Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 8

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33 **Костяева К.С.**

Москва

2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ

Хэш-функция

Хеш-функция — это специальная функция, которая:

1. Принимает на вход данные любой длины (строка, файл, сообщение и т.д.).
2. Возвращает "отпечаток" фиксированной длины — набор байт или символов, называемый хешем или хеш-значением.

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.

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 прообраза

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

SHA 1

Secure Hash Algorithm 1 — алгоритм криптографического хеширования. Для входного сообщения произвольной длины (максимум 2^64−1 бит, что примерно равно 2 эксабайта) алгоритм генерирует 160-битное (20 байт) хеш-значение, которое обычно отображается как шестнадцатеричное число длиной в 40 цифр. Используется во многих криптографических приложениях и протоколах.

SHA-1 реализует хеш-функцию, построенную на идее функции сжатия. Входами функции сжатия являются блок сообщения длиной 512 бит и выход предыдущего блока сообщения. Выход

представляет собой значение всех хеш-блоков до этого момента. Хеш-значением всего сообщения является выход последнего блока.

Инициализация

Исходное сообщение разбивается на блоки по 512 бит в каждом. Последний блок дополняется до длины, кратной 512 бит. Сначала добавляется 1 (бит), а потом — нули, чтобы длина блока стала равной 512 — 64 = 448 бит. В оставшиеся 64 бита записывается длина исходного сообщения в битах (в big-endian формате). Если последний блок имеет длину более 447, но менее 512 бит, то дополнение выполняется следующим образом: сначала добавляется 1 (бит), затем — нули вплоть до конца 512-битного блока; после этого создается ещё один 512-битный блок, который заполняется вплоть до 448 бит нулями, после чего в оставшиеся 64 бита записывается длина исходного сообщения в битах (в big-endian формате). Дополнение последнего блока осуществляется всегда, даже если сообщение уже имеет нужную длину.

ЗАДАНИЕ

Вариант 2

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

1. SHA1

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

ПРАКТИЧЕСКАЯ ЧАСТЬ

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <unordered_set>
#include <random>
#include <chrono>
#include <sstream>
#include <iomanip>
#include <algorithm>
#include <numeric>
#include <cstdint>
#include <cstring>
```

```
using namespace std;
using namespace std::chrono;

// Целочисленная степень

size_t intPow(size_t base, int exp) {
	size_t result = 1;
	for (int i = 0; i < exp; ++i) {
		result *= base;
	}
	return result;
}

// SHA1 реализация

class SHA1 {
public:
	static string hash(const string& input) {  // получает строку и возвращает её SHA-1 хеш
		// Инициализация 5 переменных (начальные значения)
		uint32_t h0 = 0x67452301;
		uint32_t h1 = 0xEFCDAB89;
		uint32_t h2 = 0x98BADCFE;
		uint32_t h3 = 0x10325476;
		uint32_t h4 = 0xC3D2E1F0;
		vector<uint8_t> data(input.begin(), input.end()); // строку превращаем в массив байтов
		uint64_t originalLength = data.size() * 8;

		//Добавление "хвоста" — паддинг
		data.push_back(0x80); // Добавляем 1 и кучу нулей...
		while ((data.size() + 8) % 64 != 0) {
			data.push_back(0x00);
		}

		// Добавление в конец данных о длине исходной строки
		for (int i = 7; i >= 0; --i) {
			data.push_back((originalLength >> (i * 8)) & 0xFF);
		}

		auto rotl = [](uint32_t v, int s) {
		return (v << s) | (v >> (32 - s));
		};

		// Разбивка на блоки по 64 байта
		for (size_t i = 0; i < data.size(); i += 64) {
		uint32_t w[80] = { 0 }; // Подготовка 80 чисел (w[80])
		for (int j = 0; j < 16; ++j) {
			w[j] = (data[i + j * 4] << 24) |
				(data[i + j * 4 + 1] << 16) |
				(data[i + j * 4 + 2] << 8) |
				(data[i + j * 4 + 3]);
		}
		for (int j = 16; j < 80; ++j) {
			w[j] = rotl(w[j - 3] ^ w[j - 8] ^ w[j - 14] ^ w[j - 16], 1);
		}
		// здесь переменные a, b, c, d, e обновляются 80 раз, комбинируя входные данные,
специальные функции f, и константы k(перемешивание)
		uint32_t a = h0, b = h1, c = h2, d = h3, e = h4;
```

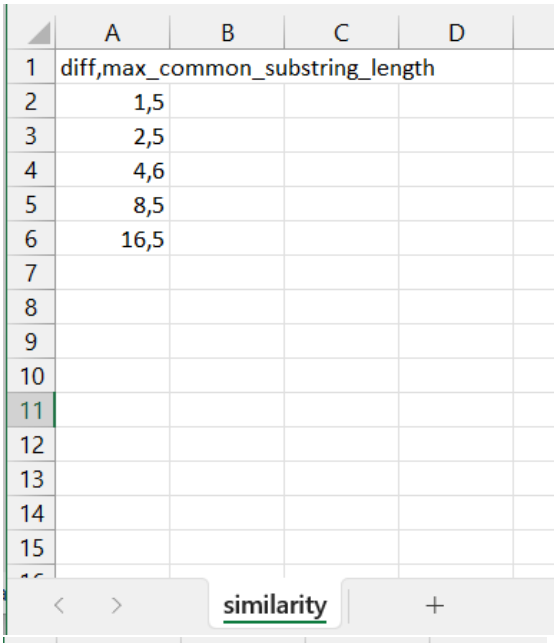
```
		for (int j = 0; j < 80; ++j) {
			uint32_t f, k;
				if (j < 20) {
					f = (b & c) | ((~b) & d);
					k = 0x5A827999;
				}
				else if (j < 40) {
					f = b ^ c ^ d;
					k = 0x6ED9EBA1;
				}
				else if (j < 60) {
					f = (b & c) | (b & d) | (c & d);
					k = 0x8F1BBCDC;
				}
				else {
					f = b ^ c ^ d;
					k = 0xCA62C1D6;
				}
				uint32_t temp = rotl(a, 5) + f + e + k + w[j];
				e = d;
				d = c;
				c = rotl(b, 30);
				b = a;
				a = temp;
			}
			// Добавляем результат к общему хешу
			h0 += a;
			h1 += b;
			h2 += c;
			h3 += d;
			h4 += e;
			}

			stringstream ss; // Преобразуем h0–h4 в шестнадцатеричную строку
			ss << hex << setfill('0')
			<< setw(8) << h0
			<< setw(8) << h1
			<< setw(8) << h2
			<< setw(8) << h3
			<< setw(8) << h4;
			return ss.str();
		}

};

// Генерация случайной строки(делает случайную строку)

string randomString(size_t length) {
	static const char charset[] =
		"abcdefghijklmnopqrstuvwxyz"
		"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
		"0123456789";
	static random_device rd;
	static mt19937 gen(rd());
	uniform_int_distribution<> dist(0, sizeof(charset) - 2);
	string s;
	s.reserve(length);
```

```
	for (size_t i = 0; i < length; ++i) {
		s += charset[dist(gen)];
	}
	return s;
}

// Модификация diff символов(меняет diff символов в строке)

string modifyString(const string& str, int diff) {
	string modified = str;
	static random_device rd;
	static mt19937 gen(rd());
	uniform_int_distribution<> posDist(0, int(str.size() - 1));
	uniform_int_distribution<> charDist(32, 126);unordered_set<int> used;
	while (int(used.size()) < diff) {
		int pos = posDist(gen);
		if (used.count(pos)) continue;
		char orig = modified[pos];
		char c;
		do {
		c = static_cast<char>(charDist(gen));
		} while (c == orig);
		modified[pos] = c;
		used.insert(pos);
	}
	return modified;
}

// Поиск максимальной общей подстроки(ищет, сколько символов подряд совпадает у двух строк.)

int maxCommonSubstring(const string& a, const string& b) {
	int maxLen = 0;
	for (size_t i = 0; i < a.size(); ++i) {
		for (size_t j = 0; j < b.size(); ++j) {
			int l = 0;
			while (i + l < a.size() && j + l < b.size() && a[i + l] == b[j + l]) {
				++l;
			} maxLen = max(maxLen, l);
		}
	}
	return maxLen;
}
	// Тест 1: схожесть хешей(показать, как сильно отличается хеш, если поменять несколько
символов во входной строке)

//В цикле по числу изменений diff (например, 1, 2, 4, 8, 16):
//Генерируем строку base и меняем в ней diff символов → получается mod.
//Получаем два хеша: h1 = SHA1::hash(base), h2 = SHA1::hash(mod).
//Сравниваем, сколько символов подряд совпадают в этих хешах (maxCommonSubstring).
//Запоминаем наибольшее значение maxLen для каждого diff.
void testSimilarity() {
	ofstream file("similarity.csv");
	file << "diff,max_common_substring_length\n";
	vector<int> diffs = { 1, 2, 4, 8, 16 };
	for (int d : diffs) {
		int maxLen = 0;
		for (int i = 0; i < 10000; ++i) {
			string base = randomString(128);
```

```
                string mod = modifyString(base, d);
                string h1 = SHA1::hash(base);
                string h2 = SHA1::hash(mod);
                maxLen = max(maxLen, maxCommonSubstring(h1, h2));
            }
            file << d << "," << maxLen << "\n";
        }
}

// Тест 2: коллизии(проверить, сколько раз разные строки случайно дают одинаковый хеш)

//В цикле по числу строк N = 10^2, 10^3, ..., 10^6:
//Генерируем N случайных строк.
//Для каждой строки считаем хеш.
//Сохраняем хеши в unordered_set<string> seen — чтобы отслеживать уникальность.
//Если хеш уже есть в сете — увеличиваем счётчик collisions.
void testCollisions() {
        ofstream file("collisions.csv");
        file << "N,collisions\n";
        for (int exp = 2; exp <= 6; ++exp) {
            size_t N = intPow(10, exp);
            unordered_set<string> seen;
            size_t collisions = 0;
            for (size_t i = 0; i < N; ++i) {
            string s = randomString(256);
            string h = SHA1::hash(s);
            if (!seen.insert(h).second) {
                ++collisions;
            }
            }
            file << N << "," << collisions << "\n";
        }

}

// Тест 3: скорость(измерить, сколько времени уходит на хеш одной строки разной длины)

//Берём разные длины строки: 64, 128, ..., 8192.
//Для каждой длины:
        //10 000 раз генерируем строку.
        //Засекаем время перед и после вызова SHA1::hash(str).
        //Считаем, сколько микросекунд прошло.
//Считаем среднее время на одну строку (avg_time_us).
void testSpeed() {
        ofstream file("speed.csv");
        file << "length,avg_time_us\n";
        vector<int> sizes = { 64, 128, 256, 512, 1024, 2048, 4096, 8192 };
        for (int sz : sizes) {
            vector<int64_t> times;
            times.reserve(10000);
            for (int i = 0; i < 10000; ++i) {
            string s = randomString(sz);
            auto start = high_resolution_clock::now();
            SHA1::hash(s);
            auto end = high_resolution_clock::now();
            times.push_back(duration_cast<microseconds>(end - start).count());
            }
            int64_t total = accumulate(times.begin(), times.end(), int64_t(0));
```

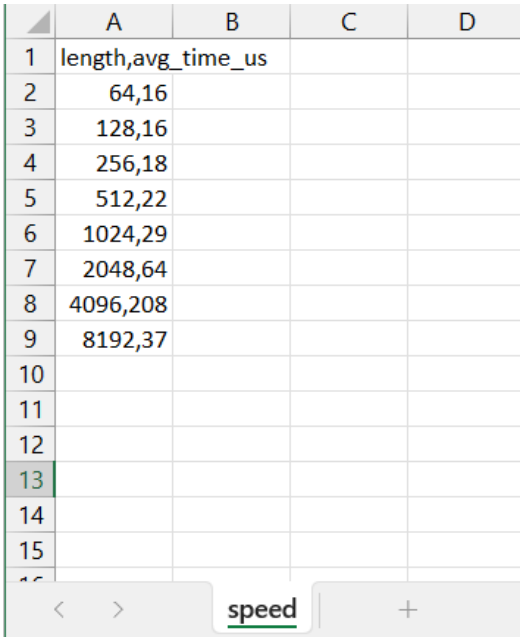
```
            int64_t avg = total / times.size();
            file << sz << "," << avg << "\n";
        }

}

int main() {
        testSimilarity();
        testCollisions();
        testSpeed();
        cout << "Tests done, CSV files generated.\n";
        return 0;
}
```

Результат работы программы:



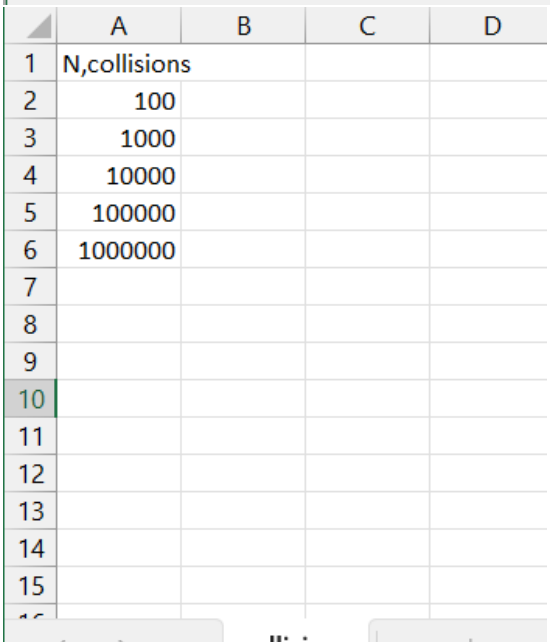
	A	B	C	D
1	diff,max_common_substring_length			
2	1,5			
3	2,5			
4	4,6			
5	8,5			
6	16,5			
7				
8				
9				
10				
11				
12				
13				
14				
15				

similarity



	A	B	C	D
1	length,avg_time_us			
2	64,16			
3	128,16			
4	256,18			
5	512,22			
6	1024,29			
7	2048,64			
8	4096,208			
9	8192,37			
10				
11				
12				
13				
14				
15				

speed



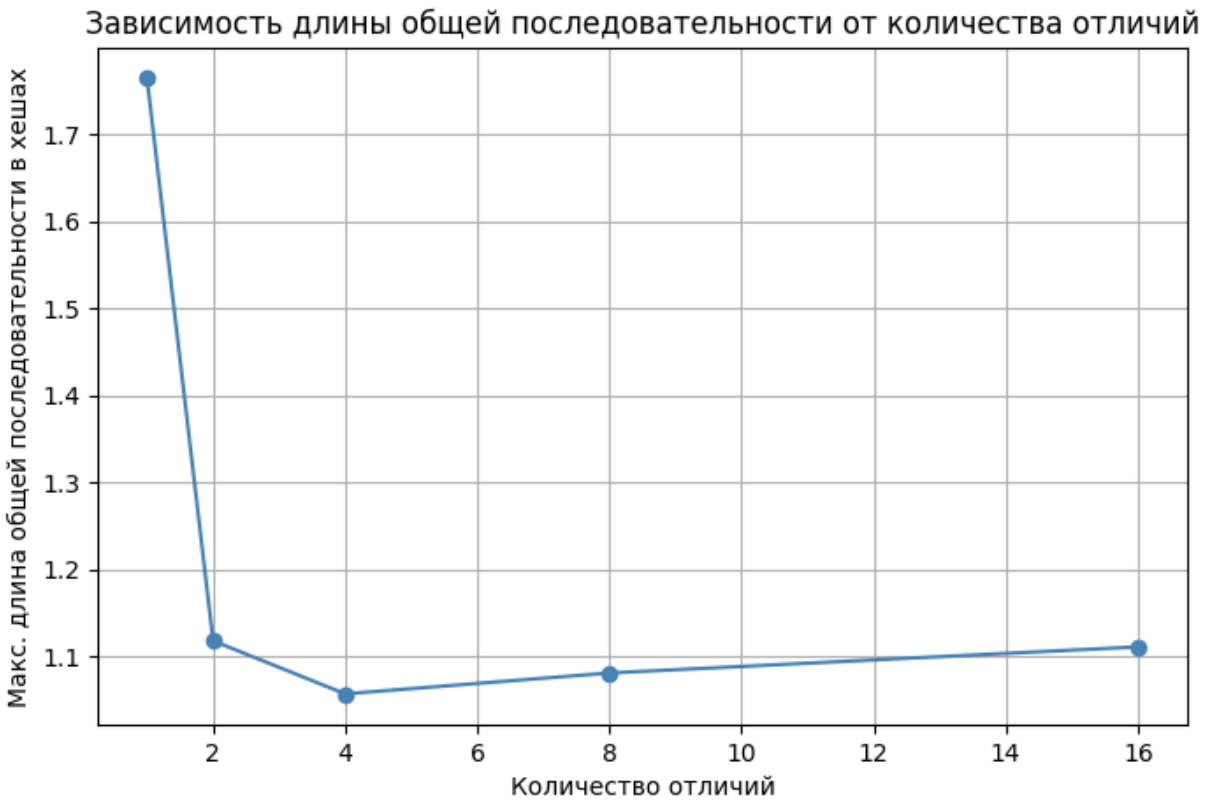
	A	B	C	D
1	N,collisions			
2	100			
3	1000			
4	10000			
5	100000			
6	1000000			
7				
8				
9				
10				
11				
12				
13				
14				
15				

d8f4590320e1343a915b6394170650a8f35d6926

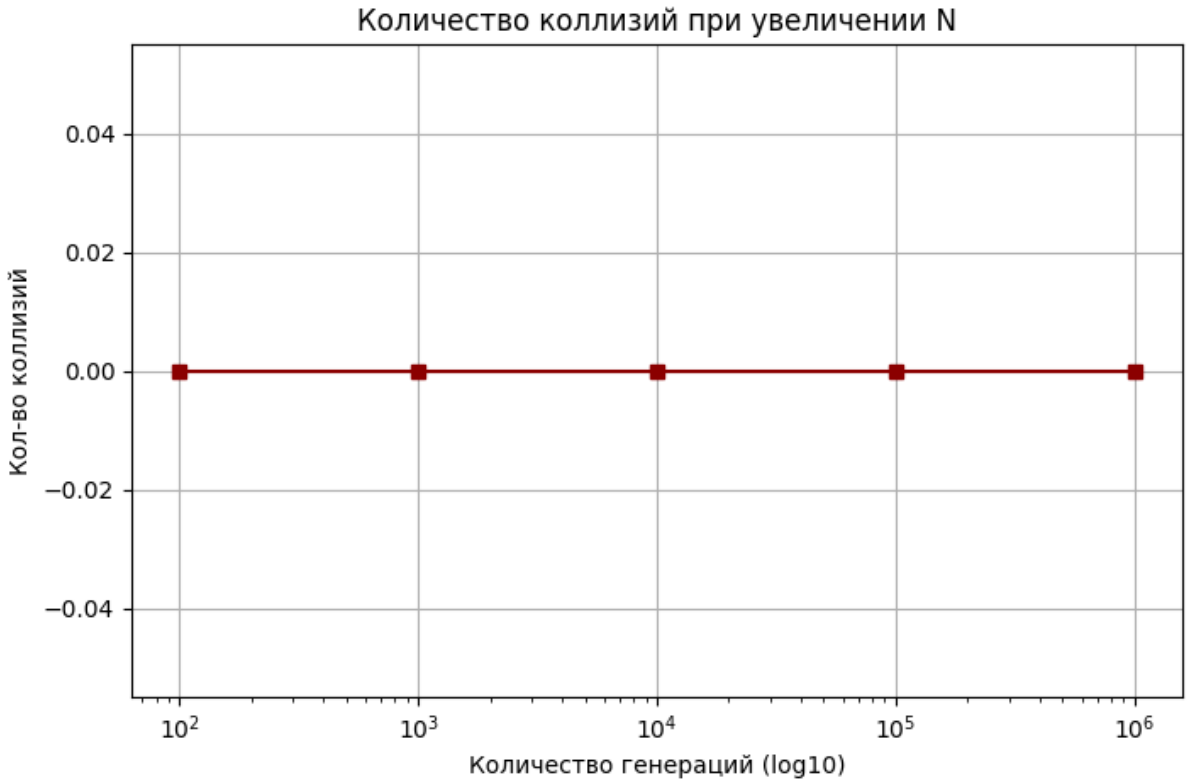
ba79baeb9f10896a46ae74715271b7f586e74640

Графики:

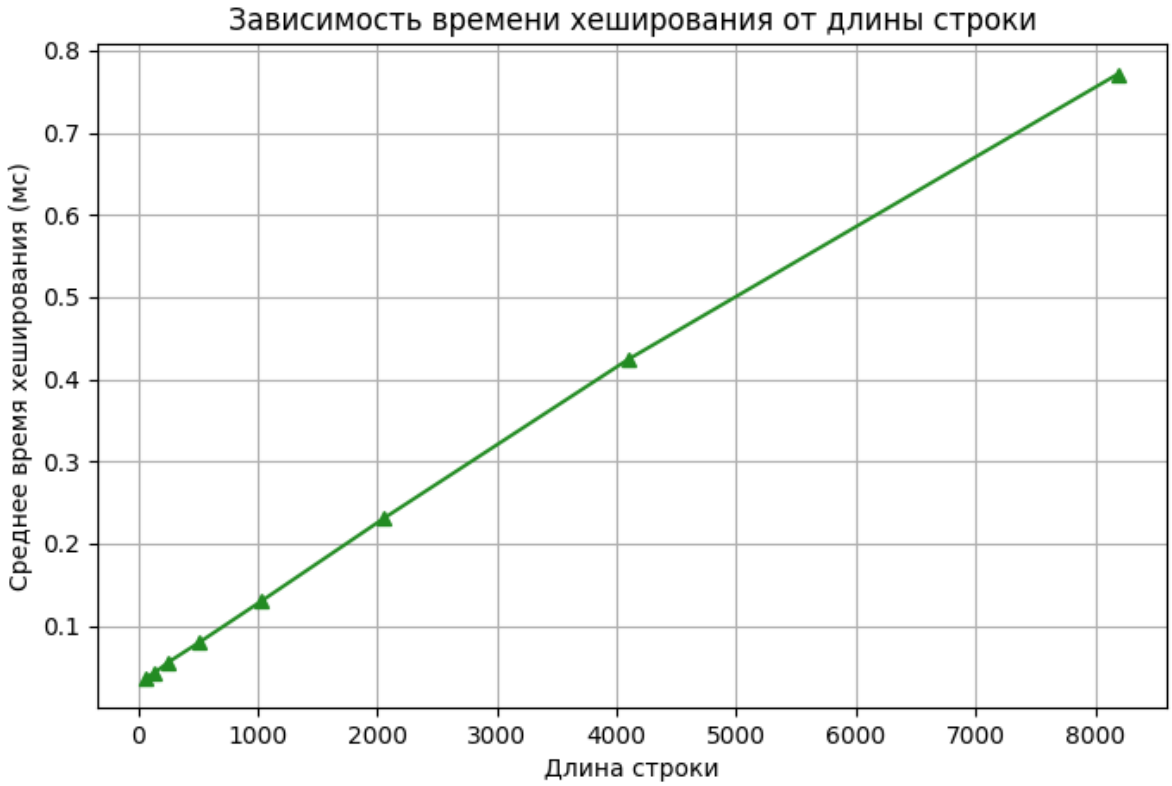
Зависимость длины совпадающей подпоследовательности от количества отличий



Количество коллизий при увеличении числа генераций



Время хеширования в зависимости от длины строки



ВЫВОД

Скорость выполнения хеширования показывает линейную зависимость от размера входной строки.

В ходе генерации хешей для случайных строк длиной 256 символов при количестве попыток $N = 10^2, 10^3, 10^4, 10^5$, и 10^6 ни одной коллизии зафиксировано не было.

При увеличении количества различий между двумя входными строками (на 1, 2, 4, 8 и 16 символов) наблюдается сокращение длины совпадающих префиксов в их хешах.

Это говорит о выраженном лавинном эффекте: даже минимальные изменения на входе приводят к значительным изменениям результата. (Примеры отражены в последних строках вывода программы.)