Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Лабораторная работа №5. Компьютерная реализация хеш-функции на примере MD5

Выполнил: cтудент гр. 853501

Яковлев А.Б.

Проверил:

Протько М.И.

Минск 2021

# Постановка задачи и описание алгоритма

**Цель:** реализовать программные средства формирования хэш-функций на примере алгоритма MD5.

Хеш-функция, или функция свёртки – функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

Хеш-функции применяются в следующих случаях:

* при построении ассоциативных массивов;
* при поиске дубликатов в сериях наборов данных;
* при построении уникальных идентификаторов для наборов данных;
* при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
* при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
* при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

и др.

В общем случае (согласно принципу Дирихле) нет однозначного соответствия между хеш-кодом (выходными данными) и исходными (входными) данными. Возвращаемые хеш-функцией значения (выходные данные) менее разнообразны, чем значения входного массива (входные данные). Случай, при котором хеш-функция преобразует более чем один массив входных данных в одинаковые сводки, называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Существует множество алгоритмов хеширования, отличающихся различными свойствами. Примеры свойств:

* разрядность;
* вычислительная сложность;
* криптостойкость.

Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшим примером хеш-функции может служить «обрамление» данных циклическим избыточным кодом (англ. CRC, cyclic redundancy code).

MD5 (англ. Message Digest 5) — 128-битный алгоритм хеширования, разработанный профессором Рональдом Л. Ривестом из Массачусетского Технологического Института (MIT, Massachusetts Institute of Technlogy) в 1991 году.

Используется для проверки подлинности опубликованных сообщений путем сравнения дайджеста сообщения с опубликованным. Эту операцию называют «проверка хеша» (hashcheck).

Описание алгоритма:

1. Добавление нулевых битов

В исходную строку дописывают единичный байт 0х80, а затем дописывают нулевые биты, до тех пор, пока длина сообщения не будет сравнима с 448 по модулю 512. То есть дописываем нули до тех пор, пока длина нового сообщения не будет равна [длина] = (512\*N+448), где N — любое натуральное число, такое, что это выражение будет наиболее близко к длине блока.

1. Добавление длины

Далее в сообщение дописывается 64-битное представление длины исходного сообщения.

1. Инициализация MD буфера

На этом шаге инициализируется буффер

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

Как можно заметить буффер состоит из четырех констант, предназначенный для сбора хэша.

1. Обработка сообщения в виде 16 блоков-слов

На четвертом шаге в первую очередь определяется 4 вспомогательные логические функции, которые преобразуют входные 32-битные слова, в, как ни странно, в 32-битные выходные.

F(X,Y,Z) = XY v not(X) Z

G(X,Y,Z) = XZ v Y not(Z)

H(X,Y,Z) = X xor Y xor Z

I(X,Y,Z) = Y xor (X v not(Z))

Также на этом шаге реализуется так называемый «белый шум» — усиление алгоритма, состоящее 64 элементного массива, содержащего псевдослучайные числа, зависимые от синуса числа i:

T[i]=4,294,967,296\*abs(sin(i))

Далее копируем каждый 16-битный блок в массив X[16] и производим манипуляции:

AA = A

BB = B

CC = C

DD = D

Затем происходят преобразования-раунды, которых всего будет 4. Каждый раунд состоит из 16 элементарных преобразований, которые в общем виде можно представить в виде [abcd k s i], которое, в свою очередь, можно представить как A = B + ((A + F(B,C,D) + X[k] + T[i]) <<< s), где

A, B, C, D — регистры

F(B,C,D) — одна из логических функций

X[k] — k-тый элемент 16-битного блока.

T[i] — i-тый элемент таблицы «белого шума»

<<< s — операция циклического сдвига на s позиций влево.

В конце суммируем результаты вычислений:

A = A + AA

B = B + BB

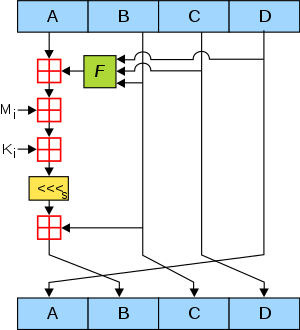
C = C + CC

D = D + DD

1. Вывод

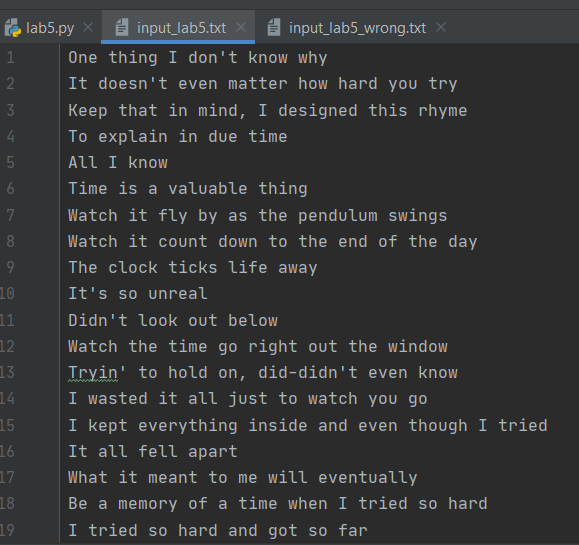
Выводя побайтово буффер ABCD начиная с A и заканчивая D получим хэш.

# Блок-схемы алгоритмов

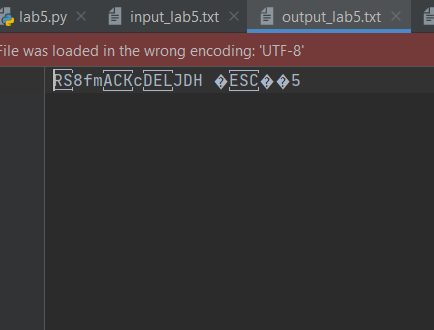


# Результаты выполнения программы

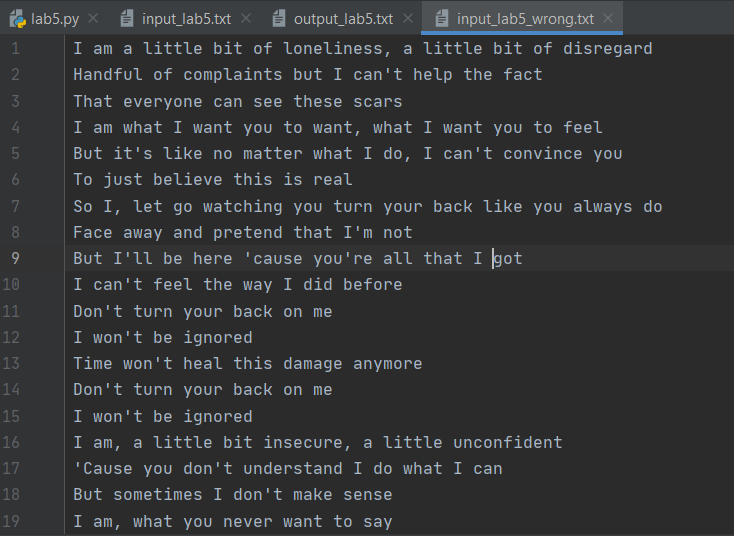
Файл с входными данными:



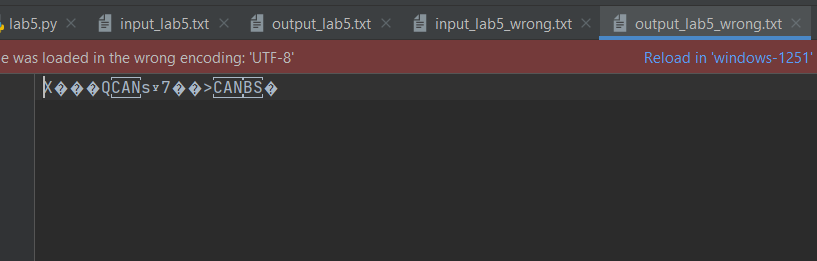
Хеш первого сообщения:



Второе сообщение сообщение:



Расшифрованное второе сообщение



# Исходный код

import typing as tp  
  
from bitarray import bitarray  
from bitarray.util import ba2int, int2ba  
  
from functions import left\_cycle\_shift  
  
  
class MD5:  
 s = [7, 12, 17, 22, 7, 12, 17, 22,  
 7, 12, 17, 22, 7, 12, 17, 22,  
 5, 9, 14, 20, 5, 9, 14, 20,  
 5, 9, 14, 20, 5, 9, 14, 20,  
 4, 11, 16, 23, 4, 11, 16, 23,  
 4, 11, 16, 23, 4, 11, 16, 23,  
 6, 10, 15, 21, 6, 10, 15, 21,  
 6, 10, 15, 21, 6, 10, 15, 21]  
  
 K = [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]  
  
 def \_\_init\_\_(self) -> None:  
 self.a\_register = bitarray()  
 self.b\_register = bitarray()  
 self.c\_register = bitarray()  
 self.d\_register = bitarray()  
  
 def \_\_reset\_registers(self) -> None:  
 self.a\_register = int2ba(0x01234567, length=32)  
 self.b\_register = int2ba(0x89ABCDEF, length=32)  
 self.c\_register = int2ba(0xFEDCBA98, length=32)  
 self.d\_register = int2ba(0x76543210, length=32)  
  
 def hash\_message(self, message: bitarray) -> bitarray:  
 extended\_message = message.copy()  
 extended\_message.extend([1])  
  
 if len(extended\_message) % 512 != 448:  
 bits\_to\_extend\_amount = ((447 + 512) - len(message) % 512) % 512  
 extended\_message.extend([0] \* bits\_to\_extend\_amount)  
 extended\_message.extend(int2ba(len(message) % (2 \*\* 64), length=64))  
  
 self.\_\_reset\_registers()  
 for chunk\_no in range(len(extended\_message) // 512):  
 temp\_a\_register = self.a\_register.copy()  
 temp\_b\_register = self.b\_register.copy()  
 temp\_c\_register = self.c\_register.copy()  
 temp\_d\_register = self.d\_register.copy()  
  
 for i in range(64):  
 logical\_function\_result = bitarray()  
 g = 0  
 if i < 16:  
 logical\_function\_result = (temp\_b\_register & temp\_c\_register) | (  
 (~temp\_b\_register) & temp\_d\_register)  
 g = i  
 elif 16 <= i < 32:  
 logical\_function\_result = (temp\_d\_register & temp\_b\_register) | (  
 (~temp\_d\_register) & temp\_c\_register)  
 g = (5 \* i + 1) % 16  
 elif 32 <= i < 48:  
 logical\_function\_result = temp\_b\_register ^ temp\_c\_register ^ temp\_d\_register  
 g = (3 \* i + 5) % 16  
 elif 48 <= i < 64:  
 logical\_function\_result = temp\_c\_register ^ (temp\_b\_register | (~temp\_d\_register))  
 g = (7 \* i) % 16  
 current\_word = extended\_message[chunk\_no \* 512 + g \* 32: chunk\_no \* 512 + (g + 1) \* 32]  
  
 function\_result = int2ba(  
 (ba2int(logical\_function\_result) + ba2int(temp\_a\_register) + self.K[i] + ba2int(current\_word)) % (  
 2 \*\* 32),  
 length=32)  
 temp\_a\_register = temp\_d\_register  
 temp\_d\_register = temp\_c\_register  
 temp\_c\_register = temp\_b\_register  
 temp\_b\_register = int2ba(  
 (ba2int(temp\_b\_register) + ba2int(left\_cycle\_shift(function\_result, self.s[i]))) % (2 \*\* 32),  
 length=32)  
  
 self.a\_register = int2ba((ba2int(temp\_a\_register) + ba2int(self.a\_register)) % (2 \*\* 32), length=32)  
 self.b\_register = int2ba((ba2int(temp\_b\_register) + ba2int(self.b\_register)) % (2 \*\* 32), length=32)  
 self.c\_register = int2ba((ba2int(temp\_c\_register) + ba2int(self.c\_register)) % (2 \*\* 32), length=32)  
 self.d\_register = int2ba((ba2int(temp\_d\_register) + ba2int(self.d\_register)) % (2 \*\* 32), length=32)  
 return self.a\_register + self.b\_register + self.c\_register + self.d\_register  
  
 def hash\_file(self, input\_file: tp.BinaryIO, output\_file: tp.BinaryIO) -> None:  
 input\_file\_message = bitarray()  
 input\_file\_message.fromfile(input\_file)  
 hashed\_input\_file\_message = self.hash\_message(input\_file\_message)  
 hashed\_input\_file\_message.tofile(output\_file)  
  
  
def main() -> None:  
 md5 = MD5()  
 with open('input\_files/input\_lab5.txt', 'r') as f:  
 message = bitarray()  
 message.frombytes(f.read().encode('UTF-8'))  
 with open('input\_files/input\_lab5\_wrong.txt', 'r') as f:  
 second\_message = bitarray()  
 second\_message.frombytes(f.read().encode('UTF-8'))  
 first\_hash = md5.hash\_message(message)  
 second\_hash = md5.hash\_message(second\_message)  
 print(first\_hash == second\_hash)  
 with open('output\_files/output\_lab5.txt', 'wb') as f:  
 first\_hash.tofile(f)  
 with open('output\_files/output\_lab5\_wrong.txt', 'wb') as f:  
 second\_hash.tofile(f)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# Вывод

В результате выполнения работы был реализован алгоритм хеширования MD5.