**Министерство науки и высшего образования Российской федерации**

**ФГБОУ ВО «Магнитогорский государственный технический университет им. Носова»**

Кафедра информатики и информационной безопасности

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы шифрования информации»

на тему: «Исследование качества алгоритмов шифрования»

Исполнитель: Шпак В.А. студент 4 курса, группа АИБ-17

Руководитель: Коновалов М.В. доцент каф. ИиИБ, к.т.н.

Работа допущена к защите "\_\_\_\_\_" \_\_\_\_\_\_\_\_\_ 2021г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Работа защищена "\_\_\_\_\_" \_\_\_\_\_\_\_\_\_ 2021г. с оценкой \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

Магнитогорск, 2021

**Министерство науки и высшего образования Российской федерации**

**ФГБОУ ВО «Магнитогорский государственный технический университет им. Носова»**

Кафедра информатики и информационной безопасности

**ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ**

Тема: «Исследование качества алгоритмов шифрования»

Студенту Шпак Виталию Алексеевичу

Исходные данные: Научная литература, учебные материалы, интернет.

Срок сдачи: «\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021г

Руководитель:             Коновалов М.В.              /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

Задание получил:              Шпак В.А.                /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

Магнитогорск, 2021

Содержание

[Введение 4](#_Toc73985034)

[1 Информационная энтропия 5](#_Toc73985035)

[2 Алгоритмы шифрования информации 11](#_Toc73985036)

[2.1 Шифры перестановки. Алгоритм блочной одинарной перестановки 11](#_Toc73985037)

[2.2 Шифры гаммирования. Регистр сдвига с линейной обратной связью 12](#_Toc73985038)

[2.3 Шифры замены. Шифр Хилла 17](#_Toc73985039)

[3 Программная реализация алгоритмов 21](#_Toc73985040)

[3.1 Общая структура приложения 21](#_Toc73985041)

[3.2 Семейство классов Palette 26](#_Toc73985042)

[3.2.1 Класс Palette 26](#_Toc73985043)

[3.2.2 Класс ImageInfo 28](#_Toc73985044)

[3.3 Семейство классов шифрования 29](#_Toc73985045)

[3.3.1 Класс Crypto 29](#_Toc73985046)

[3.3.2 Класс Asymmetric 29](#_Toc73985047)

[3.3.3 Класс HillCipher 30](#_Toc73985048)

[3.3.4 Класс Symmetric 33](#_Toc73985049)

[3.3.5 Класс SinglePermutation 34](#_Toc73985050)

[3.3.6 Класс Gamma 35](#_Toc73985051)

[3.4 Интерфейс приложения. Алгоритм работы приложения 37](#_Toc73985052)

[4 Сравнение качества шифрования алгоритмов 42](#_Toc73985053)

[Заключение 47](#_Toc73985054)

[Список использованных источников 48](#_Toc73985055)

[Приложение А 49](#_Toc73985056)

# Введение

При проектировании шифров первостепенным является вопрос обеспечения их стойкости. Этот проблема одновременно является и наиболее сложной. Оценка стойкости является одним из наиболее длительных, трудоемких и разноплановых этапов. При этом важным является, творческий подход, поскольку в настоящее время нет законченной теории блочных шифров, которая бы позволила выработать полноценную методику оценивания стойкости.

Однако уже предложены некоторые общие требования к качеству шифрующих преобразований. Если шифр удовлетворяет таким требованиям, то говорят о доказуемой стойкости (к известным методам криптоанализа) или о достижении гарантированных свойств шифрующих преобразований

Самым простым индикатором качества шифрования является энтропия *H*, рассчитанная по формуле Шеннона. Максимальное значение энтропии определяется по формуле Хартли *Hmax*. Чем ближе *Н* к *Нmax,* тем качественнее зашифрована информация.

В ходе выполнения данной работы необходимо спроектировать и разработать приложение для выполнения шифрования восьмибитного графического изображения в формате BMP при помощи шифров блочной одинарной перестановки, Хилла и гаммирования на основе регистра сдвига с линейной обратной связью.

Основной задачей курсовой работы является определение параметров выбранных алгоритмов шифрования для получения наилучшего результата на основе графиков зависимости энтропии *Н* от значений параметров алгоритмов шифрования.

# Информационная энтропия

Понятие информационной энтропии определено Шенноном для случая дискретных данных, и похоже на понятие термодинамической энтропии. Это - величина, обозначающая количество информации, содержащееся в данном сообщении (или последовательности сигналов).

По Шеннону информация снятая неопределенность. Точнее получение информации - необходимое условие для снятия неопределенности. Неопределенность возникает в ситуации выбора. Задача, которая решается в ходе снятия неопределённости – уменьшение количества рассматриваемых вариантов (уменьшение разнообразия), и в итоге выбор одного соответствующего ситуации варианта из числа возможных. Снятие неопределенности даёт возможность принимать обоснованные решения и действовать. В этом управляющая роль информации.

Чем меньше вероятность какого-либо события, тем большую неопределенность снимает сообщение о его появлении и, следовательно, тем большую информацию оно несёт.

С точки зрения криптографии, энтропия определяет количество символов, которые необходимо раскрыть, чтобы узнать содержание сообщения. Так, если некоторый 8-битовый блок данных хранит одно из двух возможных сообщений (например, ответы "Да" или "Нет"), то достаточно правильно узнать один бит, чтобы определить значение исходного сообщения. Сколько бы бит не было отведено для шифрования слов "Да" и "Нет", энтропия или неопределенность всегда будет меньше или равна 1 [1]. Таким образом, чем выше энтропия в зашифрованном сообщении, тем качественнее оно зашифровано, и наоборот.

Сведения об информационной энтропии необходимы для повышения надёжности передачи сигналов. Именно на неё ориентируются при задании избыточной информации, передаваемой по линии связи.

Информационная энтропия - мера хаотичности информации или мера внутренней неупорядоченности информационной системы. Энтропия увеличивается при хаотическом распределении информационных ресурсов и уменьшается при их упорядочении.

Избыточность - термин из теории информации, означающий превышение количества информации, используемой для передачи или хранения сообщения, над его информационной энтропией. Для уменьшения избыточности применяется сжатие данных без потерь, в то же время контрольная сумма применяется для внесения дополнительной избыточности в поток, что позволяет производить исправление ошибок при передаче информации по каналам, вносящим искажения (спутниковая трансляция, беспроводная передача и т. д.).

Впервые понятия энтропия и информация связал Клод Шеннон в 1948. С его подачи энтропия стала использоваться как мера полезной информации в процессах передачи сигналов по проводам. Следует подчеркнуть, что под информацией Шеннон понимал сигналы нужные, полезные для получателя. Неполезные сигналы, с точки зрения Шеннона, это шум, помехи. Если сигнал на выходе канала связи является точной копией сигнала на входе, то это означает отсутствие энтропии. Отсутствие шума означает максимум информации.

Величина, характеризующая количество неопределенности в теории информации, обозначается символом H и имеет название энтропия, точнее информационная энтропия. Энтропия (H) – мера неопределенности, выраженная в битах. Так же энтропию можно рассматривать как меру равномерности распределения случайной величины.

На Рис. 1 показано поведение энтропии для случая двух альтернатив, при изменении соотношения их вероятностей *(p, (1 - p))*. Максимального значения энтропия достигает в данном случае тогда, когда обе вероятности равны между собой и равны 0,5, нулевое значение энтропии соответствует случаям *(p0=0, p1=1)* и *(p0=1, p1=0)*.

Ситуация максимальной неопределенности предполагает наличие нескольких равновероятных альтернатив (вариантов), т.е. ни один из вариантов не является более предпочтительным. Причём, чем больше равновероятных вариантов наблюдается, тем больше неопределенность, тем сложнее сделать однозначный выбор и тем больше информации требуется для этого получить. Для N вариантов эта ситуация описывается распределением вероятностей: *{1/N, 1/N, … 1/N}*. Минимальная неопределенность равна *0*, т.е. эта ситуация полной определенности, означающая что выбор сделан, и вся необходимая информация получена. Распределение вероятностей для ситуации полной определенности выглядит так: *{1, 0, …0}*.

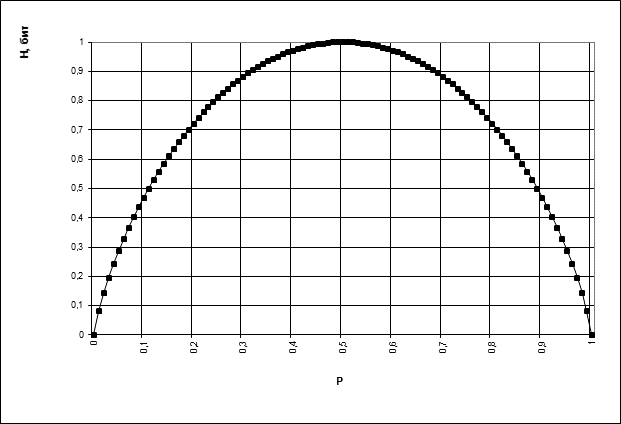


Рисунок 1 – Поведение энтропии для случая двух альтернатив

В 1948, исследуя проблему рациональной передачи информации через зашумлённый коммуникационный канал, Шеннон предложил вероятностный подход к пониманию коммуникаций и создал истинно математическую теорию энтропии. Его идеи послужили основой разработки двух направлений: теории информации, которая использует понятие вероятности и эргодическую теорию для изучения статистических характеристик данных и коммуникационных систем, и теории кодирования, в которой используются алгебраические и геометрические инструменты для разработки эффективных кодов.

Как уже упоминалось, под информационной энтропией понимают меру хаотичности информации. Можно определить энтропию случайной величины, введя предварительно понятия распределения случайной величины X, имеющей конечное число значений:

(1)

где *x X* – атом (i-ый символ сообщения)

*pi* – вероятность появления атома

– распределение атомов

Так же для введения понятия энтропии необходимо определить собственную информацию:

(2)

где I(X) – собственная информация

Собственная информация - статистическая функция дискретной случайной величины. Она является случайной величиной Собственную информацию можно понимать как «меру неожиданности» события - чем меньше вероятность события, тем больше информации оно содержит.

Исходя из определения собственной информации энтропия по Шеннону определяется как:

(3)

где H(X) – информационная энтропия сообщения

p(i) – вероятность появления i-го символа в сообщении

Информационная энтропия для независимых случайных событий *x* с *n* возможными состояниями (от 1 до *n*) в двоичной системе счисления рассчитывается по формуле:

(4)

Энтропия события *x* является суммой с противоположным знаком всех произведений относительных частот появления события i, умноженных на их же двоичные логарифмы (основание 2 выбрано только для удобства работы с информацией, представленной в двоичной форме).

Аналогично, для случая, когда каждый из символов сообщения встречается единожды, информационная энтропия в двоичной системе счисления по Хартли рассчитывается по формуле:

(5)

Шеннон предположил, что прирост информации равен утраченной неопределённости, и задал требования к её измерению:

* мера должна быть непрерывной; т. е. изменение значения величины вероятности на малую величину должно вызывать малое результирующее изменение функции;
* в случае, когда все варианты равновероятны, увеличение количества вариантов (букв) должно всегда увеличивать значение функции;
* должна быть возможность сделать выбор в два шага, в которых значение функции конечного результата должно являться суммой функций промежуточных результатов.

Мера энтропии Шеннона выражает неуверенность реализации случайной переменной. Таким образом, энтропия является разницей между информацией, содержащейся в сообщении, и той частью информации, которая точно известна (или хорошо предсказуема) в сообщении. Примером этого является избыточность языка - имеются явные статистические закономерности в появлении букв, пар последовательных букв, троек и т. д.

Общие свойства энтропии:

1. Неотрицательность: *H(X) ≥ 0*.
2. Ограниченность: . Равенство, если все элементы из X равновероятны.
3. Если *X, Y* независимы, то *H(XY) = H(X) + H(Y)*.
4. Энтропия - выпуклая вверх функция распределения вероятностей элементов.
5. Если *X, Y* имеют одинаковое распределение вероятностей элементов, то *H(X) = H(Y)* [2].

# Алгоритмы шифрования информации

Различается шифрование двух типов: симметричное и асимметричное. При симметричном шифровании создается ключ, с его помощью отправитель шифрует сообщение и пересылает результат адресату, а ключ (пароль или другой файл данных) передает отдельно по закрытому каналу связи. Адресат сможет прочитать шифротекст, расшифровав его полученным ключом. Симметричное шифрование не так надежно, как асимметричное, поскольку ключ может быть перехвачен, но из-за высокой скорости оно широко используется в операциях электронной торговли.

Асимметричное шифрование сложнее — и надежнее. Для него нужны два взаимосвязанных ключа: открытый и закрытый. Свой открытый ключ адресат сообщает всем желающим. Он позволяет шифровать данные, но не расшифровывать их. Закрытый ключ есть только у адресата. Когда кому-то нужно послать адресату зашифрованное сообщение, он выполняет шифрование, используя открытый ключ. Получив сообщение, адресат расшифровывает его с помощью своего закрытого ключа. Однако поскольку вычисления в этом случае сложнее, процедура отнимает больше времени [3].

## Шифры перестановки. Алгоритм блочной одинарной перестановки

Шифр перестановки осуществляет преобразование перестановки букв в открытом тексте. Обычно открытый текст разбивается на отрезки равной длины и каждый отрезок шифруется независимо. Пусть, например, длина отрезков равна *n* и *σ* - взаимно-однозначное отображение множества в себя. Тогда шифр перестановки действует так: отрезок открытого текста *x1… xn* преобразуется в отрезок шифрованного текста x*σ(1)*… x*σ(n).*

При использовании шифра блочной одинарной перестановки задается таблица перестановки блока символов, которая последовательно применяется до тех пор, пока исходное сообщение не закончится. Если исходное сообщение не кратно размеру блока, тогда оно при шифровании дополняется произвольными символами. Исходное сообщение разбивается на блоки длины *m*, где *m* – это длина ключа [4]. Ключ в шифре перестановки имеет вид, указанный на рисунке 2.

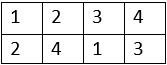


Рисунок 2 – Ключ шифра блочной одинарной перестановки

В первой строке таблицы указаны номера символов блока по порядку, а во второй строке указаны номера позиций, которые должны занимать указанные символы в зашифрованном блоке текста.

Кодирование осуществляется перестановкой букв. Таким образом первый символ из исходного блока должен быть переставлен на второе место, второй на четвертое, третий на первое, четвертый на третье.

Если данным ключом зашифровать слово «кофе», то получится слово «фкео».

Дешифрование производится в обратном порядке. На примере указанного ключа: второй символ из зашифрованного блока ставим на первое место, четвертый на второе, первый на третье, третий на четвертое.

При использовании любого блочного шифра (шифр перестановки не исключение), может возникнуть ситуация, когда текст не делится на равные блоки длины m. То есть остаток от деления длины текста n на длину ключа m не равен нулю.

В таких случаях длину исходного сообщения увеличивают на *m – (n % m)* символов, чтобы оно делилось на равные блоки длины m [5].

## Шифры гаммирования. Регистр сдвига с линейной обратной связью

Шифры гаммирования (аддитивные шифры) являются самыми эффективными с точки зрения стойкости и скорости преобразований (процедур зашифрования и дешифрования). По стойкости данные шифры относятся к классу совершенных. Для зашифрования и дешифрования используются элементарные арифметические операции – открытое/зашифрованное сообщение и гамма, представленные в числовом виде, складываются друг с другом по модулю.

В литературе шифры этого класса часто называют потоковыми, хотя к потоковым относятся и другие разновидности шифров. В шифрах гаммирования может использоваться сложение по модулю N (общий случай) и по модулю 2 (частный случай, ориентированный на программно-аппаратную реализацию).

Сложение по модулю N. В 1888 г. француз маркиз де Виари в одной из своих научных статей, посвященных криптографии, доказал, что при замене букв исходного сообщения и ключа на числа справедливы формулы:

*Ci = (Pi + Ki) mod N,* (6)

*Pi = (Ci + N - Ki) mod N* (7)

где *Pi* – *i*-ый символ открытого сообщения;

*Ci - i*-ый символ шифрованного сообщения;

*N* - количество символов в алфавите;

*Кi - i*-ый символ гаммы (ключа).

Стойкость аддитивных шифров определяется, главным образом, качеством гаммы, которое зависит от длины периода и случайности распределения по периоду.

Длиной периода гаммы называется минимальное количество символов, после которого последовательность начинает повторяться. Случайность распределения символов по периоду означает отсутствие закономерностей между появлением различных символов в пределах периода.

Для обеспечения абсолютной стойкости необходимо, чтобы последовательность символов в пределах периода гаммы обладала следующими свойствами:

* была случайной (должна отсутствовать закономерность в появлении символов гаммы);
* символы алфавита гаммы были распределены нормально (равновероятно);
* совпадала по размеру или была больше исходного открытого текста;
* применялась только один раз.

Несмотря на все достоинства шифров гаммирования, одной из ключевых проблем их применения на практике является получение качественных гамм. Для процедур зашифрования/дешифрования можно использовать истинно случайные или псевдослучайные гаммы (последовательности). Псевдослучайная последовательность – последовательность чисел, которая была вычислена по определённой процедуре, но имеет все свойства случайной последовательности чисел в рамках решаемой задачи. Отличие истинно случайных последовательностей от псевдослучайных заключается в невозможности предсказания (расчета, определения) символов в ней. Таким образом, любой алгоритмически устроенный программно-аппаратный комплекс не может генерировать истинно случайные последовательности, т.к. он работает по строго определенным правилам, а значит результат (гамма) предсказуем.

Истинно случайные гаммы могут быть получены путем оцифровки случайных физических или антропогенных процессов.

Для большинства генераторов на базе физических и антропогенных процессов характерны определенные недостатки:

* необходимость передачи гамм всем участникам обмена данными. Так как гамма генерируется в одном месте, и она случайна, то принимающая данные сторона для расшифровки предварительно должна получить эту гамму (проблема обмена ключами);
* медленная скорость генерации числовых последовательностей;
* антропогенные процессы могут иметь скрытые зависимости – каждый пользователь обладает своим «подчерком» работы с компьютером.

Псевдослучайные гаммы получают путем применения рекуррентных формул вида *an = F(n, an-1, an-2, …, an-p)*, определяющая каждый член последовательности *an*, как функцию от *p* предыдущих членов и возможно номера члена последовательности *n*, или полноценных алгоритмов. При этом отсутствие истинной случайности не мешает получать криптографически стойкие последовательности, в т.ч. и с бесконечным периодом. Генераторы псевдослучайных последовательностей получили наибольшее распространение.

Внутреннее состояние описывает текущее состояние генератора гаммы. Начальное внутреннее состояние, как правило, определяется ключом K. Два генератора, с одинаковым ключом и одинаковым внутренним состоянием, создают одинаковые гаммы. Функция переходов считывает текущее внутреннее состояние и генерирует новое внутреннее состояние. Выходная функция считывает внутреннее состояние и генерирует бит (биты) гаммы *Ki*.

Для генерации псевдослучайных последовательностей используют рекуррентные формулы или полноценные алгоритмы. В первом случае члены числовой последовательности не только рассчитываются рекуррентно, но и впоследствии становятся частью гаммы. Во втором случае для генерации гаммы используются более сложные правила, в т.ч. для повышения энтропии гаммы могут применять хеш-функции и шифрование.

В текущей работе для генерации гаммы используется Регистр сдвига с линейной обратной связью (РСЛОС). РСЛОС – упорядоченный набор битов, у которого значение входного (вдвигаемого слева, старшего) бита равно линейной булевой функции от значений остальных битов регистра до сдвига. Теорию последовательности регистров сдвига разработал в 1965 г. главный криптограф норвежского правительства Эрнст Селмер.

Длиной регистра называется количества битов в нем. В качестве булевой функции для РСЛОС чаще всего используют сложение по модулю 2. Такие РСЛОС обычно записывают в виде многочлена (полинома) или упорядоченной последовательности. Запись *x8 + x4 + x3 + x2 + 1* или (8, 4, 3, 2, 0) означает, что длина регистра 8 битов, а входной бит рассчитывается по формуле *b7 = b4 ⊕ b3 ⊕ b2 ⊕ b0* (для битов нумерация начинается справа и с нулевой позиции). Выходной (выдвигаемый справа, младший) бит будет являться частью генерируемой гаммы. Расчет требуемой гаммы осуществляется в цикле, на каждой итерации которого выполняются следующие операции:

Добавление выходного бита *b0* к гамме;

Расчет входного бита *bn-1* по формуле;

Сдвиг битов регистра вправо на одну позицию;

Занесение рассчитанного входного бита *bn-1* в позицию *n-1*.

В таблице 1 приведен пример генерации гаммы для регистра, инициализированного значением 100011002.

Таблица 1 – Пример генерации гаммы для РСЛОС *x8 + x4 + x3 + x2 + 1*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № п/п | Исходное состояние регистра | Бит гаммы, b0 | Входной бит, b7 = b4 ⊕ b3 ⊕ b2 ⊕ b0 | Сдвиг регистра вправо на одну позицию | Занесение входного бита b7 в регистр |
| 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | | 0 | 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | |
| 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | | 0 | 0 ⊕ 0 ⊕ 1 ⊕ 0 = 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 0 | 0 | 0 | 1 | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | |
| 2 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | | 1 | 0 ⊕ 0 ⊕ 0 ⊕ 1 = 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 0 | 1 | 0 | 0 | 0 | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | |
| 3 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | | 1 | 1 ⊕ 0 ⊕ 0 ⊕ 1 = 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 1 | 0 | 1 | 0 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | |
| 4 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | | 0 | 0 ⊕ 1 ⊕ 0 ⊕ 0 = 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 0 | 1 | 1 | 0 | 1 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | |
| **...** | **...** | **...** | **...** | **...** | **...** |
| Гамма | | 00110... |  | | |

Для регистра длиной n максимальное количество состояний составляет *2n–1*. Это число равно *2n–1*, а не *2n*, поскольку заполнение РСЛОС нулями влечет вывод регистром бесконечной последовательности нулей, что совершенно бесполезно. Таким образом, максимально возможная длина периода гаммы в битах составляет *2n–1*, если из регистра для гаммы на каждой итерации брать один бит. Для получения такого максимального периода многочлен должен быть примитивным по модулю 2. Примитивный многочлен степени *n* – неприводимый многочлен, который является делителем , но не является делителем *xd+1* для всех *d*, являющихся делителями *2n–1*. Чем больше битов регистра используются для расчета входного бита, тем больше повышается стойкость.

РСЛОС обладают высокой скоростью генерации чисел, хорошими статистическими свойствами, а также возможностью простой реализации на аппаратном уровне. Исследователями предложено несколько десятков различных вариантов реализации генераторов на базе РСЛОС, в т.ч. с применением комбинации нескольких РСЛОС одновременно и разных функций расчета входного или выходного бита (регистры сдвига с нелинейной обратной связью, обобщенной обратной связью, обратной связью по переносу и т.д.) [6].

## Шифры замены. Шифр Хилла

Сущность шифрования методом замены заключается в следующем. Пусть шифруются сообщения на русском языке и замене подлежит каждая буква этих сообщений. Тогда, букве «А» исходного алфавита сопоставляется некоторое множество символов (шифрозамен) *МА*, «Б» – *МБ*, …, «Я» – *МЯ*. Шифрозамены выбираются таким образом, чтобы любые два множества (*МI* и *МJ*, *i ≠ j*) не содержали одинаковых элементов (*МI ∩ МJ = Ø*).

При шифровании каждая буква «А» открытого сообщения заменяется любым символом из множества *МА*. Если в сообщении содержится несколько букв «А», то каждая из них заменяется на любой символ из *МА*. За счет этого с помощью одного ключа можно получить различные варианты шифрограммы для одного и того же открытого сообщения. Так как множества *МА, МБ, ..., МЯ* попарно не пересекаются, то по каждому символу шифрограммы можно однозначно определить, какому множеству он принадлежит, и, следовательно, какую букву открытого сообщения он заменяет. Поэтому расшифрование возможно и открытое сообщение определяется единственным образом.

Приведенное выше описание сущности шифров замены относится ко всем их разновидностям за исключением полиалфавитных шифров, в которых для зашифрования разных символов исходного алфавита могут использоваться одинаковые шифрозамены (т.е. *МI ∩ МJ ≠ Ø, i ≠ j*).

Первый практически реализуемый способ шифрования с использованием алгебры был придуман в 1929 г. математиком Лестером Хиллом - профессором из Хантер-колледжа в Нью-Йорке, статья которого «Cryptography in an Algebraic Alphabet» была опубликована в журнале «The American Mathematical Monthly».

Для преобразования сообщения шифром Хилла каждому символу алфавита длинной *N* сопоставляется число. Желательно, чтобы длина алфавита *N* была простым числом, т.е. числом, которое делится нацело только на себя и на 1. Это не обязательно, но очень удобно, потому что для расшифровки необходимо, чтобы детерминант ключа и длина алфавита были взаимно простыми, т.е. не имели общих делителей кроме 1. Если длина алфавита – простое число, то таких ключей, для которых выполняется это условие значительно больше.

Для зашифрования сообщение делится на блоки каждый по *n* символов. Каждый блок исходного сообщения рассматривается как *n*-мерный вектор чисел и умножается на матрицу-ключ зашифрования размером *n*×*n* по модулю *N*. Полученный *n*-мерный вектор – блок зашифрованного сообщения размера *n* символов. Описанные операции проводятся для каждого блока исходного сообщения.

где *p* – *i*-ый элемент блока исходного сообщения;

*kij* – *i*-ый, *j*-ый элемент матрицы ключа зашифрования;

*ci* – *i*-ый элемент блока зашифрованного сообщения.

Для расшифрования применяется обратная матрица по модулю. Матрица *A−1* называется обратной по отношению к квадратной матрице A, если выполнено условие *A−1A=AA−1=E*, где E – единичная матрица, порядок которой равен порядку матрицы A.

Невырожденная матрица – матрица, определитель которой не равен нулю. Соответственно, вырожденная матрица – та, у которой равен нулю определитель. Обратная матрица *A−1* существует тогда и только тогда, когда матрица *A* – невырожденная. Если обратная матрица *A−1* существует, то она единственная.

Пусть задана матрица . Для того, чтобы найти обратную матрицу A−1, требуется осуществить три шага:

1. Найти определитель матрицы *А* и убедиться, что , т.е. что матрица *А* – невырожденная.
2. Составить алгебраические дополнения  каждого элемента матрицы *A* и записать матрицу  из найденных алгебраических дополнений.
3. Записать обратную матрицу с учетом формулы .

Матрицу  часто именуют присоединённой (взаимной, союзной) к матрице *A.* Обратная матрица по модулю *N* находится по формуле:

где – обратная матрица по модулю N;

– союзная матрица к матрице A;

– определитель матрицы А;

– обратный определитель матрицы А в кольце по модулю N.

Обратный детерминанту элемент в кольце по модулю N можно найти благодаря расширенному алгоритму Евклида. Рассмотрим два натуральных числа: *a* и *b*. Каким может быть значение выражения *ax + by*, если взять два целых числа *x* и *y*? Можно ли подобрать такие целые чисел *x* и *y*, что значение выражения *ax + by* было бы равно *g* для некоторого целого числа *g*?

Пусть *d* - наибольший общий делитель чисел *a* и *b*. Тогда выражение *ax + by* всегда кратно *d*. Оказывается, что можно подобрать такие числа *x* и *y*, что *ax + by = d* [7].

Эту задачу решает расширенный алгоритм Евклида. Однако расширенный алгоритм Евклида находит такие ненужные параметры для нахождения обратного числа *a* в кольце по модулю *b*, как *d* и *y*. Поэтому существует полиномиальный расширенный алгоритм Евклида. Его реализация на псевдокоде:

|  |
| --- |
| function inverse(a, b)  x := 0;  newx := 1  y := b;  newy := a  **while** newy ≠ 0 **do**  quotient := y div newy  (x, newx) := (newx, x - quotient \* newx)  (r, newy) := (newy, y - quotient \* newy)  **if** y > 1 then  **return** "a is not invertible"  **if** x < 0 then  x := x + b  **return** t |

Для расшифровки шифротекста производятся те же операции, что и при шифровании:

Шифротекст делится на блоки каждый по *n* символов. Каждый блок зашифрованного сообщения рассматривается как *n*-мерный вектор чисел и умножается на найденную матрицу, обратную матрице-ключу, по модулю *N*. Полученный *n*-мерный вектор – блок расшифрованного сообщения размера *n* символов. Описанные операции проводятся для каждого блока исходного сообщения [8].

# Программная реализация алгоритмов

## Общая структура приложения

Реализованное в ходе выполнения данной курсовой работы приложение, полный листинг которого указан в приложении А, было спроектировано в соответствии с основными концепциями объектно-ориентированного программирования: инкапсуляцией, наследованием и полиморфизмом. Соответственно концепции наследования были реализованы два семейства классов:

* Семейство Palette, позволяющее удобнее взаимодействовать с обрабатываемым изображением;
* Семейство Crypto, наследники которого реализуют алгоритмы шифрования данной курсовой работы.

Семейство Palette (рис. 3) состоит из двух классов: родительский класс Palette и дочерний класс ImageInfo.

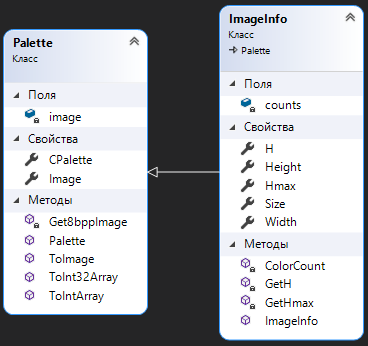


Рисунок 3 – Семейство классов Palette

Класс Palette предназначен для хранения обрабатываемого сообщения в восьмибитном формате, а также для конвертации его в массив целочисленных переменных и обратно. Содержит одно приватное поле image, в котором хранится изображение, и два свойства: CPalette – для хранения индексированной палитры изображения, и Image – для инкапсуляции поля image.

В этом классе так же есть один закрытый метод, конвертирующий 32-битное неиндексированное изображение в 8-битное индексированное с автоматическим составлением полной палитры изображения, состоящей из 256 цветов.

Оставшиеся три открытых метода в классе Palette: ToIntArray, ToInt32Array и ToImage отвечают за преобразование исходного изображения в двумерный массив целочисленных значений – для использования в шифровании алгоритмом Хилла, преобразование исходного изображения в одномерный массив целочисленных значений – для использования в шифровании гаммированием, и обратное преобразования массивов в 8-битное изображение.

Наследуемый от Palette класс ImageInfo, как становится ясно по названию, содержит всю необходимую информацию об этом изображении: размеры, текущая энтропия, максимальная возможная энтропия. Для обращения извне к этому классу доступны только свойства, хранящие эту информацию, и конструктор класса. Остальные поля и методы этого класса являются закрытыми.

Метод GetH высчитывает текущую энтропию изображения по формуле Шеннона (4), а метод GetHmax рассчитывает максимальную энтропию изображения по формуле Хартли (5). Метод ColorCount производит счет количество появлений определенного цвета палитры в изображении для метода GetH.

Семейство Crypto (рис. 4), в отличие от семейства Palette, имеет большую глубину наследования. Базовый класс этого семейства имеет только одно свойство для хранения экземпляра объекта шифруемого сообщения класса ImageInfo. От него наследуются классы Symmetric и Asymmetric – для хранения ключей шифрования симметричных и ассиметричных алгоритмов шифрования соответственно.

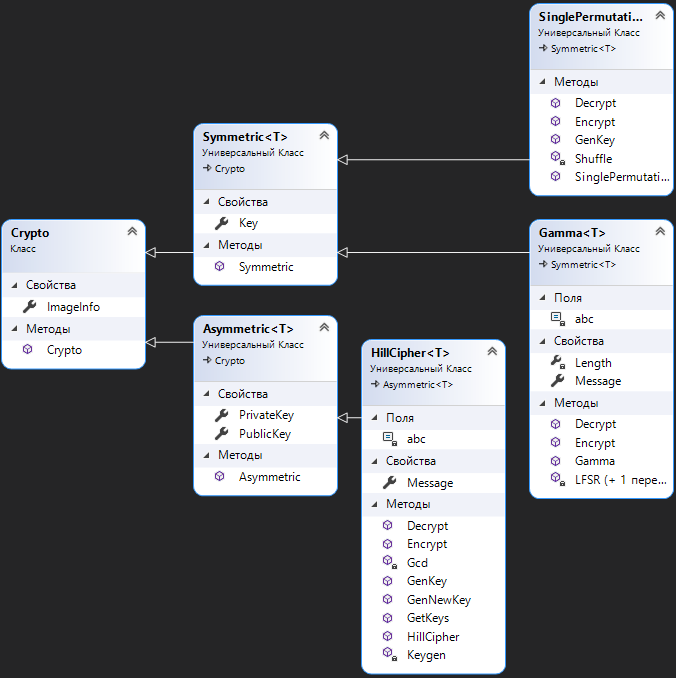


Рисунок 4 – Семейство классов Crypto

Исходя из названия, ясно, что в классе Symmetric есть только одно свойство для хранения ключа Key, в то время как в классе Asymmetric есть два свойства для хранения открытого и закрытого ключей – PublicKey и PrivateKey.

От класса Asymmetric наследуется класс HillCipher, реализующий шифрование и расшифрование изображение по алгоритму шифра Хилла. Этот класс содержит в себе приватное константное поле abc, хранящее в себе размер алфавита (количество цветов в палитре) и свойство Message, в котором хранится представление шифруемого изображения в формате матрицы для более удобного проведения операция шифрования и расшифровки в методах Encrypt и Decrypt соответственно.

Для получения ключей HillCipher содержит следующие методы:

* GenKey – Метод, создающий пару связанных ключей для шифрования и расшифрования;
* GenNewKey – Метод, генерирующий только ключ зашифрования;
* Keygen – Метод генерации псевдослучайной последовательности целочисленных значений, из которой метод GenKey создает ключи
* GetKeys – Метод, получающий заранее сгенерированную пару ключей из файла.

От класса Symmetric наследуются классы Gamma – для реализации алгоритма шифра гаммирования, и SinglePermutation – для реализации алгоритма блочной одинарной перестановки.

Приватное константное поле abc в классе Gamma выполняет ту же функцию, что и в классе HillCipher, а свойство Message, в отличие от HillCipher имеет тип данных одномерный целочисленный массив. Так же в класс добавлено новое свойство Length, регулирующее длину гаммы, по которой будет шифроваться в методе Encrypt и расшифровываться в методе Decrypt изображение. Генерация гаммы происходит в перегружаемом методе LFSR.

Класс Matrix (рис. 5), необходимый для матричных операций в шифре Хилла, содержит инкапсулируемые свойствами поля array, length, column, row. Каждое из них отвечает за хранение матрицы в виде двумерного целочисленного массива, его длины, количества столбцов и строк соответственно.

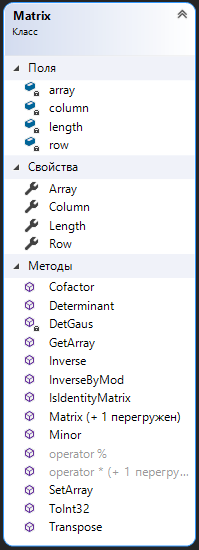


Рисунок 5 – Схема класса Matrix

Класс Matrix определяет такой функционал, как:

* Нахождение матрицы кофактора;
* Нахождение детерминанта матрицы;
* Нахождение обратной матрицы;
* Нахождение обратной матрицы по модулю;
* Нахождение миноров матрицы;
* Нахождение остатка от деления матрицы по модулю;
* Умножение матрицы на другую матрицу и на число;
* Транспонирование матрицы;
* Изменение содержимого матрицы блоками;
* Конвертация матрицы в одномерный целочисленный массив.

## Семейство классов Palette

### Класс Palette

При инициализации класса в конструкторе, как показано на рис. 6, программа проверяет формат изображения. Если формат изображения – 8-битное индексированное изображение, то программа сохраняет его и его палитру в свойства класса. В случае, когда принимаемое изображение имеет любой другой формат, программа конвертирует его в 8-битное индексированное изображение методом Get8bppImage (рис. 7), и только потом заполняет свойства.

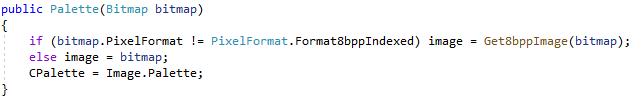


Рисунок 6 – Конструктор класса Palette

Конвертация изображения происходит за счет клонирования исходного изображения с форматом 8-битное индексированное изображение. Однако в таком случае из клонированного изображения исчезнет Alpha канал, и палитра будет заполнена не полностью, что может привести к ошибкам при шифровании. Для решения этой проблемы нужно вручную изменить палитру изображения путем занесения в нее своих ARGB цветов.

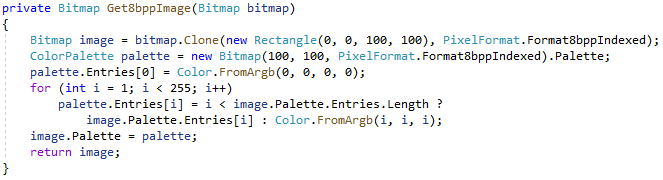


Рисунок 7 – Метод конвертации изображения в 8-битный формат

В шифровании часто неудобно работать с сообщением в его исходном отображении. Поэтому существуют кодировки, представляющие алфавит как множество натуральных чисел в разных системах счисления. Так же и в данном случае: производить математические операции с цветами невозможно. В языке программирования C# есть встроенная кодировка цветов, но в ней закодировано 262 цветов ∈ (-231… 231), что слишком много и не подходит для решения поставленных задач курсовой работы.

Для решения проблемы кодировки цветов и преобразования изображения в целочисленные массивы били реализованы методы ToIntArray и ToInt32Array (рис. 8). В этих методах создаются целочисленные массивы длинной сопоставимой с размерами изображения и заполняются в цикле номерами цветов пикселей изображения в палитре изображения. Обратная операция происходит в методе ToImage (рис. 9).

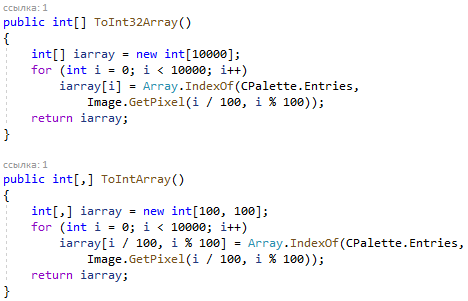


Рисунок 8 – Методы конвертации изображения в целочисленные массивы

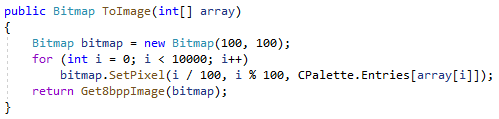


Рисунок 9 – Восстановление изображения из целочисленного массива

Такой метод кодировки позволяет избавиться от отрицательных чисел и позволяет перейти от 262 к 28 закодированных цветов ∈ (0 … 255). Таким образом время генерации ключей шифрования и расшифровки изображения будет существенно снижено.

### Класс ImageInfo

В конструкторе ImageInfo (рис. 10) сохраняются размеры изображения, после чего вызывается метод ColorCount (рис. 11), в котором рассчитывается количество появлений каждого из цветов в изображении. Как только было выявлено количество появлений для каждого цвета, производится расчёт энтропии изображения (рис 12).

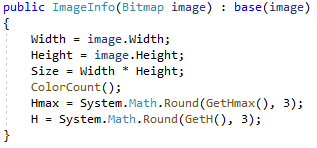


Рисунок 10 – Конструктор класса ImageInfo

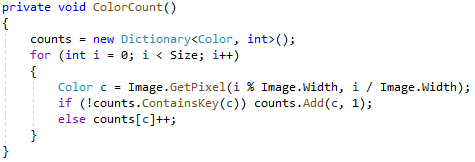


Рисунок 11 – Подсчет количества появлений цветов

ColorCount в цикле проходит по всему изображению, заполняю коллекцию Dictionary с ключом в виде цвета и счетчиком в значении. Т.е. если цвет данного пикселя не найден в коллекции, то в нее заносится новая пара цвет-счетчик с начальным значением 1, иначе счетчик увеличивается на единицу.

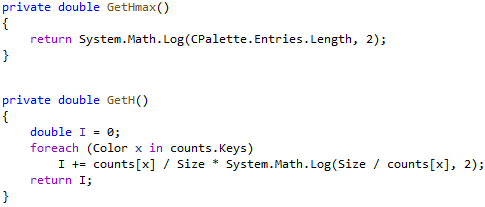


Рисунок 12 – Расчёт текущей и максимальной энтропии изображения

Подсчет количества появлений цветов, а не частоту их появления, позволяет избавиться от минуса в формуле Шеннона (4) переворотом дроби в логарифме.

## Семейство классов шифрования

### Класс Crypto

Класс Crypto (рис. 13), являясь родительским классом для семейства Crypto, содержит только свойство для хранения объекта класса ImageInfo для его шифрования или дешифрования. В конструкторе из Bitmap создается объект ImageInfo и сохраняется в свойстве.



Рисунок 13 – Класс Crypto

Использование объекта класса ImageInfo в качестве свойства класса Crypto позволяет при шифровании изображения автоматически получать всю информацию об измененном изображении. Таким образом, в Main программе, программист, создав объект одного из дочерних классов семейства Crypto и зашифровав с его помощью изображение, получает возможность сразу же получить энтропию зашифрованного изображения без создания лишних объектов.

### Класс Asymmetric

Класс Asymmetric (рис. 14) в свойствах хранит ключи шифрования и расшифрования. Особенность этого класса состоит в использовании обобщенных типов для хранения переменных в свойствах. Использование обобщенных типов позволяет хранить ключи любого типа данных без создания огромного количества классов под каждый из типов данных.

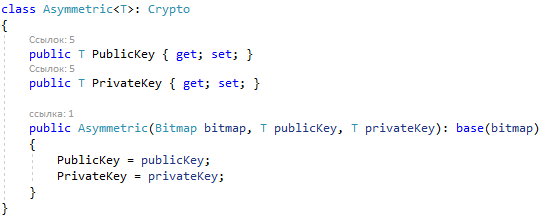


Рисунок 14 – Класс Asymmetric

В данном случае от класса Asymmetric наследуется только класс HillCipher, который пользуется ключами с типом данных Matrix. Поэтому использование обобщенных типов данных оправдано только возможностью будущего расширения функционала программы с добавлением других несимметричных криптоалгоритмов.

### Класс HillCipher

Класс HillCipher в конструкторе (рис. 15) конвертирует изображение в матрицу и сохраняет полученный результат в свойство Message.

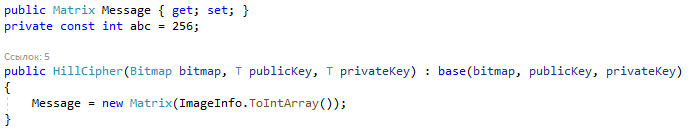


Рисунок 15 – Конструктор класса HillCipher

Методы Encrypt и Decrypt (рис. 16) идентичны и отличаются только используемым ключом. В обоих случаях сообщение разбивается на блоки той же длины, что и длина ключа. После каждый из блоков умножается по модулю длины алфавита с открытым ключом – в случае шифрования, и с закрытым ключом – в случае расшифрования.

В результате выполнения этих методов возвращается объект класса HillCipher с теми же самыми ключами, но с новым изображением в свойстве ImageInfo.

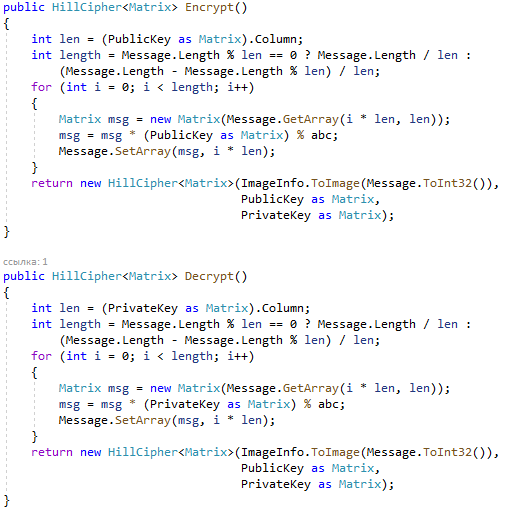


Рисунок 16 – Методы Encrypt и Decrypt класса HillCipher

Ключи для шифра Хилла можно получить двумя способами: прочитать и восстановить заранее сгенерированную пару ключей из файла (рис. 17), или сгенерировать пару ключей при инициализации объекта класса HillCipher.

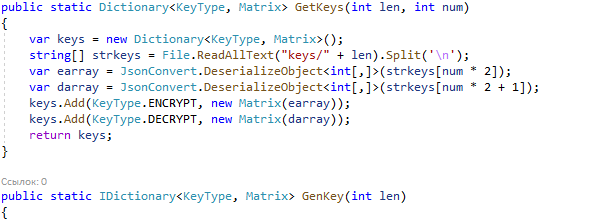


Рисунок 17 – Метод GetKeys класса HillCipher

Для данной курсовой работы были сгенерированы и записаны в файл 10 пар ключей для каждой длины ключа от 2 до 11. Поэтому в методе GetKeys, который восстанавливает ключи из файла, нужно указать требуюмую длину ключа и номер пары ключей в файле. Метод прочитает весь файл с указанной длиной ключа, разделит его содержимое на строки, и, используя библиотеку JsonConvert, восстановит матрицы пары ключей по номеру их расположения в файле. В результате исполнения метода будет возвращена коллекция Dicitionary с ключом в виде Enum перечисления с указанием на тип ключа шифрования (Encrypt или Decrypt) и значением в виде матрицы ключа шифрования.

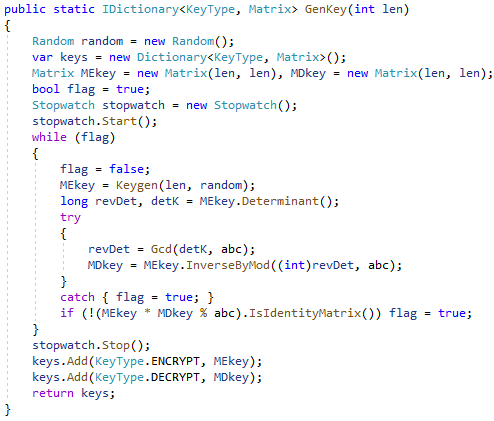


Рисунок 18 – Метод GenKey класса HillCipher

Генерация ключа начинается с составления псевдослучайной последовательности методом Keygen. Эта последовательность и есть открытый ключ для шифра Хилла.

Следующий этап – нахождение определителя матрицы открытого ключа. Определитель рассчитывается в классе Matrix в методе Determinant по методу Гаусса. Метод Гаусса, в отличие от рекурсивного метода, намного быстрее находит определитель матрицы, но делает это с некоторой погрешностью. Поэтому нужно отслеживать ошибки на следующих этапах поиска закрытого ключа.

Далее, нужно найти обратный определитель матрицы открытого числа в кольце по модулю длины алфавита. За это отвечает метод Gcd, который решает благодаря полиномиальному расширенному алгоритму Евклида. В половине случаев этот метод возвращает ошибку, т.к. в случае, когда определитель матрицы открытого ключа – четное число, определитель матрицы открытого ключа и длина алфавита, равная 256, не будут взаимно простыми, т.е. решения задачи Евклида существовать не будет.

Следующим шагом будет получен закрытый ключ путем нахождения матрицы, обратной матрице открытого ключа по модулю длины алфавита в методе InverseByMod класса Matrix. В этом методе матрица алгебраических дополнений открытого ключа делится по модулю длины ключа, умножается на обратный детерминанту элемент и снова делится по модулю длины ключа.

Последний этап – проверка, действительно ли открытый и закрытый ключи математически связаны. Если результат умножения пары ключей по модулю длины алфавита равен единичной матрице, то ключи сгенерированы корректно и метод вернет коллекцию Dicitionary из пары ключей. Иначе алгоритм генерации ключей начнется по новой с составления псевдослучайной последовательности методом Keygen.

### Класс Symmetric

Класс Symmetric (рис. 19) в свойствах хранит один ключ шифрования. Этот класс, так же, как и класс Asymmetric, использует обобщенный тип данных для хранения переменных в свойствах.

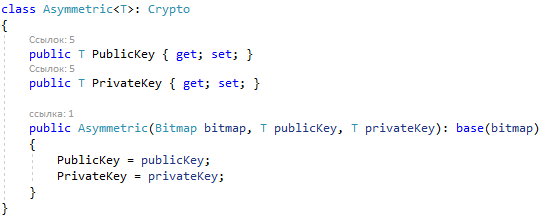


Рисунок 19 – Класс Symmetric

В данном случае от класса Symmetric наследуется классы SinglePermutation и Gamma, которые используют ключи с типами данных в виде целочисленного массива и целого числа соответственно. Поэтому использование обобщенных типов данных необходимо.

### Класс SinglePermutation

Класс SinglePermutation, реализующий алгоритм шифрования информации блочную одинарную перестановку, благодаря особенностям алгоритма не требует кодирования алфавита для зашифровывания и расшифровки информации. Поэтому в методах Encrypt и Decrypt (рис. 20) производятся преобразования напрямую с исходным сообщением.

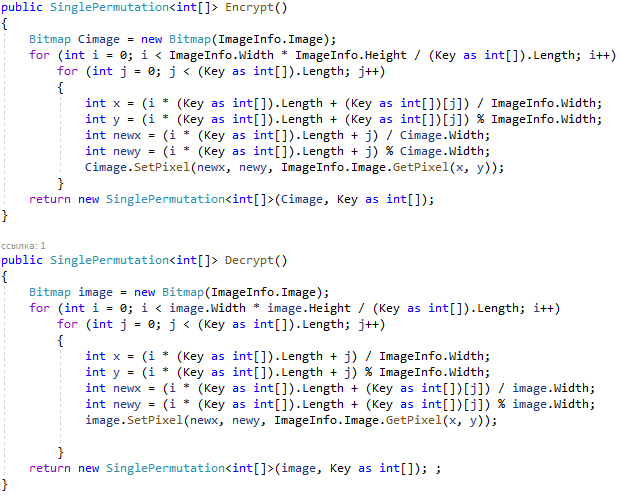


Рисунок 20 – Методы Encrypt и Decrypt класса SinglePermutation

Эти методы разбивают исходное изображение на блоки той же длины, что и длина ключа. После, в соответствии с ключом шифрования, метод Encrypt перемещают определенный пиксель из исходного положения в исходном изображении в другое положение зашифрованного изображения, таким образом перемешивая пиксели по блокам. Метод Decrypt восстанавливает исходное изображение из зашифрованного аналогичным образом.

Генерация ключа разделена на два метода: GenKey и Shuffle (рис. 21).

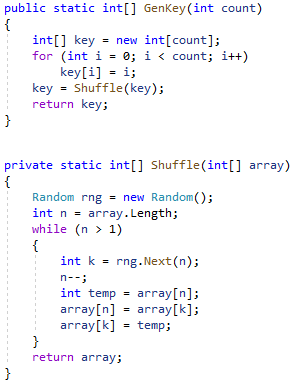


Рисунок 21 – Методы генерации ключа класса SinglePermutation

Для генерации ключа сначала формируется массив натуральных чисел с включенным нулем в счетном порядке. Далее полученный массив отправляется в метод Shuffle, реализующий алгоритм Тасование Фишера-Йетса. Тасование Фишера-Йетса гарантирует, что каждая перестановка генерируется с одинаковой вероятностью, к тому же современная версия алгоритма очень эффективна и требует время, пропорциональное числу элементов множества, и не требует дополнительной памяти. Поэтому этот алгоритм идеально подходит для генерации ключа для алгоритма блочной одинарной перестановки.

### Класс Gamma

Класс Gamma в конструкторе (рис. 22) конвертирует изображение в одномерный целочисленный массив и сохраняет полученный результат вместе с длиной ключа в свойства Message и Length соответственно. Message понадобится для гаммирования изображения, а Length – для генерации гаммы нужной длины.

Шифрование и расшифровка происходят в методах Encrypt и Decrypt (рис. 23).

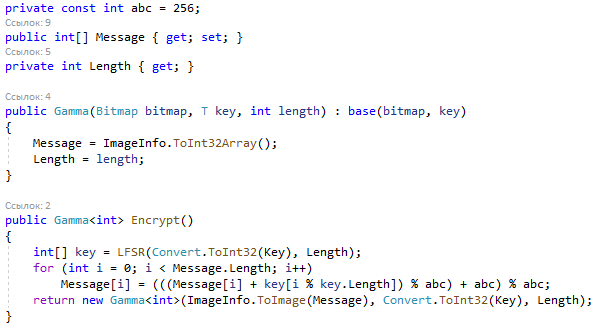


Рисунок 22 – Конструктор класса Gamma

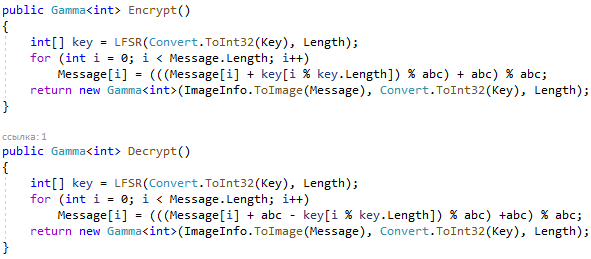


Рисунок 23 – Методы Encrypt и Decrypt класса Gamma

В обоих методах сначала генерируется гамма из первого ключевого значения заданной длины. Далее, в цикле по всему сообщению, происходит гаммирование сообщения по формулам (6) – для шифрования информации, (7) – для расшифровывания информации.

Однако, учитывая исследовательскую направленность данной курсовой работы, гамма может быть как равной длине сообщения, так и значительно меньше её. Поэтому, как только гамма кончается, программа начинает читать её заново.

Генерация гаммы (рис. 24) происходит методом регистра сдвига с линейной обратной связью по формуле *x8 + x4 + x3 + x2 + 1*. В основе вычленения бита с определенным номером из зерна служит операция конъюнкции с последующим побитовым сдвигом вправо. После вычленения нулевого, второго, третьего и четвертого бита из зерна, между полученными битами производится операция логического исключающего или. Для получения следующего значения гаммы зерно побитово сдвигается вправо на 1, а заместо исчезнувшего седьмого бита записывается результат предыдущей операции.

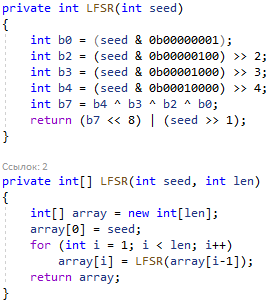


Рисунок 24 – Методы генерации гаммы

Для того, чтобы от гаммы бесконечной длины перейти к конечному множеству создается массив необходимой длины и заполняется рассчитанной гаммой, причем нулевой элемент массива равен зерну (ключу).

## Интерфейс приложения. Алгоритм работы приложения

Интерфейс приложения (рис. 25) состоит из нескольких блоков: блоки энтропии, блок изображений, блок изменения длины ключа, блок выбора алгоритма шифрования, блок с кнопками шифрования и расшифровки изображения.

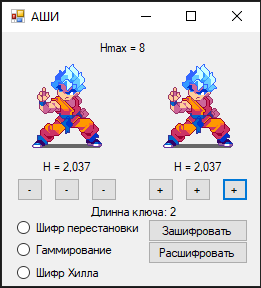


Рисунок 25 – Вид приложения при запуске

Сверху в центре на форме находится текстовое поле, в котором рассчитана максимальная возможная энтропия для данного изображения. Под ним две картинки: слева – исходное изображение, которое не будет меняться, справа – шифруемое и расшифровываемое изображение. Под ними соответственно энтропия оригинального изображения и энтропия шифруемого изображения.

Ниже находится ряд из шести кнопок, отвечающих за изменение длины ключа шифрования. Эти кнопки, слева направо, производят следующие действия с ключом:

* Уменьшение длины ключа на 100, ограничивая минимальную длину 2;
* Уменьшение длины ключа на 10, ограничивая минимальную длину 2;
* Уменьшение длины ключа на 1, ограничивая минимальную длину 2;
* Увеличение длины ключа на 1, ограничивая максимальную длину 10000;
* Увеличение длины ключа на 10, ограничивая максимальную длину 10000;
* Увеличение длины ключа на 100, ограничивая максимальную длину 10000.

Под кнопками регулирования длины ключа есть текстовое поле, показывающее текущую длину ключа. Стартовое значение длины ключа – 2.

В самом низу пользовательской формы слева находятся переключатель для выбора алгоритма шифрования, которым будет обрабатываться изображения, а справа – кнопки для зашифрования или расшифровки изображения.

При нажатии на кнопку зашифрования изображения выполняется код, указанный на рис. 26. Действия, выполняемые этим кодом, зависят от выбранного пользователем переключателя.



Рисунок 26 – Шифрование изображения по одному из алгоритмов

Если выбран переключатель, ответственный за шифрование изображения шифром Хилла, в глобальную переменную сохраняется этот переключатель для сохранения очередности расшифрования. Далее выбирается одна из десяти пар заранее сгенерированных ключей. Если размер матрицы ключа меньше 11x11, инициализируется объект класса HillCipher с ключами по выбранному номеру соответствующей длины. В ином случае, инициализируется объект класса HillCipher только с одним ключом шифрования соответствующей длины, а кнопка расшифрования на пользовательской форме блокируется. Такой алгоритм действий связан со сложностью генерации ключей для шифра Хилла В результате изображение будет зашифровано, и текущая энтропия изображения будет пересчитана. Результат будет выведен на пользовательскую форму.

Если выбран переключатель, ответственный за шифрование изображения блочной одинарной перестановкой, в глобальную переменную сохраняется этот переключатель для сохранения очередности расшифрования и инициализируется новый объект класса SinglePermutation с ключом, сгенерированным в статическом методе GenKey. В результате изображение будет зашифровано, и текущая энтропия изображения будет пересчитана. Результат будет выведен на пользовательскую форму.

Если выбран переключатель, ответственный за шифрование методом гаммирования, в глобальную переменную сохраняется этот переключатель для сохранения очередности расшифрования и инициализируется новый объект класса Gamma с ключом в виде сгенерированного псевдослучайного числа в диапазоне от 0 до 256. В результате изображение будет зашифровано, и текущая энтропия изображения будет пересчитана. Результат будет выведен на пользовательскую форму.

Если ни один из переключателей не будет выбран пользователю, будет воспроизведен предупреждающий звук.

При нажатии на кнопку зашифрования изображения выполняется код, указанный на рис. 27. Действия, выполняемые этим кодом, зависят от выбранного пользователем переключателя при шифровании изображения.

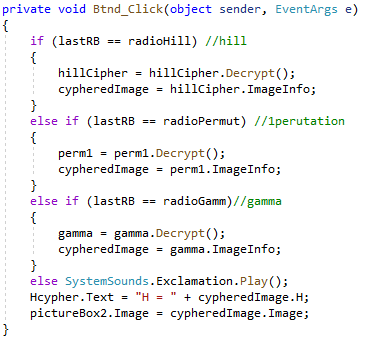


Рисунок 27 – Расшифровка изображения по одному из алгоритмов

В зависимости от того, какой переключатель был выбран последним при шифровании изображения, будет выбран метод Decrypt одного из классов: HillCipher, SinglePermutation или Gamma. В результате изображение будет расшифровано, и текущая энтропия изображения будет пересчитана. Результат будет выведен на пользовательскую форму.

# Сравнение качества шифрования алгоритмов

Индикатором качества шифрования используется энтропия *H*, рассчитанная по формуле Шеннона. Максимальное значение энтропии определяется по формуле Хартли *H*max. Чем ближе *Н* к *Н*max, тем качественнее зашифрована информация.

Для проверки качества шифрования было измерено изменение энтропии в зависимости от длины ключа. Для измерения изменения энтропии было сгенерировано по десять ключей каждой длины в диапазоне от 1 до 10000 (если такой ключ существует). По результатам измерения были построены графики с логарифмическим масштабом по оси абсцисс (ось длины ключей), по которым можно судить о качестве шифрования информации.

Для алгоритма шифрования блочной одинарной перестановки был построен график (рис. 28).

Рисунок 28 – Зависимость энтропии от длины ключа для шифра перестановки

Как видно из этого графика, для шифра блочной одинарной перестановки нет зависимости между энтропией зашифрованного изображения и длины ключа шифрования. Вне зависимости от длины ключа энтропия зашифрованного изображения остается равной энтропии исходного изображения, следовательно качество шифрования алгоритма блочной одинарной перестановки минимальна. Для шифрования важной информации рекомендуется использовать шифр блочной одинарной перестановки только в составе комбинированных алгоритмов шифрования.

Результат исследования качества шифрования для шифра гаммирования с алгоритмом генерации гаммы «регистр сдвига с линейной обратной связью» указан на рис. 29.

Рисунок 29 – Зависимость энтропии от длины ключа для гаммирования

Из этого графика видно, что энтропия с сильными колебаниями растет по логарифмическому закону до длины гаммы около 120 – 130 единиц, после чего, так же с колебаниями, придерживается энтропии в 7 бит. Максимальное значение энтропии на этом графике – 7,114 бит.

Качество шифрования гаммированием очень высоко. Для получения наилучшего результата шифрования информации рекомендуется использовать гамму, размером не меньше 120 единиц. Однако получение энтропии, близкой к максимальной, не гарантировано из-за недостатков генерации гаммы методом регистра сдвига с линейной обратной связью (заполнение РСЛОС нулями влечет вывод регистром бесконечной последовательности нулей, что совершенно бесполезно).

Из-за сложности генерации пар ключей для шифра Хилла так же был составлен график зависимости среднего времени генерации десяти пар ключей от длины ключа (рис. 30).

Рисунок 30 – Зависимость времени генерации ключа от длины ключа

Из этого графика видно, что время генерации пары ключей незначительно при размере ключа от 2x2 до 9x9 и составляет менее 100 мс. При размере матрицы ключа 10x10 среднее время генерации ключа уже составляет 37,63 с. Пара матриц размером 11x11 в среднем составляет уже 9 м. 37,16 с. Таким образом генерация ключей большего размера не представляется возможным.

Для сгенерированных ключей шифра Хилла (до размера 11x11) был построен следующий график зависимости (рис. 31).

Рисунок 31 – Зависимость энтропии от длины ключа до размера 11x11 для шифра Хилла

По графику видно, что колебания энтропии в шифре Хилла минимальны, и придерживаются вида логарифмической функции. Максимальной энтропии в 4,289 бит график достигает при размере матрицы 10x10. Таким образом, качество шифрования шифром Хилла среднее, а сложность генерации пары ключей не позволяет шифровать информацию более качественно.

В исследовательских целях был построен график зависимости энтропии от длины ключа до размера ключа 100x100 (рис. 32). Для этого генерировался только ключ шифрования, без ключа расшифровки.

Рисунок 32 – Зависимость энтропии от длины ключа для шифра Хилла

Из этого графика видно, что зависимость энтропии от длины ключа при любой длине ключа имеет форму графика логарифмической функции. На этой функции видны два выделяющихся пика при размерах ключей 10x10 и 100x100, что равно 1/10 размера шифруемого изображения и размеру шифруемого изображения соответственно. Это значит, что шифр Хилла чувствителен к соотношению размеров матрицы ключа и шифруемого сообщения. Максимальная возможная энтропия зашифрованного шифром Хилла изображения составляет 5,993 бит.

# Заключение

В ходе выполнения данной курсовой работы было спроектировано и разработано приложение для выполнения шифрования восьмибитного графического изображения в формате BMP шифрами блочной одинарной перестановки, Хилла и гаммирования на основе регистра сдвига с линейной обратной связью.

Для исследовательской работы общим изменяемым параметром выбранных алгоритмов шифрования для получения наилучшего результата на основе графиков зависимости энтропии *Н* от значений параметров алгоритмов шифрования.

Для измерения изменения энтропии было сгенерировано по десять ключей каждой длины в диапазоне от 1 до 10000. По результатам измерения были построены графики с логарифмическим масштабом по оси абсцисс, по которым можно судить о качестве шифрования информации.

Качество шифрования алгоритма блочной одинарной перестановки минимальна, потому что энтропия зашифрованного изображения вне зависимости от длинны ключа остается неизменной. Для шифрования важной информации шифр блочной одинарной перестановки не подходит.

Второй по качеству шифрования в данной работе – шифр Хилла. Этот алгоритм позволяет увеличить энтропию зашифрованного сообщения относительно зашифрованного в 2 раза. Этот алгоритм является блочным и поэтому дешифровка зашифрованного сообщения – сложная задача. Однако сложность генерации пары ключей не позволяет шифровать информацию более качественно.

Качество шифрования в алгоритме гаммирования лучшее среди всех шифров, рассмотренных в данной курсовой работе. Энтропия зашифрованного сообщения может достигать 88% от максимально возможной энтропии сообщения. Однако шифрование сообщения с оптимальными параметрами шифрования не гарантирует высокой энтропии зашифрованного сообщения.

# Список использованных источников

1. А. Сериков. Администратор информационной безопасности [Электронный ресурс]: Основы криптографии / А. Сериков.; Национальный Открытый Университет «ИНТУИТ». – электрон. текст. дан. – Москва: НОУ «ИНТУИТ», 19.09.2011. – Режим доступа: https://intuit.ru/studies/mini\_mba/5398/courses/547/lecture/12395?page=2#
2. Бекман И. Н. ИНФОРМАТИКА Курс лекций [Электронный ресурс]: Лекция 8. ЭНТРОПИЯ и ИНФОРМАЦИЯ / Бекман И. Н.. – электрон. текст. дан. – Москва: Каталог сайтов ManyWeb.ru, 2009. – Режим доступа: http://profbeckman.narod.ru/InformLekc.files/Inf08.pdf
3. Эндрю Брандт, Александра Красне. Как работает шифрование / Эндрю Брандт, Александра Красне// Мир ПК. – 2000. – №04.
4. Анисимов В. В. Криптографические методы защиты информации [Электронный ресурс]: Шифры перестановки /Анисимов В. В. – электрон. текст. дан. – Москва: Сайты Google, 2021. – Режим доступа: https://sites.google.com/site/anisimovkhv/learning/kripto/lecture/tema5
5. Шифр перестановки [Электронный ресурс]: Алгоритм, реализация на C# / vscode.ru. – электрон. текст. дан. – Москва: vscode.ru, 22.09.2017. – Режим доступа: https://vscode.ru/prog-lessons/shifr-perestanovki.html
6. Анисимов В. В. Криптографические методы защиты информации [Электронный ресурс]: Шифры гаммирования /Анисимов В. В. – электрон. текст. дан. – Москва: Сайты Google, 2021. – Режим доступа: https://sites.google.com/site/anisimovkhv/learning/kripto/lecture/tema6
7. Арифметические алгоритмы [Электронный ресурс]: Расширенный алгоритм Евклида / Фоксфорд. – электрон. текст. дан. – Москва: Фоксфорд. – Режим доступа: https://foxford.ru/wiki/informatika/rasshirennyy-algoritm-evklida
8. Шифр Хила [Электронный ресурс]: Подробный разбор / Хабр. – электрон. текст. дан. – Москва: Хабр. – Режим доступа: https://habr.com/ru/post/332714/

# Приложение А

(обязательное)

**namespace** BMPscrambler

{

**public** **partial** **class** Form1 : Form

{

**readonly** Random random;

**readonly** ImageInfo originalImage;

**readonly** Bitmap bmp = Properties.Resources.\_21;

ImageInfo cypheredImage;

HillCipher<Matrix> hillCipher;

SinglePermutation<**int**[]> perm1;

Gamma<**int**> gamma;

RadioButton lastRB = new RadioButton();

**int** k = 2;

**public** Form1()

{

InitializeComponent();

originalImage = new ImageInfo(bmp);

cypheredImage = new ImageInfo(bmp);

random = new Random();

}

**private** **void** Form1\_Load(**object** sender, EventArgs e)

{

pictureBox1.Image = originalImage.Image;

Horig.Text = "H = " + originalImage.H;

pictureBox2.Image = originalImage.Image;

Hcypher.Text = "H = " + originalImage.H;

Hmax.Text = "Hmax = " + originalImage.Hmax;

}

**private** **void** Btne\_Click(**object** sender, EventArgs e)

{

**if** (radioHill.Checked) *//hill*

{

lastRB = radioHill;

**int** rnd = random.Next(0, 10);

hillCipher = new HillCipher<Matrix>(originalImage.Image,

HillCipher<Matrix>.GetKeys(k, rnd)[KeyType.ENCRYPT],

HillCipher<Matrix>.GetKeys(k, rnd)[KeyType.DECRYPT]).Encrypt();

btnd.Enabled = k < 11 ? **true** : **false**;

cypheredImage = hillCipher.ImageInfo;

}

**else** **if** (radioPermut.Checked) *//1perutation*

{

lastRB = radioPermut;

perm1 = new SinglePermutation<**int**[]>(originalImage.Image,

SinglePermutation<**int**>.GenKey(k)).Encrypt();

cypheredImage = perm1.ImageInfo;

}

**else** **if** (radioGamm.Checked) *//gamma*

{

lastRB = radioGamm;

gamma = new Gamma<**int**>(originalImage.Image, random.Next(0, 256), k).Encrypt();

cypheredImage = gamma.ImageInfo;

}

**else** SystemSounds.Exclamation.Play();

Hcypher.Text = "H = " + cypheredImage.H;

pictureBox2.Image = cypheredImage.Image;

}

**private** **void** Btnd\_Click(**object** sender, EventArgs e)

{

**if** (lastRB == radioHill) *//hill*

{

HillCipher<Matrix> res = hillCipher.Decrypt();

cypheredImage = res.ImageInfo;

}

**else** **if** (lastRB == radioPermut) *//1perutation*

{

perm1 = perm1.Decrypt();

cypheredImage = perm1.ImageInfo;

}

**else** **if** (lastRB == radioGamm)*//gamma*

{

gamma = gamma.Decrypt();

cypheredImage = gamma.ImageInfo;

}

**else** SystemSounds.Exclamation.Play();

Hcypher.Text = "H = " + cypheredImage.H;

pictureBox2.Image = cypheredImage.Image;

}

**private** **void** Btnmin1\_Click(**object** sender, EventArgs e)

{

k = k > 2 ? k -= 1 : k;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** Btnmin10\_Click(**object** sender, EventArgs e)

{

k = k > 12 ? k -= 10 : k;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** Btnmin100\_Click(**object** sender, EventArgs e)

{

k = k > 102 ? k -= 100 : k;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** Btnplus1\_Click(**object** sender, EventArgs e)

{

k += 1;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** Btnplus10\_Click(**object** sender, EventArgs e)

{

k += 10;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** Btnplus100\_Click(**object** sender, EventArgs e)

{

k += 100;

keyLen.Text = "Длинна ключа: " + k;

}

**private** **void** BtnAuto\_Click(**object** sender, EventArgs e)

{

Stopwatch stopwatch;

**string** fileheader = new StringBuilder().AppendLine(

"Key length;Hmax;H;Encryption time").ToString();

File.WriteAllText("hill.csv", fileheader);

File.WriteAllText("gamma.csv", fileheader);

File.WriteAllText("1permutation.csv", fileheader);

**for** (**int** k = 2; k <= 11; k++)

{

StringBuilder hcs = new StringBuilder();

StringBuilder gcs = new StringBuilder();

StringBuilder pcs = new StringBuilder();

**for** (**int** j = 0; j < 10; j++)

{

Gamma<**int**> gc = new Gamma<**int**>(bmp, random.Next(0, 256), k);

SinglePermutation<**int**[]> pc = new SinglePermutation<**int**[]>(bmp,

SinglePermutation<**int**>.GenKey(k));

**if** (k > 100 & k % 100 == 0)

{

HillCipher<Matrix> hc = new HillCipher<Matrix>(

bmp,

HillCipher<Matrix>.GenNewKey(k/100)[KeyType.ENCRYPT],

HillCipher<Matrix>.GenNewKey(k/100)[KeyType.ENCRYPT]);

stopwatch = new Stopwatch();

stopwatch.Start();

hc = hc.Encrypt();

stopwatch.Stop();

hcs.AppendLine(**String**.Format("{0};{1};{2};{3}",

k,

hc.ImageInfo.Hmax,

hc.ImageInfo.H,

stopwatch.ElapsedMilliseconds));

}

stopwatch = new Stopwatch();

stopwatch.Start();

gc = gc.Encrypt();

stopwatch.Stop();

gcs.AppendLine(**String**.Format("{0};{1};{2};{3}",

k,

gc.ImageInfo.Hmax,

gc.ImageInfo.H,

stopwatch.ElapsedMilliseconds));

stopwatch = new Stopwatch();

stopwatch.Start();

pc = pc.Encrypt();

stopwatch.Stop();

pcs.AppendLine(**String**.Format("{0};{1};{2};{3}",

k,

pc.ImageInfo.Hmax,

pc.ImageInfo.H,

stopwatch.ElapsedMilliseconds));

}

File.AppendAllText("hill.csv", hcs.ToString());

File.AppendAllText("gamma.csv", gcs.ToString());

File.AppendAllText("1permutation.csv", pcs.ToString());

}

MessageBox.Show("Done");

}

}

}

**namespace** BMPscrambler.Classes

{

**class** Palette

{

**private** Bitmap image;

**public** Bitmap Image { **get** { **return** image; } **set** { image = Image; } }

**public** ColorPalette CPalette { **get**; }

**public** Palette(Bitmap bitmap)

{

**if** (bitmap.PixelFormat != PixelFormat.Format8bppIndexed) image = Get8bppImage(bitmap);

**else** image = bitmap;

CPalette = Image.Palette;

}

**public** **int**[] ToInt32Array()

{

**int**[] iarray = new **int**[10000];

**for** (**int** i = 0; i < 10000; i++)

iarray[i] = Array.IndexOf(CPalette.Entries, Image.GetPixel(i / 100, i % 100));

**return** iarray;

}

**public** **int**[,] ToIntArray()

{

**int**[,] iarray = new **int**[100, 100];

**for** (**int** i = 0; i < 10000; i++)

iarray[i / 100, i % 100] = Array.IndexOf(CPalette.Entries,

Image.GetPixel(i / 100, i % 100));

**return** iarray;

}

**public** Bitmap ToImage(**int**[] array)

{

Bitmap bitmap = new Bitmap(100, 100);

**for** (**int** i = 0; i < 10000; i++)

bitmap.SetPixel(i / 100, i % 100, CPalette.Entries[array[i]]);

**return** Get8bppImage(bitmap);

}

**private** Bitmap Get8bppImage(Bitmap bitmap)

{

Bitmap image = bitmap.Clone(new Rectangle(0, 0, 100, 100), PixelFormat.Format8bppIndexed);

ColorPalette palette = new Bitmap(100, 100, PixelFormat.Format8bppIndexed).Palette;

palette.Entries[0] = Color.FromArgb(0, 0, 0, 0);

**for** (**int** i = 1; i < 255; i++)

palette.Entries[i] = i < image.Palette.Entries.Length ?

image.Palette.Entries[i] : Color.FromArgb(i, i, i);

image.Palette = palette;

**return** image;

}

}

**class** ImageInfo: Palette

{

**public** **double** H { **get**; }

**public** **double** Hmax { **get**; }

**public** **double** Size { **get**; }

**public** **int** Width { **get**; }

**public** **int** Height { **get**; }

**private** IDictionary<Color, **int**> counts;

**public** ImageInfo(Bitmap image) : **base**(image)

{

Size = image.Width \* image.Height;

ColorCount();

Hmax = System.Math.Round(GetHmax(), 3);

H = System.Math.Round(GetH(), 3);

Width = image.Width;

Height = image.Height;

}

**private** **double** GetHmax()

{

**return** System.Math.Log(CPalette.Entries.Length, 2);

}

**private** **double** GetH()

{

**double** I = 0;

**foreach** (Color x **in** counts.Keys)

I += counts[x] / Size \* System.Math.Log(Size / counts[x], 2);

**return** I;

}

**private** **void** ColorCount()

{

counts = new Dictionary<Color, **int**>();

**for** (**int** i = 0; i < Size; i++)

{

Color c = Image.GetPixel(i % Image.Width, i / Image.Width);

**if** (!counts.ContainsKey(c)) counts.**Add**(c, 1);

**else** counts[c]++;

}

}

}

**class** Matrix

{

**private** **int**[,] array;

**readonly** **int** row, column, length;

**public** **int** Row { **get** { **return** row; } }

**public** **int** Column { **get** { **return** column; } }

**public** **int** Length { **get** { **return** length; } }

**public** **int**[,] Array { **get** { **return** array; } **set** { array = Array; } }

**public** Matrix(**int** row, **int** column)

{

**this**.row = row;

**this**.column = column;

length = row \* column;

array = new **int**[row, column];

}

**public** Matrix(**int**[,] matrix)

{

row = matrix.GetLength(0);

column = matrix.GetLength(1);

length = matrix.Length;

array = matrix;

}

**public** **bool** IsIdentityMatrix()

{

**if** (array == **null** || column != row) **return** **false**;

**for** (**int** i = 0; i < row; i++)

**for** (**int** j = 0; j < column; j++)

**if** ((i != j && array[i, j] != 0) || (i == j && array[i, j] != 1))

**return** **false**;

**return** **true**;

}

**public** **int**[] ToInt32()

{

**int**[] ret = new **int**[Column \* Row];

**for** (**int** i = 0; i < Column; i++)

**for** (**int** j = 0; j < Row; j++)

ret[i \* Column + j] = Array[i, j];

**return** ret;

}

**public** **int**[,] GetArray(**int** start, **int** len)

{

**int**[,] arr = new **int**[1, len];

**for** (**int** i = 0; i < len; i++)

arr[0, i] = Array[((start + i) / Column) % Column, (start + i) % Column];

**return** arr;

}

**public** **void** SetArray(Matrix matrix, **int** start)

{

**for** (**int** i = 0; i < matrix.Column; i++)

array[start / Column, (start + i) % Column] = matrix.Array[0, i];

}

**public** Matrix Transpose()

{

Matrix m = new Matrix(column, row);

**for** (**int** i = 0; i < row; i++)

**for** (**int** j = 0; j < column; j++)

m.array[j, i] = array[i, j];

**return** m;

}

**public** Matrix Inverse()

{

**int** det = Determinant();

**if** (det == 0) **throw** new Exception("Матрица вырождена");

Matrix m = new Matrix(row, column);

**for** (**int** i = 0; i < row; i++)

**for** (**int** j = 0; j < column; j++)

m.array[i, j] = Cofactor(array, i, j);

**return** m;

}

**public** Matrix InverseByMod(**int** revDet, **int** abc)

{

**int** det = Determinant();

**if** (det == 0) **throw** new Exception("Матрица вырождена");

Matrix m = **this**.Inverse();

m %= abc;

m \*= revDet;

m %= abc;

**for** (**int** i = 0; i < row; i++)

**for** (**int** j = 0; j < column; j++)

m.array[i, j] = m.array[i, j] < 0 ? abc + m.array[i, j] : m.array[i, j];

**return** m.Transpose();

}

**public** **int** Determinant()

{

**if** (column != row) **throw** new Exception("Расчет определителя невозможен");

**return** DetGaus(array);

}

**private** **int** DetGaus(**int**[,] c)

{

**double** det = 1;

**const** **double** EPS = 1E-9;

**int** n = c.GetLength(0);

**int**[,] a = (**int**[,])c.Clone();

**for** (**int** i = 0; i < n; ++i)

{

**int** k = i;

**for** (**int** j = i + 1; j < n; ++j)

**if** (Math.Abs(a[j, i]) > Math.Abs(a[k, i]))

k = j;

**if** (Math.Abs(a[k, i]) < EPS)

{

det = 0;

**break**;

}

**for** (**int** x = 0; x < n; x++)

{

**int** buf = a[i, x];

a[i, x] = a[k, x];

a[k, x] = buf;

}

**if** (i != k) det = -det;

det \*= a[i, i];

**for** (**int** j = i + 1; j < n; ++j)

a[i, j] /= a[i, i];

**for** (**int** j = 0; j < n; ++j)

**if** ((j != i) && (Math.Abs(a[j, i]) > EPS))

**for** (k = i + 1; k < n; ++k)

a[j, k] -= a[i, k] \* a[j, i];

}

**return** (**int**)Math.Round(det);

}

**public** **int** Cofactor(**int**[,] array, **int** row, **int** column)

{

**return** Convert.ToInt32(Math.Pow(-1, column + row)) \* DetGaus(Minor(array, row, column));

}

**public** **int**[,] Minor(**int**[,] array, **int** row, **int** column)

{

**int** n = (**int**)Math.Sqrt(array.Length);

**int**[,] minor = new **int**[n - 1, n - 1];

**int** \_i = 0;

**for** (**int** i = 0; i < n; i++)

{

**if** (i == row) **continue**;

**int** \_j = 0;

**for** (**int** j = 0; j < n; j++)

{

**if** (j == column) **continue**;

minor[\_i, \_j] = array[i, j];

\_j++;

}

\_i++;

}

**return** minor;

}

**public** **static** Matrix **operator** \*(Matrix m1, Matrix m2)

{

**if** (m1.column != m2.row) **throw** new Exception("Умножение невозможно");

Matrix m = new Matrix(m1.row, m2.column);

**for** (**int** i = 0; i < m1.row; i++)

{

**for** (**int** j = 0; j < m2.column; j++)

{

m.array[i, j] = 0;

**for** (**int** k = 0; k < m1.column; k++)

m.array[i, j] += m1.array[i, k] \* m2.array[k, j];

}

}

**return** m;

}

**public** **static** Matrix **operator** %(Matrix m1, **int** mod)

{

Matrix m = new Matrix(m1.row, m1.column);

**for** (**int** i = 0; i < m1.row; i++)

**for** (**int** j = 0; j < m1.column; j++)

m.array[i, j] = (**int**)m1.array[i, j] % mod;

**return** m;

}

**public** **static** Matrix **operator** \*(Matrix m1, **int** a)

{

Matrix m = new Matrix(m1.row, m1.column);

**for** (**int** i = 0; i < m1.row; i++)

**for** (**int** j = 0; j < m1.column; j++)

m.array[i, j] = (**int**)m1.array[i, j] \* a;

**return** m;

}

}

**public** **enum** KeyType

{

ENCRYPT,

DECRYPT

}

**class** Crypto

{

**public** ImageInfo ImageInfo { **get**; }

**public** Crypto(Bitmap bitmap)

{

ImageInfo = new ImageInfo(bitmap);

}

} **class** Asymmetric<T>: Crypto

{

**public** T PublicKey { **get**; **set**; }

**public** T PrivateKey { **get**; **set**; }

**public** Asymmetric(Bitmap bitmap, T publicKey, T privateKey): **base**(bitmap)

{

PublicKey = publicKey;

PrivateKey = privateKey;

}

}

**class** HillCipher<T>: Asymmetric<T>

{

**public** Matrix Message { **get**; **set**; }

**private** **const** **int** abc = 256;

**public** HillCipher(Bitmap bitmap, T publicKey, T privateKey) : **base**(bitmap, publicKey, privateKey)

{

Message = new Matrix(ImageInfo.ToIntArray());

}

**public** HillCipher<Matrix> Encrypt()

{

**int** len = (PublicKey **as** Matrix).Column;

**int** length = Message.Length % len == 0 ? Message.Length / len :

(Message.Length - Message.Length % len) / len;

**for** (**int** i = 0; i < length; i++)

{

Matrix msg = new Matrix(Message.GetArray(i \* len, len));

msg = msg \* (PublicKey **as** Matrix) % abc;

Message.SetArray(msg, i \* len);

}

**return** new HillCipher<Matrix>(ImageInfo.ToImage(Message.ToInt32()),

PublicKey **as** Matrix,

PrivateKey **as** Matrix);

}

**public** HillCipher<Matrix> Decrypt()

{

**int** len = (PrivateKey **as** Matrix).Column;

**int** length = Message.Length % len == 0 ? Message.Length / len :

(Message.Length - Message.Length % len) / len;

**for** (**int** i = 0; i < length; i++)

{

Matrix msg = new Matrix(Message.GetArray(i \* len, len));

msg = msg \* (PrivateKey **as** Matrix) % abc;

Message.SetArray(msg, i \* len);

}

**return** new HillCipher<Matrix>(ImageInfo.ToImage(Message.ToInt32()),

PublicKey **as** Matrix,

PrivateKey **as** Matrix);

}

**public** **static** Dictionary<KeyType, Matrix> GetKeys(**int** len, **int** num)

{

**var** keys = new Dictionary<KeyType, Matrix>();

**string**[] strkeys = File.ReadAllText("keys/" + len).Split('**\n**');

**var** earray = JsonConvert.DeserializeObject<**int**[,]>(strkeys[num \* 2]);

**var** darray = JsonConvert.DeserializeObject<**int**[,]>(strkeys[num \* 2 + 1]);

keys.**Add**(KeyType.ENCRYPT, new Matrix(earray));

keys.**Add**(KeyType.DECRYPT, new Matrix(darray));

**return** keys;

}

**public** **static** IDictionary<KeyType, Matrix> GenKey(**int** len)

{

Random random = new Random();

**var** keys = new Dictionary<KeyType, Matrix>();

Matrix MEkey = new Matrix(len, len), MDkey = new Matrix(len, len);

**bool** flag = **true**;

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

**while** (flag)

{

flag = **false**;

MEkey = Keygen(len, random);

**long** revDet, detK = MEkey.Determinant();

**try**

{

revDet = Gcd(detK, abc);

MDkey = MEkey.InverseByMod((**int**)revDet, abc);

}

**catch** { flag = **true**; }

**if** (!(MEkey \* MDkey % abc).IsIdentityMatrix()) flag = **true**;

}

stopwatch.Stop();

keys.**Add**(KeyType.ENCRYPT, MEkey);

keys.**Add**(KeyType.DECRYPT, MDkey);

**return** keys;

}

**public** **static** IDictionary<KeyType, Matrix> GenNewKey(**int** len)

{

Random random = new Random();

**var** keys = new Dictionary<KeyType, Matrix>();

Matrix MEkey = Keygen(len, random), MDkey = Keygen(len, random);

keys.**Add**(KeyType.ENCRYPT, MEkey);

keys.**Add**(KeyType.DECRYPT, MDkey);

**return** keys;

}

**private** **static** Matrix Keygen(**int** len, Random random)

{

**int**[,] key = new **int**[len, len];

**for** (**int** i = 0; i < Math.Pow(len, 2); i++)

key[i / len, i % len] = random.Next(0, 255);

**return** new Matrix(key);

}

**private** **static** **long** Gcd(**long** det, **long** mod)

{

**long** t = 0;

**long** r = mod;

**long** newt = 1;

**long** newr = det;

**while** (newr != 0)

{

**long** qoutient = r / newr;

**long** nnt = t - qoutient \* newt;

t = newt;

newt = nnt;

**long** nnr = r - qoutient \* newr;

r = newr;

newr = nnr;

}

**if** (r > 1) **throw** new Exception("a is not invertible");

**if** (t < 0) t += mod;

**return** t;

}

}

**class** Symmetric <T>: Crypto

{

**public** T Key { **get**; }

**public** Symmetric(Bitmap bitmap, T key): **base**(bitmap)

{

Key = key;

}

}

**class** SinglePermutation<T> : Symmetric<T>

{

**public** SinglePermutation(Bitmap bitmap, T key) : **base**(bitmap, key)

{ }

**public** SinglePermutation<**int**[]> Encrypt()

{

Bitmap Cimage = new Bitmap(ImageInfo.Image);

**for** (**int** i = 0; i < ImageInfo.Width \* ImageInfo.Height / (Key **as** **int**[]).Length; i++)

**for** (**int** j = 0; j < (Key **as** **int**[]).Length; j++)

{

**int** x = (i \* (Key **as** **int**[]).Length + (Key **as** **int**[])[j]) / ImageInfo.Width;

**int** y = (i \* (Key **as** **int**[]).Length + (Key **as** **int**[])[j]) % ImageInfo.Width;

**int** newx = (i \* (Key **as** **int**[]).Length + j) / Cimage.Width;

**int** newy = (i \* (Key **as** **int**[]).Length + j) % Cimage.Width;

Cimage.SetPixel(newx, newy, ImageInfo.Image.GetPixel(x, y));

}

**return** new SinglePermutation<**int**[]>(Cimage, Key **as** **int**[]);

}

**public** SinglePermutation<**int**[]> Decrypt()

{

Bitmap image = new Bitmap(ImageInfo.Image);

**for** (**int** i = 0; i < image.Width \* image.Height / (Key **as** **int**[]).Length; i++)

**for** (**int** j = 0; j < (Key **as** **int**[]).Length; j++)

{

**int** x = (i \* (Key **as** **int**[]).Length + j) / ImageInfo.Width;

**int** y = (i \* (Key **as** **int**[]).Length + j) % ImageInfo.Width;

**int** newx = (i \* (Key **as** **int**[]).Length + (Key **as** **int**[])[j]) / image.Width;

**int** newy = (i \* (Key **as** **int**[]).Length + (Key **as** **int**[])[j]) % image.Width;

image.SetPixel(newx, newy, ImageInfo.Image.GetPixel(x, y));

}

**return** new SinglePermutation<**int**[]>(image, Key **as** **int**[]); ;

}

**public** **static** **int**[] GenKey(**int** count)

{

**int**[] key = new **int**[count];

**for** (**int** i = 0; i < count; i++)

key[i] = i;

key = Shuffle(key);

**return** key;

}

**private** **static** **int**[] Shuffle(**int**[] array)

{

Random rng = new Random();

**int** n = array.Length;

**while** (n > 1)

{

**int** k = rng.Next(n);

n--;

**int** temp = array[n];

array[n] = array[k];

array[k] = temp;

}

**return** array;

}

}

**class** Gamma<T> : Symmetric<T>

{

**private** **const** **int** abc = 256;

**public** **int**[] Message { **get**; **set**; }

**private** **int** Length { **get**; }

**public** Gamma(Bitmap bitmap, T key, **int** length) : **base**(bitmap, key)

{

Message = ImageInfo.ToInt32Array();

Length = length;

}

**public** Gamma<**int**> Encrypt()

{

**int**[] key = LFSR(Convert.ToInt32(Key), Length);

**for** (**int** i = 0; i < Message.Length; i++)

Message[i] = (((Message[i] + key[i % key.Length]) % abc) + abc) % abc;

**return** new Gamma<**int**>(ImageInfo.ToImage(Message), Convert.ToInt32(Key), Length);

}

**public** Gamma<**int**> Decrypt()

{

**int**[] key = LFSR(Convert.ToInt32(Key), Length);

**for** (**int** i = 0; i < Message.Length; i++)

Message[i] = (((Message[i] + abc - key[i % key.Length]) % abc) +abc) % abc;

**return** new Gamma<**int**>(ImageInfo.ToImage(Message), Convert.ToInt32(Key), Length);

}

**private** **int** LFSR(**int** seed)

{

**int** b0 = (seed & 0b00000001);

**int** b2 = (seed & 0b00000100) >> 2;

**int** b3 = (seed & 0b00001000) >> 3;

**int** b4 = (seed & 0b00010000) >> 4;

**int** b7 = b4 ^ b3 ^ b2 ^ b0;

**return** (b7 << 8) | (seed >> 1);

}

**private** **int**[] LFSR(**int** seed, **int** len)

{

**int**[] array = new **int**[len];

**for** (**int** i = 0; i < len; i++)

{

array[i] = LFSR(seed);

seed = array[i];

}

**return** array;

}

}

}