переданы без искажений. Будем поэтому производить проверку таким образом , чтобы ее результат зависел только от позиций с единичными символами, точнее , от номеров этих позиций. С этой целью для произвольного двоичного слова u=x1x2…xn составим сумму

(1)

В сумме (1) не нулевые слагаемые соответствуют только единичным символам и каждое из них совпадает с номером этого символа, т. е. число S (u) равно сумму номеров единичных позиций слова u.

Как обычно, постараемся найти простое условие, выделяющее кодовые слова среди прочих. Будем искать это условие в виде сравнения по некоторому модулю l.Представим себе, что число l уже выбрано, и рассмотрим код, состоящий из всех таких слов v = x1 x2 ... xn, для которых  (2)

т.е. из слов, для которых сумма номеров единичных позиций делится на l без остатка. Обозначим указанный код через Vn,l. Так, при n = 4, l=5 получим следующее множество кодовых слов:



Нетрудно убедиться, хотя бы перебором всех возможных случаев, что данный код исправляет любые одиночные ошибки вида 0->1. Например, ошибка в третьем символе слова 1001 переводит его в слово 1011.При этом никакое другое кодовое слово не могло преобразоваться в результате одиночной ошибки в слове 1011. Поэтому получатель декодирует слово 1011 однозначно, считая, что было послано слово 1001. Аналогично обстоит дело с другими двоичными наборами, содержащими одиночные замещения нуля на единицу. Отметим, что в некоторых случаях (но не всегда!) возможно исправление также и двойных замещений нуля на единицу. Один из таких случаев – ошибка в двух первых символах слова 0000. Получающееся при этом слово 1100 не является кодовым и не могло получиться ни из какого кодового слова в результате одиночной ошибки. К тому же имеется единственный вариант двойного замещения, переводящего кодовое слово в слово 1100, - именно тот, который был указан выше.

Подобные свойства сохраняются и для произвольного кода Vn,l, если l >= n+1. В частности такой код исправляет любые одиночные ошибки вида 0->1; при этом алгоритм их исправления чрезвычайно прост. Действительно, пусть в кодовом слове v =x1 x2 … xn произошло не более одной ошибки и получилось слово u. Если ошибка произошла в j-м символе, то понятно, что S(u)=S(v)+j. Поскольку т. е. номеру позиции, в которой произошла ошибка. Если же ошибки не произошло, то вычет S(u) по модулю l равен нулю.

Проиллюстрируем исправление одиночной ошибки на примере рассмотренного выше кода V4,5. Пусть принято слово u= 0111. Тогда S(u) = 2 + 3 + 4 = 9 => 4 (mod 5). Следовательно, ошибка произошла в четвертой позиции, т.е. передавалось кодовое слово 0110.

Наиболее употребимы кода Vn,l при минимально возможном l : l = n + 1. Именно они впервые были предложены советскими специалистами по кодированию Р. Р. Варшамовым и Г. М. Тененгольцем.

Коды Vn,l обладают способностью исправлять еще один тип искажений кодовых слов, характерный для несимметричных каналов. Это так называемые выпадения и вставки символов.

Предположим, что в некотором слове v= x1x1…xn произошло выпадение одного символа, в результате чего получилось слово u=y1y2…yn-1 длинны n -1. Пусть n1 – число единиц, а n0 – число нулей, расположенных правее выпавшего символа. Оказывается, числа n1 и n0 могут быть определены с помощью суммы



В самом деле, если выпал символ 0, то S(v) – S(u) = n1, а если выпал символ 1, то S(v) – S(u) = n – n0.Если w(u) – вес слова u, то, очевидно,  (3)

Так как S(v)=(0) (mod l), то вычет числа – S(u) по модулю l равен либо n1 (в случае выпадения нуля), либо n – n0 (в случае выпадения единици). Неравенства (3) показывают, что если этот вычет не превосходит w(u), то выпал выпавший символ 0 надо вставить в слово u так, чтобы правее него было число единиц, равное вычету числа – S(u) по модулю l.Если же этот вычет больше, чем w(u),то вставляем в слово u единицу так , чтобы правее нее было число нулей, равное разности n и вычета числа – S(u).Если, например, при использовании кода V4,5 принято слово u = 101, то S(u) = 4, а w(u) = 2, вычет числа – S(u) по модулю 5 равен 1, и , следовательно , выпал символ 0. Вставить его надо так, чтобы правее него была одна единица.Получается кодовое слово 1001.

## Реализация кода Хэмминга

### Подготовка.

Допустим, у нас есть сообщение «0100010000111101», которое необходимо передать без ошибок.

Далее нам 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 (взяты в скобки):

Было:

0100010000111101

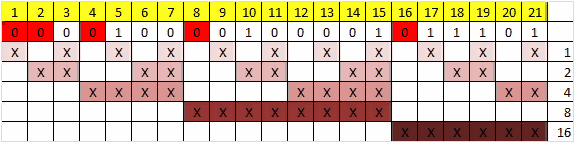
Стало:

(0)(0)0(0)100(0)0100001(0)11101

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, 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.



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).  
Высчитав контрольные биты для нашего информационного слова получаем следующее:

Было:

(0)(0)0(0)100(0)0100001(0)11101

Стало:

(1)(0)0(1)100(0)0100001(0)11101

### Декодирование и исправление ошибок.

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

(1)(0)0(1)100(0)01\*1\*0001(0)11101

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

(0)(1)0(1)100(1)01\*1\*0001(0)11101

Как мы видим, контрольные биты под номерами: 1, 2, 8 не совпадают с такими же контрольными битами, которые мы получили. 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

### Вывод.

С помощью кода Хэмминга мы можем не только находить ошибку но и исправлять ее.

## Бит четности.

В [вычислительной технике](https://dic.academic.ru/dic.nsf/ruwiki/627834) и [сетях передачи данных](https://dic.academic.ru/dic.nsf/ruwiki/214719) би́том чётности называют контрольный [бит](https://dic.academic.ru/dic.nsf/ruwiki/4904), принимающий значения '0' или '1' и служащий для проверки общей чётности двоичного числа (чётностиколичества единичных битов в числе).

### Примеры

Бит чётности или контрольный разряд формируется при выполнении операции «Исключающее - ИЛИ» последовательность (1 бит с 2, результат с 3, результат с 4 и т.д.).

Рассмотрим схему, использующую девятибитные кодовые слова, состоящие из восьми бит данных, за которыми следует бит четности.

* Число 10111101 содержит 6 '1' битов. Бит чётности будет 0, получаем кодовое слово 101111010.
* Число 01110011 содержит 5 '1' битов. Бит чётности будет 1, получаем кодовое слово 011100111.
* Число 00000000 содержит 0 '1' битов. Бит чётности будет 0, получаем кодовое слово 000000000.

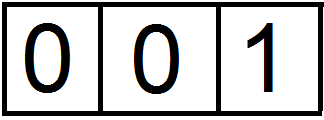
Пустой или несуществующий поток битов также имеет ноль единичных битов, поэтому бит четности будет ‘0’.

### Вывод.

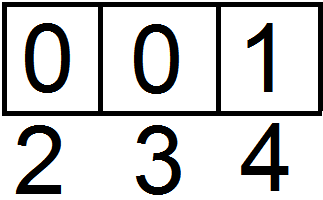
C помощью бита четности, мы можем проверить была ли совершена ошибка при передаче данных.

## Мой собственный алгоритм или алгоритм контрольной суммы.

Допустим у нас есть сообщение «001», которое нам нужно передать, оно будет размешаться в некотором блоке.

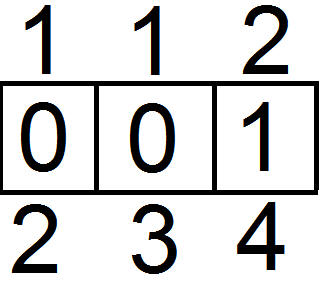


Но за нахождение в также у блока присутствует «стоимость за нахождение в нем», допустим блок[0] = 2 , блок [1] = 3 и так далее . Блок[i] = 2+i;



Но так же у элементов находящихся в блок присутствует некоторая стоимость.

У нуля это единица, у единицы это двойка.



И путем перемножения мы получаем «контрольную сумму» данного сообщения в данном случаи = (1\*2) + (1\*3) + (2\*4) = 13.

### Плюсы и минусы алгоритма.

Плюсы - достаточно легкая и понятная реализация, при многократном использовании можно создать «словарь» в который записывается контрольная сумма и полученное сообщение, то есть когда в нашем словаре будет достаточно данных мы можем восстанавливать сообщение по контрольной сумме.

Минусы - непонятно как лучше отправлять контрольную либо отдельным сообщением, либо вместе с сообщением, непонятно как лучше переводить сумму в двоичную систему или оставлять в десятичной.

### Выводы.

Алгоритм не достаточно хорош, но работает, да и ладно.

# Проектная часть.

## Функционал

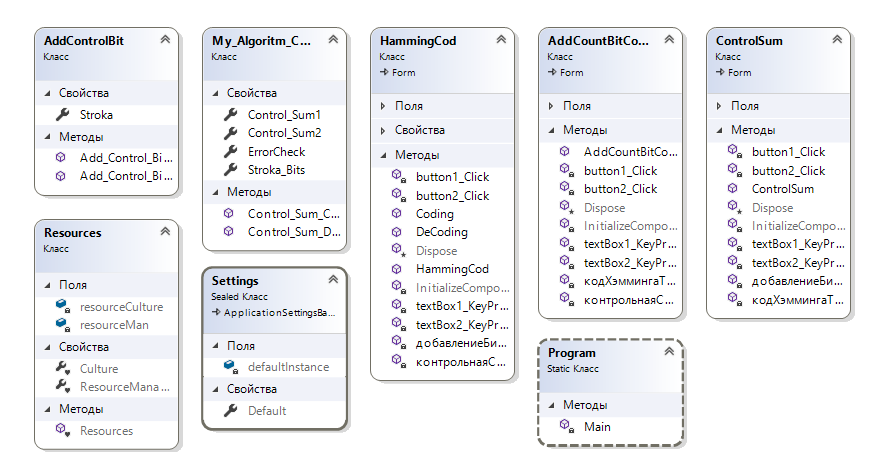
Программа определяет была ли совершена ошибка, при передаче данных.

Если вы используете алгоритм Хэмминга, то еще и исправляет ошибку.

## Архитектура приложения

Программа состоит из двух частей: из Интерфейса Windows Forms и классов в которых находятся сами алгоритмы.

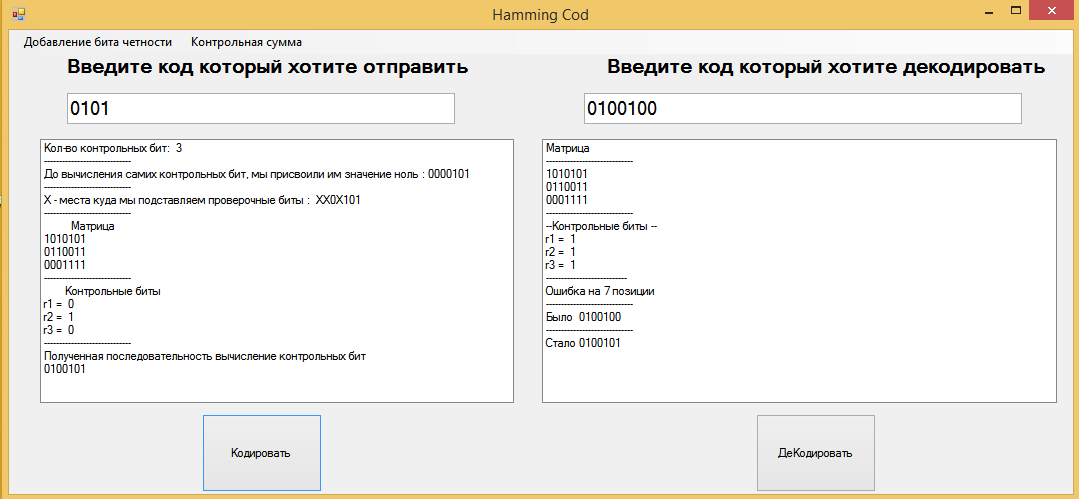
## Диаграмма классов



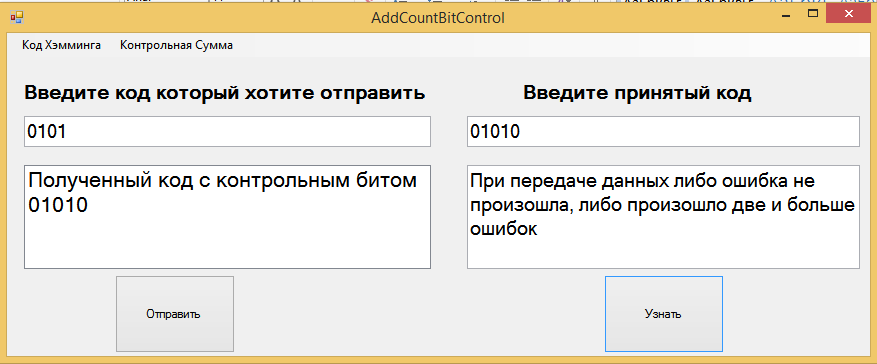
## Реализация проекта

Интерфейс программы:

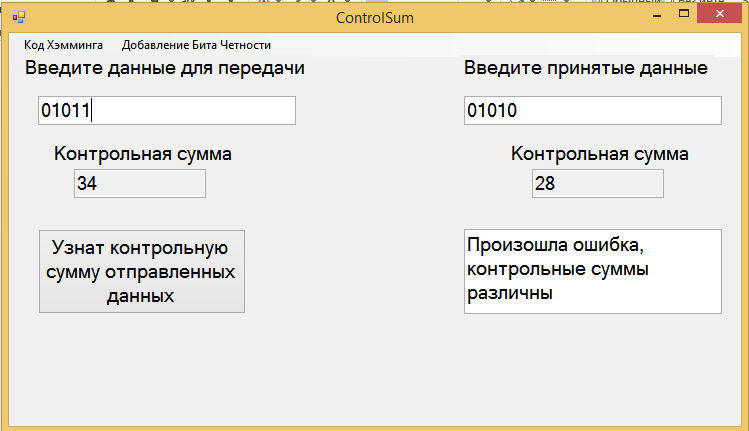
Код Хэмминга:



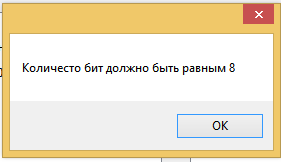
Бит четности

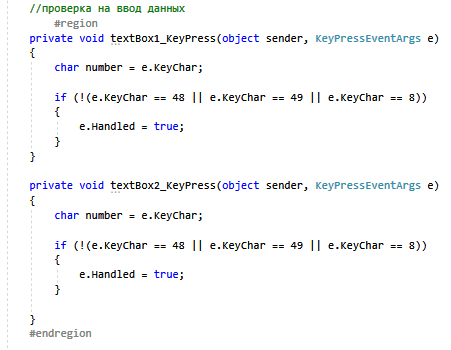


Контрольная сумма:



Так же в приложении присутствуют обработчики ошибок





Среда разработки:

Microsoft Visual Studio

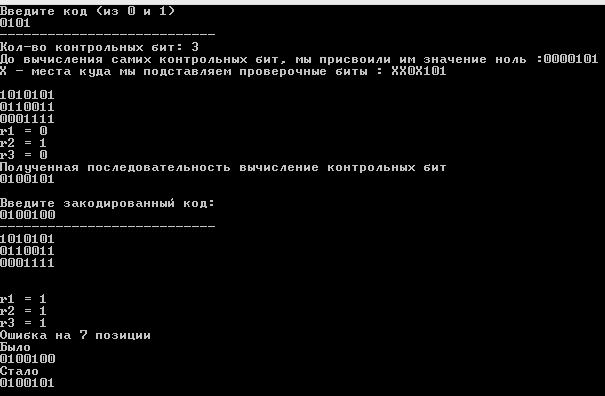
Инструменты:

C# + .NET

## Хронология разработки проекта

Для удобства хранения и контроля версиями использовался такой сервис, как GitHub.

6 декабря 2018 года была разработана тестовая версия кода Хэмминга в консольном приложении

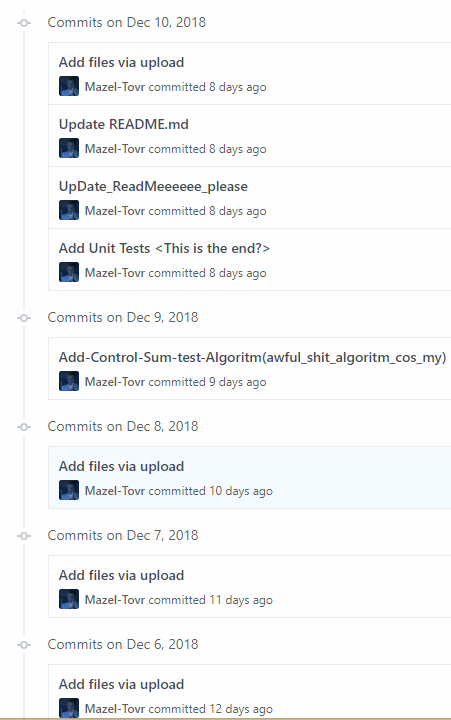


7 декабря к коду Хэмминга была добавлена Windows Forms

8 декабря был добавлен алгоритм «Добавление бита четности»

9 декабря был разработан собственный алгоритм «Контрольной суммы» , исправленный некоторые баги , добавленный обработчики

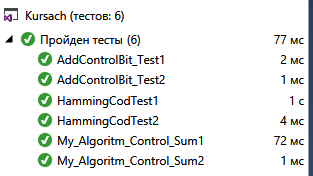
10 декабря были добавлены Unit Tests.

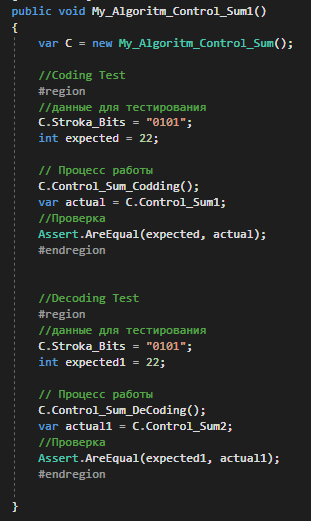


## Тестирование

Для автоматического тестирования я добавил в проект UnitTestProject.

Результаты тестов:





Все тесты успешно пройдены.

# Заключение

В данной курсовой работе я реализовал 3 алгоритма поиска ошибок при передаче текста, также я получил представление о том, что такое самокорректирующийся код, изучил алгоритм кодирования и декодирования кодом Хэмминга.

Проект был успешно протестирован.

# Источники

Код Хэмминга <https://habr.com/post/140611/>

Бит четности <https://ru.wikipedia.org/wiki/Бит_чётности>

М.Н.Аршинов, Л.Е.Садовский. КОДЫ И МАТЕМАТИКА