ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9
MD5

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать 1 из ниже приведенных алгоритмов хеширования:

1. MD5

Для реализованной хеш функции провести следующие тесты:

- Провести сгенерировать 1000 пар строк длинной 128 символов отличающихся друг от друга 1,2,4,8,16 символов и сравнить хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности. Результаты для каждого количества отличий нанести на график, где по оси x кол-во отличий, а по оси y максимальная длинна одинаковой последовательности.
- Провести N = 10^i(i от 2 до 6) генерацию хешей для случайно сгенерированных строк длинно 256 символов, и выполнить поиск одинаковых хешей в итоговом наборе данных, результаты привести в таблице где первая колонка это N генераций, а вторая таблица наличие и кол-во одинаковых хешей, если такие были.
- Провести по 1000 генераций хеша для строк длинной n (64, 128, 256, 512, 1024, 2048, 4096, 8192)(строки генерировать случайно для каждой серии), подсчитать среднее время и построить зависимость скорости расчета хеша от размера входных данных

Описание метода/модели.

Хеш-таблицы

Хеш-табли́ца — структура данных, реализующая интерфейс ассоциативного массива. При этом по сути это простая таблица, в которой есть столбец ключа, и столбец значения. Каждый ключ в такой таблице соответствует какому-то значению и новые значения вставляются в такую таблицу всегда сопоставляясь с каким-то ключом

Хранение

Хеш-таблица хранится в памяти как массив длинною некоторая M. При этом для set в таком массиве хранится напрямую значение добавленные в таблицу которое является и ключом, и значением одновременно, в свою очередь для map, каждой ячейке соответствует какой-то ключ, а сама ячейка хранит уже значение, ассоциирующееся с этим ключом.

Как же определяется позиция записываемого значения?

Для определения позиции значения требуется использовать какую-либо функцию, которая превратит наш ключ в номер ячейке в массиве значений. Самый простой вариант, о котором можно подумать, это преобразования значения к числу и взятие модуля от этого значения. Т.е.:
i = int(key) % M,
где i – позиция записываемого элемента в массвие,
M – размер массива.

Коллизии

Стоит обратить внимание, что в предыдущей формуле M может быть различного значения и напрямую зависит от размера хранимого значения. Поэтому, если взять M достаточно большим, то количество элементов в массиве будет неподъемным ни для одного компьютера. С другой стороны, если взять M достаточно малым, то мы обнаружим, что функция будет давать одинаковый результат на совершенно разные входные данные:
Положим, что key = 25, а M = 20: тогда i = 5.
Положим, что key = 45, при том же M: тогда i = 5.
Как мы видим, результат одинаковый, а значит мы будем записывать данные в одну и ту же ячейку.

Возможно ли создать какую-либо функцию, которая гарантированно вычислит уникальную позицию для любых пар элементов? Нет. Такая ситуация называется коллизией, и в рамках структуры хеш-таблицы, принимается, что могут существовать пересечения значений индекса, поэтому за индексом всегда скрывается дополнительная структура, либо список, либо дерево, либо другая хеш-таблица.

Часто используют функцию: h(k) = ((k*A) % P) % M), где, P – случайное, простое, большое число, A – случайное, число от 1 до P – 1. Эта функция дает вероятность коллизий в пределах

Хеширование

Хеширование, это применение особого алгоритма к некоторым входным данным произвольного типа, преобразующего их в битовую строку установленной длинны, строка часто выводиться в форме шестнадцатеричного чиста.

Алгоритмом, который применяется для хеширования называют хеш-функцией или функцией свертки.

Результат работы алгоритма хеширования называют хешем, хеш-кодом, хеш-суммой.

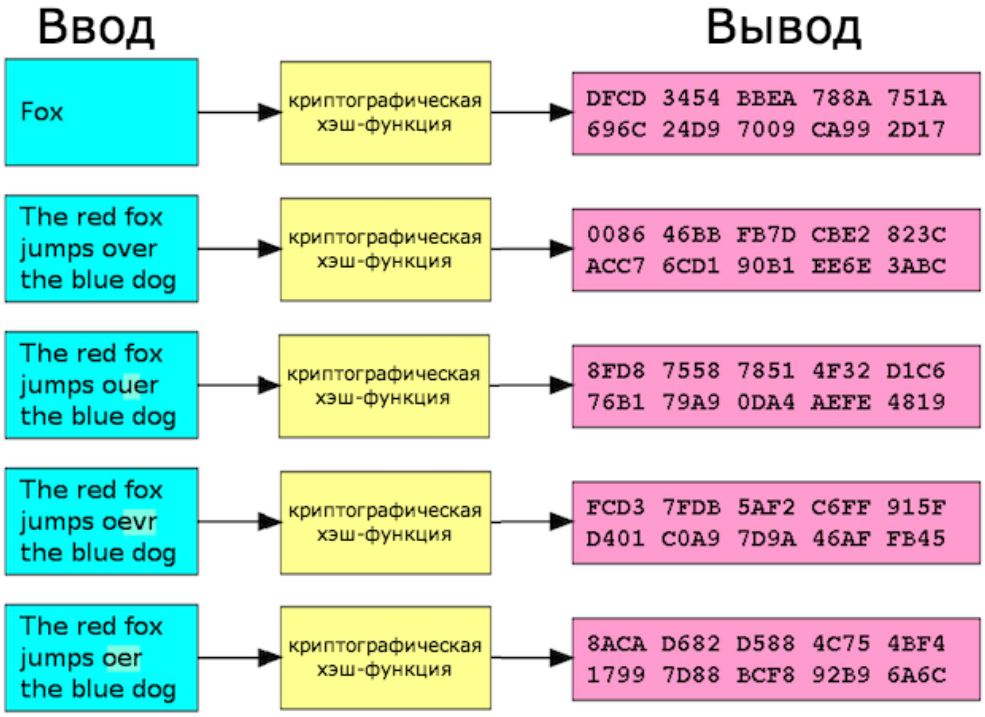

Рис.1.Хеширование

- при построении ассоциативных массивов;
- при поиске дубликатов в сериях наборов данных;
- при построении уникальных идентификаторов для наборов данных;
- при вычислении контрольных сумм от данных для последующего обнаружения в них ошибок, возникающих при хранении и/или передаче данных;
- при сохранении паролей в системах защиты в виде хеш-кода;
- при выработке электронной подписи;

По умолчанию, базовая реализация хэш-функции должна удовлетворять следующим свойствам:

Детерминированность – т.е. функция должна, в обязательном порядке, выдавать одинаковый вывод на одинаковы ввод.

Скорость вычисления – функция должна быстро вычисляться, что бы ее можно было эффективно использовать для итеративных и постоянно возникающих процессов.

Минимальное количество коллизий. – все хеширующие функции не гарантируют полное отсутствие колизий на бесконечном наборе входных данных.

Коллизии это пересечение значений выдаваемых хеш-функцией как результат своей работы для двух абсолютно разных вводов.

№	Ввод	Итоговый хеш	Наличие коллизий
1	Первый набор данных	F98C469D0CF7E5D466DF06D339990EE138674B6D	Коллизия
2	Второй набор данных	CF1E415CEEA860D81EC9233D626AE90EBAA823A7	Нет коллизий
3	Третий набор данных	F98C469D0CF7E5D466DF06D339990EE138674B6D	Коллизия

Рис.2.Хеш-функции

Универсальное хеширование

Универсальным хешированием называют алгоритм в котором хеш-функция выбирается случайным образом из заранее заданного набора хеш-функций что обеспечивает значительное сокращение коллизий и более равномерное хеширование.

Идеальная хеш-функци

Идеальной хеш-функцией называется и считается та хеш функция которая никогда не дает одинаковых результатов на различные входные данные, т.е. никогда не возникает коллизий.Для обеспечения такого результата хеширование производиться в 2 уровня.

Разумеется, есть ограничение: идеальная хешь-функция существует только для конечного набора входных данных, т.е. предполагается что после создания контейнера состав ключей уже не меняется, что будет гарантировать быстроту и однозначность доступа к данным, но вызывать трудоемкую перегенерацию при каждом добавлении.

Криптографические хеш-функции

Криптографическая хэш-функция — это специальный класс хэш-функций, который имеет различные свойства, необходимые для криптографии.

Свойство 1: Коллизионная устойчивость

Это свойство обозначает, что для данной функции еще не было найдено данных которые создают коллизию, в общем же смысле означает, что для поиска коллизий необходимо потратить огромное количество времени.

Свойство 2: Устойчивость к поиску первого прообраза

Это свойство означает, что для данной функции крайне сложно выполнить операцию поиска исходного сообщения имея итоговый хеш, причем это свойство должно соблюдаться вне зависимости от того будет ли поиск выполнятся перебором или каким-либо образом раскручиваться из итогового результата.

Свойство 3: Устойчивость к поиску второго прообразу

Это свойство означает, что для данной функции крайне сложно выполнить операцию поиска исходного сообщения имея итоговый хеш и исходное сообщение дающее коллизию.

Доказуемо безопасные

Аналитическое доказательство выполнения приведенных ранее свойств весьма сложная задача, а само словосочетание доказумо безопасная функция означает, что задачи поиска колизии и поиска первого прообраза являются нерешаемыми за полиномиальное время и время требуемое на их решение экспоненциально зависит от размера итогового ключа.

У доказательного подхода есть определенные недостатки:

- Алгоритмы, которые на данный момент являются доказуемо безопасными вычислительно сложны, что не позволяет использовать их в потоковых задачах.
- Трудоемкость разработки и обоснования доказуемо безопасной функции огромна.
- Само доказательство строиться на сведении решения задач поиска коллизий, первого и второго прообразов к задачам, которые имеют требуемую сложность в среднем и худшем случаях, что потенциально делает эти функции уязвимыми.

Примеры таких функций:

- VSH – основана на сложности задачи нахождения нетривиальных квадратных корней по модулю составного числа
- ECOH – основана на идее эллиптических кривых, задаче о сумме подмножеств и суммировании полиномов
- FSB – показано что взломать функцию так же сложно как решить задачу известную как регулярное синдромное декодирование.
- Knapsack-based hash functions – это функции которые основаны на задаче о рюкзаке.

Идеальная криптографическая хеш-функция

Идеальной можно назвать ту функцию, которая отвечает следующим свойствам:

- Детерминированность;
- Высокая скорость вычисления значения хеш-функции для любого заданного сообщения;
- Невозможность сгенерировать сообщение из его хеш-значения, за исключением попыток создания всех возможных сообщений;

- Наличие лавинного эффекта;
- Невозможность найти два разных сообщения с одинаковыми хеш-значениями.
- Высокая энтропия информации на выходе.

Лавинны эффект

Лавинным эффектом называют ситуацию, в которой при генерации хеша для двух слабо отличающихся друг от друга изначальных набора данных, результат будет отличаться колоссально.

Лавинным	f0702fb787271faa27f4b6ae3f9c9d5f037df97575489b2de136d40cdcb7ea5e
лавинным	ae4ab7df13b47ed51303569d00c0f8a57e95d5a9f5689747de19fa45dd95195a
лавиннын	d5607d1591c52e9b1fb319dd0e0ca84f99b3d83c93545cc653fd60fd3e9001f2

Рис.3.Лавинный эффект

Соль

Соль, это дополнительные данные которые вносятся в хеш функцию и генерируются случайным образом.

Соль делится на статическую – одинаковая на каждый хеш и динамическую, разная для каждого хеша.

Сама соль, в случае если она динамическая, как правило записывается рядом с итоговым хешом и нужна для повторного хеширования данных при поиске.

Наличие соли защищает от атаки, связанной с перебором по словарю возможных исходных значений (например, перебор по словарю популярных паролей), т.е. позволяет скрыть факт использования одинаковых входных значения.

Проблема соли

Малая длина соли и низкая энтропия

Если соль имеет малую длину, злоумышленнику будет легко создать радужную таблицу, состоящую из всех возможных солей определённой длины, добавляемых к каждому вероятному паролю.

Повторное использование соли для разных прообразов

Хотя использование статической соли для одинаковых прообразов сделает некоторые существующие радужные таблицы бесполезными, следует заметить, что если соль статично вписана в исходный код популярного продукта, то она может быть рано или поздно извлечена, после чего на основе этой соли можно создать новую радужную таблицу.

MD5

MD5 (англ. Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского технологического института (Massachusetts Institute of Technology, MIT) в 1991 году. Предназначен для создания «отпечатков» или дайджестов сообщения произвольной длины и последующей проверки их подлинности. Широко применялся для проверки целостности информации и хранения хешей паролей.

MD5 обрабатывает сообщение переменной длины в выходной сигнал фиксированной длины, состоящий из 128 бит. Входное сообщение разбивается на фрагменты по 512-битных блоков (шестнадцать 32-битных слов); сообщение дополняется таким образом, чтобы его длина была кратна 512. Заполнение работает следующим образом: сначала в конец сообщения добавляется один бит, 1. За этим следует столько нулей, сколько требуется для увеличения длины сообщения до 64 бит, что меньше, чем кратно 512. Оставшиеся биты заполняются 64 битами, представляющими длину исходного сообщения по модулю 264.

Основной алгоритм MD5 работает со 128-битным состоянием, разделенным на четыре 32-битных слова, обозначаемых A, B, C и D. Они инициализируются определенными фиксированными константами. Затем основной алгоритм использует каждый 512-битный блок сообщения по очереди для изменения состояния. Обработка блока сообщений состоит из четырех аналогичных этапов, называемых раундами; каждый раунд состоит из 16 аналогичных операций, основанных на нелинейной функции F, модульном сложении и повороте влево. На рисунке 1 показана одна операция в рамках раунда. Существует четыре возможные функции; в каждом раунде используется своя:

$$F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D)$$
$$G(B,C,D) = (B \wedge D) \vee (C \wedge \neg D)$$
$$H(B,C,D) = B \oplus C \oplus D$$
$$I(B,C,D) = C \oplus (B \vee \neg D)$$

Рис.4.Функции на Раунде

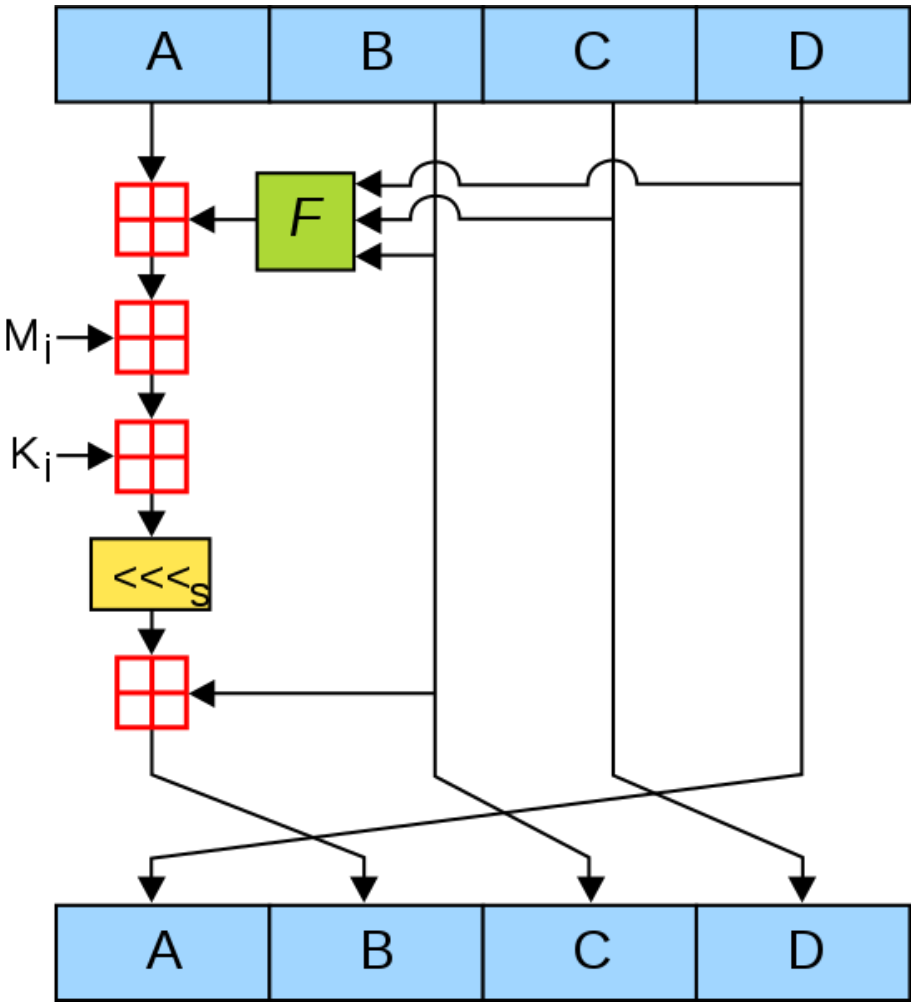

Рис.5. Одна операция MD5.

MD5 состоит из 64 таких операций, сгруппированных в четыре раунда по 16 операций. F - нелинейная функция; в каждом раунде используется одна функция. Mi обозначает 32-разрядный блок ввода сообщения, а Ki обозначает 32-разрядную константу, различную для каждой операции. <<<s обозначает поворот левого бита на s мест; s меняется для каждой операции. ⊞ обозначает сложение по модулю 232.

Примеры MD5-хешей

Хеш содержит 128 бит (16 байт) и обычно представляется как последовательность из 32 шестнадцатеричных цифр.

Несколько примеров хеша:

```
MD5("md5") = 1BC29B36F623BA82AAF6724FD3B16718
```

Даже небольшое изменение входного сообщения (в нашем случае на один бит: ASCII символ «5» с кодом 3516 = 0001101012 заменяется на символ «4» с кодом 3416 = 0001101002) приводит к полному изменению хеша. Такое свойство алгоритма называется лавинным эффектом.

```
MD5("md4") = C93D3BF7A7C4AFE94B64E30C2CE39F4F
```

Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

Код Бинарной кучи:

```
namespace MD5_Lib
{
    /// <summary>
    /// MD5
    /// </summary>
    public class MD5_h
    {
     /// <summary>
     /// Сдвиг влево
     /// </summary>
        static int[] s = new int[64] {
            /* 0..15*/7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,
             /* 16..31*/5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,
             /* 32..47*/4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,
             /* 48..63*/6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21
        };

        /// <summary>
        /// Таблица констант по "2^32 × abs (sin(i + 1))"
        /// </summary>
        static uint[] T = new uint[64] {
            0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee,
            0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501,
            0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be,
            0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821,
            0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
            0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
            0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
            0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
            0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c,
            0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
            0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,
            0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
            0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039,
            0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1,
            0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1,
            0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391
        };
        /// <summary>
        /// Циклический сдвиг влево
        /// </summary>
        /// <param name="x">Сдвигаемые данные</param>
        /// <param name="c">Таблица сдвига</param>
        /// <returns>Сдвинутые влево данные</returns>
        public static uint leftRotate(uint x, int c)
        {
            return (x << c) | (x >> (32 - c));
        }

        /// <summary>
        /// Хеш-функция MD5
        /// </summary>
        /// <param name="input">Входные данные</param>
        /// <returns>готовый хеш</returns>
        public string Calculate(byte[] input)
        {
            uint a0 = 0x67452301;   // A
            uint b0 = 0xefcdab89;   // B
            uint c0 = 0x98badcfe;   // C
            uint d0 = 0x10325476;   // D
            //дозаполняем до кратности 64-ем байтам(или 512 битам)
            var processedInputBuilder = new List<byte>(input) { 0x80 };//добавляем 1 к изначальному Input
```

```
        while (processedInputBuilder.Count % 64 != 56) processedInputBuilder.Add(0x0);//добавляем нули до заполнения 56 байтами(448 битами)
        processedInputBuilder.AddRange(BitConverter.GetBytes((long)input.Length * 8)); // заполняем оставшиеся 8 байтов(64 битами) символами длиной с изначальный инпут
        var processedInput = processedInputBuilder.ToArray();
        //Разбиваем подготовленное сообщение на 64-байтные (512 - битные) "куски"
        for (int i = 0; i < processedInput.Length / 64; ++i)
        {
            uint[] M = new uint[16];//массив для хранения input разделенного на 16 частей 4-байтовых(32-битных) слов
            for (int j = 0; j < 16; ++j)
                M[j] = BitConverter.ToUInt32(processedInput, (i * 64) + (j * 4));

            // инициализация 4рёх слов для текущего куска
            uint A = a0, B = b0, C = c0, D = d0, F = 0, g = 0;

            // основные операции
            for (uint k = 0; k < 64; ++k)//проходимся пао каждому байту(биту)
            {
                if (k <= 15)//1ый раунд
                {
                    F = (B & C) | (~B & D);
                    g = k;//сдвиг по k
                }
                else if (k >= 16 && k <= 31)//второй раунд
                {
                    F = (D & B) | (~D & C);
                    g = ((5 * k) + 1) % 16;
                }
                else if (k >= 32 && k <= 47)//третий раунд
                {
                    F = B ^ C ^ D;
                    g = ((3 * k) + 5) % 16;
                }
                else if (k >= 48)//четвертый раунд
                {
                    F = C ^ (B | ~D);
                    g = (7 * k) % 16;
                }
                //меняем местами
                var dtemp = D;
                F = (A + F + T[k] + M[g]);
                D = C;
                C = B;
                B = B + leftRotate(F, s[k]);//получение F и битовый сдвиг
                A = dtemp;
            }
            // Прибавляем результат текущего "куска" к общему результату
            a0 += A;
            b0 += B;
            c0 += C;
            d0 += D;
        }

        return GetByteString(a0) + GetByteString(b0) + GetByteString(c0) + GetByteString(d0);//результат
    }
    /// <summary>
    /// Перевод из байтов в строку
    /// </summary>
    /// <param name="x">Переводимая строка</param>
    /// <returns>Байты в формате строки</returns>
    private static string GetByteString(uint x)
    {
        return String.Join("", BitConverter.GetBytes(x).Select(y => y.ToString("x2")));
    }
  }
}
```

Реализация:

Алгоритм хеширования MD5 я реализовал в классе MD5_h c, который содержит

1) Поле s, таблицу для циклических сдвигов влево, разделенную на 4 части для каждого раунда
2) Таблицу констант T, построенную по формуле 2^32 × abs (sin(i + 1)) для 64 элементов
3) Метод leftRotate, циклические сдвигающую влево переданные ей даныне.
4) Метод Calculate, принимающий массив байтов и реализующий внури себя основной функционал MD4
    a. Инициализирую «инициализирующий вектор» из A, B, C, D, которые будут хранить результаты промежуточных вычислений
    b. Добавляю длину сообщения до кратного 512, чтобы потом разделить его на блоки. Делаю это сначала дополнив единицей, а оставшуюся часть нулями до 448 бита, 64 после него заполняются длиной изначального инпута.
    c. Как мы закончили с подготовительной частью, переходим к основной, где мы делим наши входные данные на 512-битные куски, которые мы еще делим на 16 кусков по 32 бита каждый. Ициализируем 4-ре слова для текущего куска и проходимся по каждому биту, который, в зависимости от его места, запускает цепочку раундов.
    d. От 0 до 16 (в байтах) Это первый раунд, в котором F = (B & C) | (~B & D);
    e. От 17 до 32(байт. Второй раунд, в котором F= (D & B) | (~D & C);
    f. От 33 до 48(байт). Третий раунд, в котором F= B ^ C ^ D;
    g. От 49 до 64(байт). Четвертый раунд, в котором F= C ^ (B | ~D);
    h. Получаем F = (A + F + T[k] + M[g]) и B = B + leftRotate(F, s[k]);
    i. Меняем местами D = C, C = B, A = D
    j. Закончив с данным куском, добавляем текущий результат к общему
5) Метод GetByteString возвращает переведенные обратно в тип string данные
6) Функция Test1 проводит сгенерирацию 1000 пар строк длинной 128 символов отличающихся друг от друга 1,2,4,8,16 символов и сравниает хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности.
7) Функция Test2 проводит N = 10^i(i от 2 до 6) генерацию хешей для случайно сгенерированных строк длинно 256 символов, и выполняет поиск одинаковых хешей в итоговом наборе данных
8) Функция Test3 проводит по 1000 генераций хеша для строк длинной n (64, 128, 256, 512, 1024, 2048, 4096, 8192) подсчитывает среднее время.

Результаты:

1) Измеряем Максимальную длину одинаковой последовательности в парах хеша с n-ным кол-вом отличающихся элементов и наносим на график

Таблица 1

Макс. Длина одинаковой последовательности в зависимости от отличий в исходном слове

Макс. длина послед.	Кол-во отличий
5	1
4	2
5	4
5	8
5	16

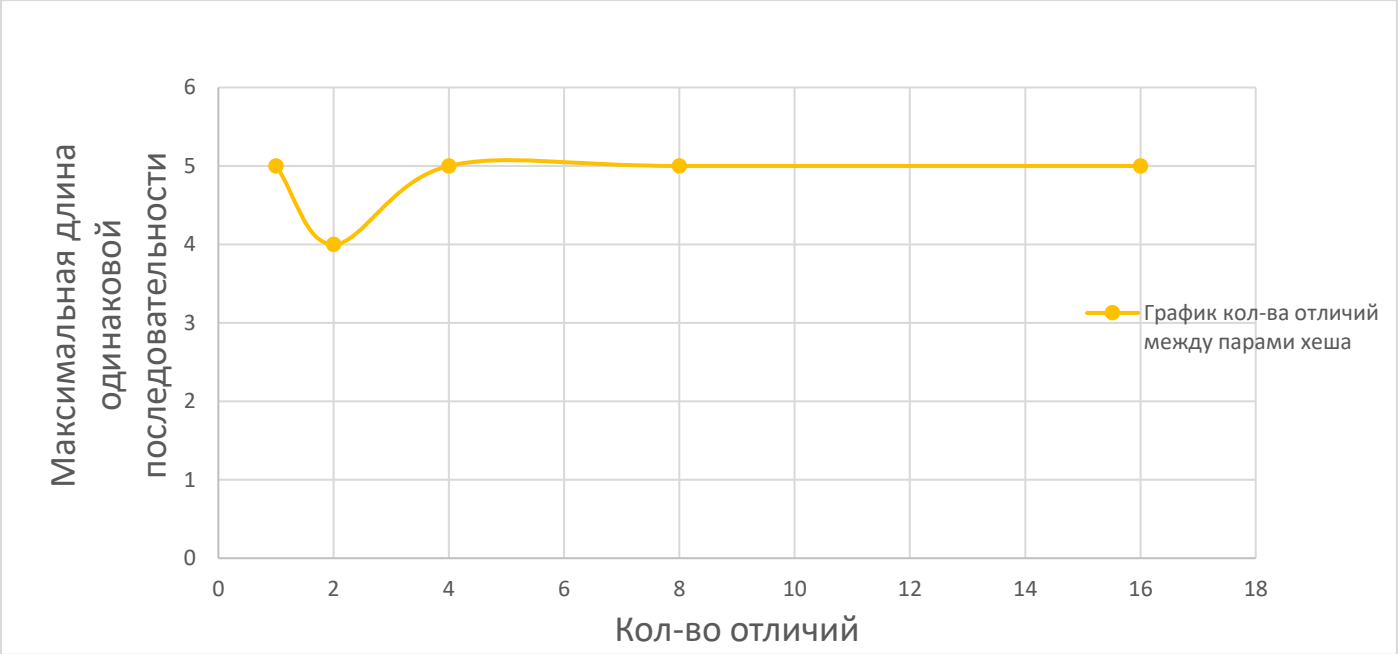

Рис.6. График кол-ва изменения максимальной одинаковой последовательности в хеше в зависимости от кол-ва отличий между парами в изначальных данных

На графике видно, что даже при отличии на один символ итоговый хеш сильно изменяется.

2) Таблица одинаковых хешей среди рандомно сгенерированных наборов на N=10^(от i=2 до 6) хешей по 256 символов каждый

Таблица 2

Одинаковые хеши

10^	N-наборов	Одинаковых хешей
2	100	0
3	1000	0
4	10000	0
5	100000	0
6	1000000	0

По таблице видно, что на всех наборах данных не было обнаружено коллизии, что говорит о хорошей работе алгоритма хеширования

3) Замерил среднее время хеширования для строк длинной (64, 128, 256, 512, 1024, 2048, 4096, 8192).

Таблица 3

Время среднего хеширования строк

Время	Длина строк
0.006	64
0.012	128
0.021	256
0.051	512
0.133	1024

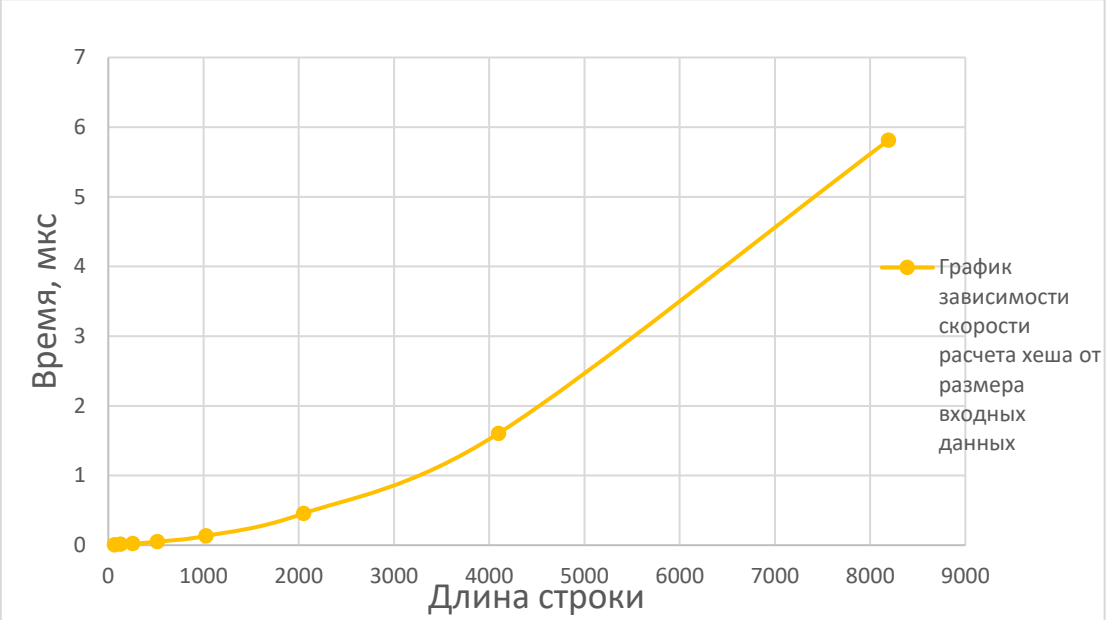

Рис.7.График зависимости скорости расчета хеша от размера входных данных

Видно, что время выполнения хеширование возрастает от кол-ва элементов в изначальном слове.

4)Работа реализованного MD5

```
MD5("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6
```

Рис.8.1-ый Пример из вики

```
C:\Users\torna\source\repos\C sharp\algorithms\Laba9v2\Laba9\bin\Debug\net7.0\Laba9.exe
Введите сообщение
The quick brown fox jumps over the lazy dog
9e107d9d372bb6826bd81d3542a419d6
Введите сообщение
```

Рис.9.1-ый Результат

```
MD5("The quick brown fox jumps over the lazy dog.") =
e4d909c290d0fb1ca068ffaddf22cbd0
```

Рис.10.2-ой пример из вики

```
The quick brown fox jumps over the lazy dog.
e4d909c290d0fb1ca068ffaddf22cbd0
```

Рис.11.2-ой Результат

Заключение.

Реализация MD5 не особо сложная, если следовать алгоритму, но вот с понимаем происходящего в ней могут возникнуть сложности. Как показали результаты, данная хеш-функция выдает даже на слегка изменённое слово совершенно другой результат и справляется с генерацией хешей для большого количества данных. При этом, время, требуемое на генерацию хеша, зависит от длины строки и возрастает с её увеличением.