Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 3](#_Toc63548272)

[Описание метода/модели. 3](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 8](#_Toc63548275)

# Описание задачи.

**Задание:**

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

2. [SHA1](https://ru.bmstu.wiki/SHA-1_(Secure_Hash_Algorithm_1)#:~:text=Secure%20Hash%20Algorithm%201%20%E2%80%94%20%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC,%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%2C%20%D0%BD%D0%B0%D0%B7%D1%8B%D0%B2%D0%B0%D0%B5%D0%BC%D0%BE%D0%B5%20%D1%82%D0%B0%D0%BA%D0%B6%D0%B5%20%D0%B4%D0%B0%D0%B9%D0%B4%D0%B6%D0%B5%D1%81%D1%82%D0%BE%D0%BC%20%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F.)

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 х 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.

**Теоретическая часть**

**Алгоритм SHA-1**

SHA-1 (Secure Hash Algorithm 1) — это криптографическая хеш-функция, разработанная Национальным институтом стандартов и технологий США (NIST) в 1993 году. Она принимает входное сообщение произвольной длины (до 2⁶⁴ бит) и генерирует фиксированный 160-битный (20-байтный) хеш, представленный в виде 40-символьной шестнадцатеричной строки. SHA-1 широко использовалась для обеспечения целостности данных, цифровых подписей и аутентификации, но в настоящее время считается устаревшей из-за уязвимостей к атакам на коллизии.

**Принцип работы SHA-1**

SHA-1 работает в несколько этапов:

1. **Подготовка сообщения**:
   * К входному сообщению добавляется бит 1 (байт 0x80).
   * Сообщение дополняется нулями до длины, кратной 512 битам, за вычетом 64 бит.
   * Последние 64 бита кодируют длину исходного сообщения в битах (в формате big-endian).
2. **Инициализация регистров**:
   * Используются пять 32-битных регистров (h0, h1, h2, h3, h4), инициализированных стандартными значениями:
     + h0 = 0x67452301
     + h1 = 0xEFCDAB89
     + h2 = 0x98BADCFE
     + h3 = 0x10325476
     + h4 = 0xC3D2E1F0
3. **Обработка блоками**:
   * Сообщение разбивается на блоки по 512 бит (64 байта).
   * Для каждого блока:
     + Блок делится на 16 32-битных слов.
     + Слова расширяются до 80 слов с использованием побитовых операций:  
       w[i] = leftRotate(w[i-3] ^ w[i-8] ^ w[i-14] ^ w[i-16], 1) для i от 16 до 79.
     + Выполняется 80 итераций, разбитых на 4 раунда по 20 итераций. В каждом раунде используются разные логические функции (f) и константы (k):
       - Раунд 1 (0–19): f = (b & c) | ((~b) & d), k = 0x5A827999
       - Раунд 2 (20–39): f = b ^ c ^ d, k = 0x6ED9EBA1
       - Раунд 3 (40–59): f = (b & c) | (b & d) | (c & d), k = 0x8F1BBCDC
       - Раунд 4 (60–79): f = b ^ c ^ d, k = 0xCA62C1D6
     + На каждой итерации обновляются значения рабочих переменных a, b, c, d, e:  
       temp = leftRotate(a, 5) + f + e + k + w[i]  
       Затем переменные сдвигаются: e = d, d = c, c = leftRotate(b, 30), b = a, a = temp.
   * После обработки блока значения h0–h4 обновляются: h0 += a, h1 += b и т.д.
4. **Формирование хеша**:
   * Итоговый хеш формируется как конкатенация h0, h1, h2, h3, h4 в шестнадцатеричном формате.

**Свойства SHA-1**

* **Односторонность**: Вычисление хеша — простая операция, но восстановление исходного сообщения по хешу практически невозможно.
* **Лавинный эффект**: Малейшее изменение входных данных (даже 1 бит) приводит к значительным изменениям в хеше.
* **Устойчивость к коллизиям**: В теории, найти два разных сообщения с одинаковым хешем должно быть сложно. Однако в 2017 году SHA-1 был признан уязвимым: Google продемонстрировал практическую атаку на коллизии с вычислительной сложностью 2⁶³ операций.
* **Фиксированный выход**: Хеш всегда имеет длину 160 бит (40 символов в шестнадцатеричном формате).

**Вычислительная сложность**

* **Временная сложность**:
  + Подготовка сообщения: O(L), где L — длина входного сообщения в битах.
  + Обработка блоков: Каждый блок (512 бит) обрабатывается за фиксированное количество операций (80 итераций). Для N блоков (N ≈ L / 512) сложность обработки составляет O(N) = O(L).
  + Общая сложность: O(L), линейная относительно длины входного сообщения.
  + Формирование хеша: O(1), так как выход фиксирован (160 бит).
* **Пространственная сложность**:
  + Хранилище регистров: 5 × 32 бита = 20 байт.
  + Массив слов: 80 × 32 бита = 320 байт на блок.
  + Хранилище сообщения: O(L) для входного сообщения с padding’ом.
  + Общая сложность: O(L).
* **Устойчивость к атакам**:
  + Коллизии: Теоретическая сложность атаки — 2⁸⁰ операций (по принципу дня рождения), но практическая атака снижена до 2⁶³ операций.
  + Предобраз: Поиск сообщения по хешу требует 2¹⁶⁰ операций (брутфорс).

**Особенности реализации**

* SHA-1 работает с побитовыми операциями (AND, OR, XOR, NOT, циклический сдвиг), что делает её эффективной на аппаратном уровне.
* В реализации используется 32-битная арифметика (тип uint32\_t в C++), что соответствует спецификации алгоритма.
* Входное сообщение интерпретируется как массив байтов, что позволяет обрабатывать любые символы (например, ASCII).
* Реализация на C++ включает обработку блоков, побитовые операции и форматирование результата в шестнадцатеричный вид.

**Преимущества и недостатки**

**Преимущества**:

* Простота реализации.
* Высокая скорость вычисления на небольших данных.
* Широкое историческое применение в криптографии.

**Недостатки**:

* Уязвимость к атакам на коллизии (не рекомендуется для современных криптографических приложений).
* Линейная зависимость времени вычисления от длины входных данных.
* Ограниченная длина входного сообщения (до 2⁶⁴ бит).

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

**//лаб9**

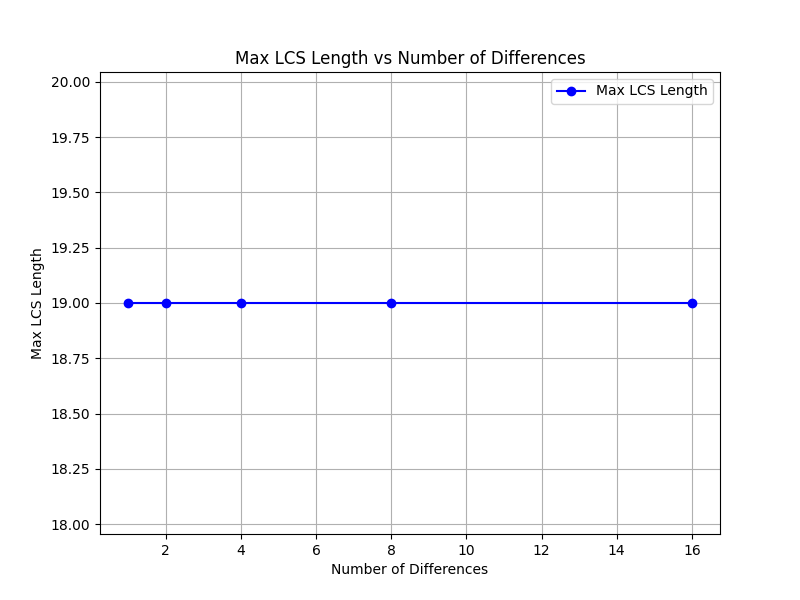
#include <iostream>  
#include <string>  
#include <vector>  
#include <random>  
#include <chrono>  
#include <set>  
#include <algorithm>  
#include <fstream>  
#include <iomanip>  
  
// Функция циклического сдвига влево  
uint32\_t leftRotate(uint32\_t x, uint32\_t n) {  
 return (x << n) | (x >> (32 - n));  
}  
  
// Реализация алгоритма SHA-1  
std::string sha1(const std::string& input) {  
 // Инициализация начальных значений регистров  
 uint32\_t h0 = 0x67452301;  
 uint32\_t h1 = 0xEFCDAB89;  
 uint32\_t h2 = 0x98BADCFE;  
 uint32\_t h3 = 0x10325476;  
 uint32\_t h4 = 0xC3D2E1F0;  
  
 // Подготовка сообщения: добавление бита 1 и padding  
 std::string message = input;  
 uint64\_t originalLength = message.length() \* 8; // Длина в битах  
 message += (char)0x80; // Добавляем бит 1  
 while (message.length() % 64 != 56) { // Дополняем нулями до 448 бит (mod 512)  
 message += (char)0x00;  
 }  
 // Добавляем длину сообщения (64 бита) в big-endian  
 for (int i = 7; i >= 0; i--) {  
 message += (char)((originalLength >> (i \* 8)) & 0xFF);  
 }  
  
 // Обработка сообщения блоками по 512 бит  
 for (size\_t i = 0; i < message.length(); i += 64) {  
 uint32\_t w[80]; // Массив слов для обработки  
 // Разбиваем блок на 16 слов по 32 бита  
 for (int j = 0; j < 16; j++) {  
 w[j] = (message[i + j \* 4] << 24) |  
 (message[i + j \* 4 + 1] << 16) |  
 (message[i + j \* 4 + 2] << 8) |  
 (message[i + j \* 4 + 3]);  
 }  
 // Расширяем 16 слов до 80  
 for (int j = 16; j < 80; j++) {  
 w[j] = leftRotate(w[j - 3] ^ w[j - 8] ^ w[j - 14] ^ w[j - 16], 1);  
 }  
  
 // Инициализация рабочих переменных  
 uint32\_t a = h0, b = h1, c = h2, d = h3, e = h4;  
 // Основной цикл: 80 итераций  
 for (int j = 0; j < 80; j++) {  
 uint32\_t f, k;  
 // Выбор функции и константы в зависимости от раунда  
 if (j < 20) {  
 f = (b & c) | ((~b) & d);  
 k = 0x5A827999;  
 } else if (j < 40) {  
 f = b ^ c ^ d;  
 k = 0x6ED9EBA1;  
 } else if (j < 60) {  
 f = (b & c) | (b & d) | (c & d);  
 k = 0x8F1BBCDC;  
 } else {  
 f = b ^ c ^ d;  
 k = 0xCA62C1D6;  
 }  
 // Обновление переменных  
 uint32\_t temp = leftRotate(a, 5) + f + e + k + w[j];  
 e = d;  
 d = c;  
 c = leftRotate(b, 30);  
 b = a;  
 a = temp;  
 }  
  
 // Обновление регистров  
 h0 += a;  
 h1 += b;  
 h2 += c;  
 h3 += d;  
 h4 += e;  
 }  
  
 // Формирование итогового хеша в шестнадцатеричном формате  
 char result[41];  
 snprintf(result, sizeof(result), "%08x%08x%08x%08x%08x", h0, h1, h2, h3, h4);  
 return std::string(result);  
}  
  
// Генерация случайной строки заданной длины  
std::string generateRandomString(size\_t length) {  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<> dis('a', 'z');  
 std::string s;  
 for (size\_t i = 0; i < length; i++) {  
 s += (char)dis(gen);  
 }  
 return s;  
}  
  
// Создание строки, отличающейся от исходной на заданное количество символов  
std::string createDifferentString(const std::string& base, size\_t diffs) {  
 std::string modified = base;  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<> pos(0, base.length() - 1);  
 std::uniform\_int\_distribution<> letter('a', 'z');  
 std::set<size\_t> positions;  
 // Выбираем уникальные позиции для замены  
 while (positions.size() < diffs) {  
 size\_t p = pos(gen);  
 char newChar = (char)letter(gen);  
 if (modified[p] != newChar) {  
 positions.insert(p);  
 modified[p] = newChar;  
 }  
 }  
 return modified;  
}  
  
// Поиск длины наибольшей общей подпоследовательности (LCS)  
int longestCommonSubsequence(const std::string& s1, const std::string& s2) {  
 int m = s1.length(), n = s2.length();  
 std::vector<std::vector<int>> dp(m + 1, std::vector<int>(n + 1, 0));  
 // Заполнение таблицы динамического программирования  
 for (int i = 1; i <= m; i++) {  
 for (int j = 1; j <= n; j++) {  
 if (s1[i - 1] == s2[j - 1]) {  
 dp[i][j] = dp[i - 1][j - 1] + 1;  
 } else {  
 dp[i][j] = std::max(dp[i - 1][j], dp[i][j - 1]);  
 }  
 }  
 }  
 return dp[m][n];  
}  
  
// Тест 1: Сравнение хешей для строк с разным количеством отличий  
void test1() {  
 std::vector<int> diffs = **{**1, 2, 4, 8, 16**}**;  
 std::vector<int> maxLCS;  
 std::ofstream out("test1\_results.txt");  
  
 // Вывод заголовка в консоль  
 std::cout << "Test 1: Maximum LCS Length for Different Number of Changes\n";  
 std::cout << "-------------------------------------------------------\n";  
 std::cout << "Number of Differences | Maximum LCS Length\n";  
 std::cout << "-------------------------------------------------------\n";  
  
 // Цикл по количеству отличий  
 for (int diff : diffs) {  
 int maxLen = 0;  
 // Генерация 1000 пар строк  
 for (int i = 0; i < 1000; i++) {  
 std::string s1 = generateRandomString(128);  
 std::string s2 = createDifferentString(s1, diff);  
 std::string hash1 = sha1(s1);  
 std::string hash2 = sha1(s2);  
 int lcs = longestCommonSubsequence(hash1, hash2);  
 maxLen = std::max(maxLen, lcs);  
 }  
 maxLCS.push\_back(maxLen);  
 out << diff << " " << maxLen << std::endl;  
 // Вывод результата в консоль  
 std::cout << std::setw(21) << diff << " | " << maxLen << std::endl;  
 }  
 out.close();  
 std::cout << "-------------------------------------------------------\n";  
}  
  
// Тест 2: Поиск коллизий для разного количества хешей  
void test2() {  
 std::vector<int> Ns = **{**100, 1000, 10000, 100000, 1000000**}**;  
 std::ofstream out("test2\_results.txt");  
  
 // Вывод заголовка в консоль  
 std::cout << "\nTest 2: Collision Detection\n";  
 std::cout << "-------------------------------------\n";  
 std::cout << "Number of Hashes | Collisions\n";  
 std::cout << "-------------------------------------\n";  
  
 // Цикл по количеству хешей  
 for (int N : Ns) {  
 std::set<std::string> hashes;  
 int collisions = 0;  
 // Генерация хешей  
 for (int i = 0; i < N; i++) {  
 std::string s = generateRandomString(256);  
 std::string hash = sha1(s);  
 if (hashes.find(hash) != hashes.end()) {  
 collisions++;  
 } else {  
 hashes.insert(hash);  
 }  
 }  
 std::string collisionResult = collisions > 0 ? std::to\_string(collisions) : "None";  
 out << N << " " << collisionResult << std::endl;  
 // Вывод результата в консоль  
 std::cout << std::setw(16) << N << " | " << collisionResult << std::endl;  
 }  
 out.close();  
 std::cout << "-------------------------------------\n";  
}  
  
// Тест 3: Измерение времени вычисления хеша  
void test3() {  
 std::vector<int> lengths = **{**64, 128, 256, 512, 1024, 2048, 4096, 8192**}**;  
 std::ofstream out("test3\_results.txt");  
  
 // Вывод заголовка в консоль  
 std::cout << "\nTest 3: Average Hash Computation Time\n";  
 std::cout << "---------------------------------------------\n";  
 std::cout << "Input Length | Average Time (ms)\n";  
 std::cout << "---------------------------------------------\n";  
  
 // Цикл по длинам строк  
 for (int len : lengths) {  
 double totalTime = 0.0;  
 // 1000 генераций для каждой длины  
 for (int i = 0; i < 1000; i++) {  
 std::string s = generateRandomString(len);  
 auto start = std::chrono::high\_resolution\_clock::now();  
 sha1(s);  
 auto end = std::chrono::high\_resolution\_clock::now();  
 totalTime += std::chrono::duration<double, std::milli>(end - start).count();  
 }  
 double avgTime = totalTime / 1000.0;  
 out << len << " " << avgTime << std::endl;  
 // Вывод результата в консоль  
 std::cout << std::setw(12) << len << " | " << std::fixed << std::setprecision(4) << avgTime << std::endl;  
 }  
 out.close();  
 std::cout << "---------------------------------------------\n";  
}  
  
int main() {  
 std::cout << "Starting SHA-1 Laboratory Work\n\n";  
 test1(); // Запуск теста 1  
 test2(); // Запуск теста 2  
 test3(); // Запуск теста 3  
 std::cout << "\nAll tests completed. Results saved to files.\n";  
 return 0;  
}

**Сравнительный анализ результатов**

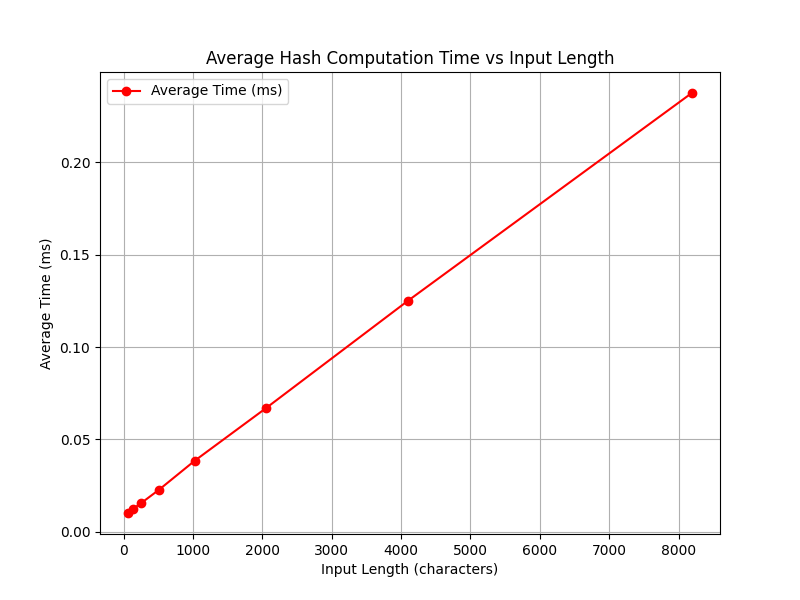
Тест 1: Сравнение хешей (LCS)

сравнить хеши для пар между собой, проведя поиск одинаковых последовательностей символов в хешах и подсчитав максимальную длину такой последовательности, по оси х кол-во отличий, а по оси y максимальная длинна одинаковой последовательности.

**График зависимости:**



* **Тест 2:** 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.
* **100 None**
* **1000 None**
* **10000 None**
* **100000 None**
* **1000000 None**
* Тест 3: 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
* **График зависимости:**



Линейная зависимость подтверждает теоретическую сложность O(L), где L — длина входного сообщения. Увеличение времени связано с ростом количества блоков (каждый блок — 512 бит), которые обрабатываются SHA-1.

# Заключение.

* SHA-1 остаётся эффективным алгоритмом для некриптографических задач, таких как контроль целостности данных, но его использование в криптографии не рекомендуется из-за уязвимостей.
* Реализация и тесты позволили глубже понять свойства хеш-функций, включая лавинный эффект и устойчивость к коллизиям.
* Для улучшения анализа сходства хешей в будущем рекомендуется использовать побитовые метрики (например, расстояние Хэмминга) вместо LCS.