

**Санкт-Петербургский Национальный Исследовательский
Университет Информационных технологий, механики и оптики**

Отчет

**Дисциплина: Объектно-ориентированное
программирование.**

Лабораторная работа 1. Алгоритмы хеширования.

Выполнил: Влазнев Д.В.

Группа № K3223

Проверил: доцент Иванов С.Е.

Санкт-Петербург

2024

Цель работы:

Изучить алгоритмы хеширования, реализовать алгоритм Рабина-Карпа поиска подстроки в строке с применением хеширования.

Задачи:

1. Изучить алгоритм Рабина-Карпа.
2. Выполнить задания для лабораторной работы.

Задание для лабораторной работы:

1. Реализовать поиск одинаковых строк. Дан список строк $S[1..n]$, каждая длиной не более m символов. Требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.
2. Реализовать алгоритм Рабина-Карпа поиска подстроки в строке за $O(n)$.

Содержание

Содержание.....	3
Задание 1	4
Задание 2	5
Контрольные вопросы	8
Выводы.....	10

Задание 1

Для того, чтобы найти одинаковые строки, необходимо найти хэш каждой строки и сравнить их. Для этого напомним функцию вычисления полиномиального хэша (См. Рисунок 1).

```
public static int getHash(string str, int stringBase = 256, int modulus = 101)
{
    int hash = 0;
    for (int i = 0; i < str.Length; i++)
    {
        hash += str[i] * (int)Math.Pow(stringBase, str.Length - i - 1) % modulus;
    }
    return hash;
}
```

Рисунок 1 – Код программы

Таким образом, считывая строки и вычисляя их хэш, группируем их по хэшу (См. Рисунок 2–3).

```
static void Main(string[] args)
{
    int n = int.Parse(Console.ReadLine());
    string[] stringArray = new string[n];

    for (int i = 0; i < n; i++)
    {
        stringArray[i] = Console.ReadLine();
    }

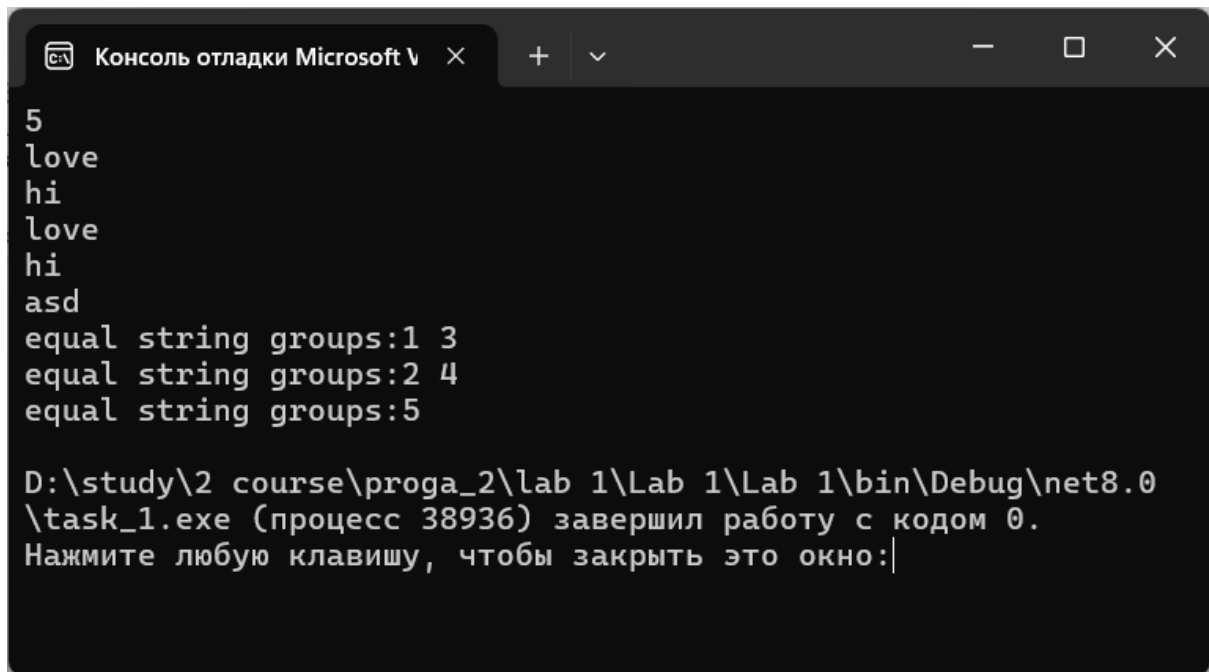
    Dictionary<int, int> hashes = new Dictionary<int, int>();

    for (int i = 0; i < n; i++)
    {
        hashes.Add(i, getHash(stringArray[i]));
    }

    var groups = hashes.GroupBy(x => x.Value);

    foreach (var group in groups)
    {
        Console.WriteLine("equal string groups:" + string.Join(" ", group.Select(x
=> x.Key + 1)));
    }
}
```

Рисунок 2 – Код программы

A screenshot of a Microsoft Visual Studio debug console window. The window has a title bar with the text 'Консоль отладки Microsoft V' and standard window controls. The console output shows the following text: '5', 'love', 'hi', 'love', 'hi', 'asd', 'equal string groups:1 3', 'equal string groups:2 4', 'equal string groups:5', and a final message: 'D:\study\2 course\proga_2\lab 1\Lab 1\Lab 1\bin\Debug\net8.0\task_1.exe (процесс 38936) завершил работу с кодом 0. Нажмите любую клавишу, чтобы закрыть это окно:'.

```
5
love
hi
love
hi
asd
equal string groups:1 3
equal string groups:2 4
equal string groups:5

D:\study\2 course\proga_2\lab 1\Lab 1\Lab 1\bin\Debug\net8.0
\task_1.exe (процесс 38936) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:
```

Рисунок 3 – Пример вывода программы

Задание 2

Для реализации алгоритма Рабина-Карпа для начала нужно также задать функцию хэша. Также для оптимизации была разработана функция возведения в степень с постоянным делением по модулю, чтобы не произошел оверфлоу (См. Рисунок 4).

```
int GetHashCode(string str)
{
    int hash = str[0] % _modulus;

    for (int i = 1; i < str.Length; i++)
    {
        hash *= _baseNumber;
        hash += str[i];
        hash %= _modulus;
    }

    return hash;
}

int PowMod(int x, int y)
{
    int result = x;
    for (int i = 2; i <= y; i++)
    {
        result = result * x % _modulus;
    }
    return result;
}
```

Рисунок 4 – Код программы

При создании класса необходимо указать базовое число, которое будет возводиться в степень, по умолчанию 256, и простое число для модуляции, по умолчанию, 3673 (См. Рисунок 5).

```
private readonly int _baseNumber;
private readonly int _modulus;

public RabinKarp(int baseNumber = 256, int modulus = 3673)
{
    _baseNumber = baseNumber;
    _modulus = modulus;
}
```

Рисунок 5 – Код программы

Основная функция находит все вхождения паттерна в тексте и возвращает список индексов. Для улучшения производительности используется класс Span для сравнения строк и кольцевой кэш (См. Рисунок 6–7).

```
public List<int> FindOccurences(string pattern, string text)
{
    var patternSpan = pattern.AsSpan();
    var textSpan = text.AsSpan();

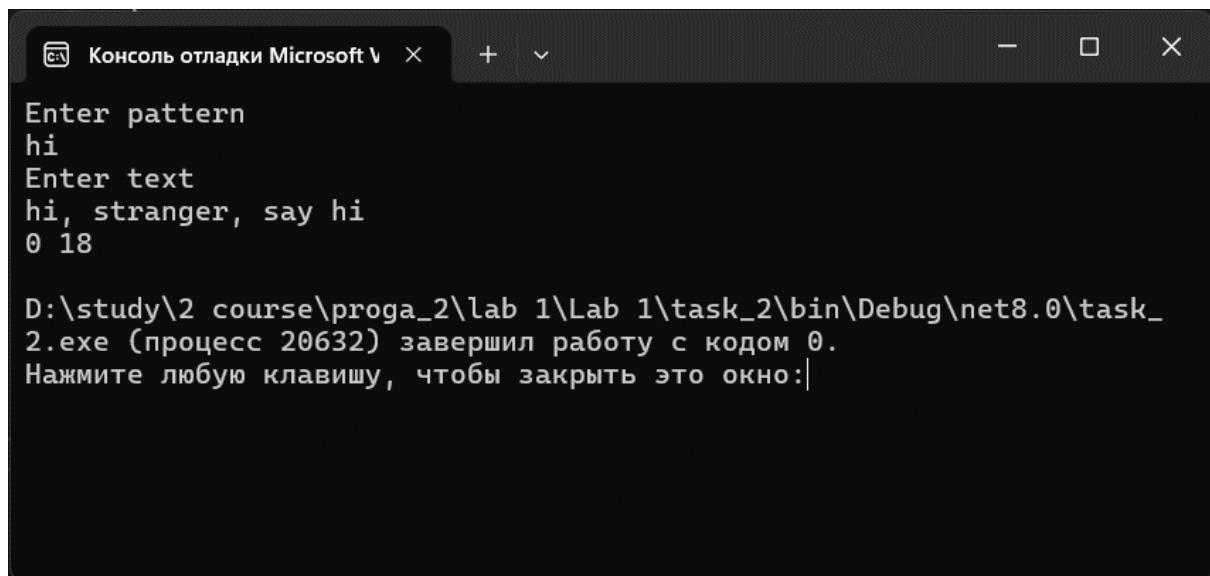
    int patternHash = GetHash(pattern);
    int textHash = GetHash(text.Substring(0, pattern.Length));
    int baseHash = PowMod(_baseNumber, pattern.Length - 1);
    List<int> result = new List<int>();

    for (int i = 0; i < text.Length - pattern.Length; i++)
    {
        if (textHash == patternHash && patternSpan.SequenceEqual(textSpan.Slice(i,
pattern.Length)))
        {
            result.Add(i);
        }

        textHash -= (text[i] * baseHash) % _modulus;
        textHash += _modulus;
        textHash *= _baseNumber;
        textHash += text[i + pattern.Length];
        textHash %= _modulus;
    }
    if (textHash == patternHash)
    {
        result.Add(text.Length - pattern.Length);
    }

    return result;
}
```

Рисунок 6 – Код программы

The image shows a screenshot of the Microsoft Visual Studio console window. The title bar at the top reads "Консоль отладки Microsoft V" with standard window controls (minimize, maximize, close) and a tab management icon. The console output is as follows:
Enter pattern
hi
Enter text
hi, stranger, say hi
0 18

D:\study\2 course\proga_2\lab 1\Lab 1\task_2\bin\Debug\net8.0\task_2.exe (процесс 20632) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно:|

Рисунок 7 – Пример вывода программы

Контрольные вопросы

1. Алгоритмы хеширования используются для решения различных задач, включая:

- Уникальная идентификация данных или объектов.
- Быстрый поиск и доступ к данным по ключу.
- Хранение пар "ключ-значение" в хеш-таблицах.
- Цифровая подпись и проверка целостности данных.

2. Проблема коллизий возникает, когда два различных входных данных дают одинаковый хеш-код. Для решения этой проблемы существуют различные методы:

- Метод цепочек (открытое хеширование): при коллизии элементы с одинаковым хешем добавляются в связанный список или другую структуру данных.
- Линейное пробирование: при коллизии происходит поиск следующего свободного слота в хеш-таблице.
- Двойное хеширование: используется вторая хеш-функция для определения нового индекса при коллизии.

3. Ограничения и условия на применение алгоритма хеширования включают:

- Хеш-функция должна быть быстрой и равномерно распределенной.
- Коллизии должны быть эффективно обрабатываемыми.
- Размер хеш-таблицы должен быть достаточно большим для уменьшения коллизий.

4. Для реализации алгоритма хеширования в C# часто используются следующие конструкции ООП:

- Классы для представления хеш-таблиц и элементов данных.
- Методы для вычисления хеш-кода объектов.

5. Сложность алгоритма хеширования оценивается по времени выполнения операций вставки, поиска и удаления элементов в хеш-таблице. Обычно она выражается как $O(1)$ для успешных операций при равномерном распределении данных.

6. Точность алгоритма хеширования оценивается по уровню коллизий и равномерности распределения хеш-кодов. Чем меньше коллизий и лучше распределены значения хеш-кодов, тем выше точность алгоритма.

Выводы

В ходе выполнения работы изучены и реализованы средствами ООП С# алгоритмы хеширования.

Ссылка на код: <https://github.com/geherious/OOP-2/tree/main/lab%201>