Лабораторная работа № 4 по курсу дискретного анализа: строковые алгоритмы

Выполнил студент группы М80-208Б-22 МАИ *Цирулев Николай*.

Условие

Необходимо реализовать поиск одного образца в тексте с использованием алгоритма Z-блоков. Алфавит — строчные латинские буквы.

- Формат ввода:

  На первой строке входного файла текст, на следующей — образец. Образец и текст помещаются в оперативной памяти.

- Формат вывода:

  В выходной файл нужно вывести информацию о всех позициях текста, начиная с которых встретились вхождения образца. Выводить следует по одной позиции на строчке, нумерация позиций в тексте начинается с 0.

Метод решения

Для поиска образца в тексте необходимо было реализовать алгоритм z-блоков. Сначала строка образца и текста объединяются, и для каждой позиции вычисляется z-функция, описывающая длину наибольшего префикса строки, совпадающего с подстрокой, начинающейся в данной позиции. После этого, анализируя z-функцию строки, программа находит индексы в тексте, где образец встретился, выводя их на экран.

Описание алгоритма z-блоков: Будем идти слева направо и хранить z-блок — самую правую подстроку, равную префиксу, которую мы успели обнаружить. Будем обозначать его границы как l и r включительно.

Пусть мы сейчас хотим найти z_i, а все предыдущие уже нашли. Новый i-й символ может лежать либо правее z-блока, либо внутри него:

- Если правее, то мы просто наивно перебором найдем z_i (максимальный отрезок, начинающийся с s_i и равный префиксу), и объявим его новым z-блоком.

- Если i-й элемент лежит внутри z-блока, то мы можем посмотреть на значение z_{i-l} и использовать его, чтобы инициализировать z_i чем-то, возможно, отличным от нуля. Если z_{i-l} левее правой границы z-блока, то $z_i = z_{i-l}$ — больше z_i быть не может. Если он упирается в границу, то «обрежем» его до неё и будем увеличивать на единицу.

Описание программы

Исходный код программы содержится в файле main.cpp. Для хранения входных данных (образца и текста) используется контейнер стандартной библиотеки C++ `std::string`. Для хранения z-функции строки был выбран `std::vector`.

Первая строчка `main` отключает синхронизацию потоков ввода-вывода. Вторая отвязывает стандартный поток ввода от стандартного потока вывода, блягодаря чему при каждом вызове `std::cin` не сбрасывается буфер. Обе эти строчки позволяют значительно ускорить ввод-вывод в программе.

Далее программа считывает строки и объединяет их в одну. `z_algorithm()` вычисляет z-функцию используя алгоритм z-блоков. Далее остается пройтись по полученной z-функции и вывести индексы элементов, значение которых больше или равно длине образца.

```
#include <iostream>
#include <vector>

std::vector<long long> z_algorithm(std::string s) {
    long long n = (long long) s.size();
    std::vector<long long> z(n, 0);
    long long l = 0, r = 0;
    for (long long i = 1; i < n; i++) {
        if (i <= r) {
            z[i] = std::min(z[i - l], r - i + 1);
        }

        while (i + z[i] < n && s[i + z[i]] == s[z[i]]) {
            ++z[i];
        }

        if (i + z[i] - 1 > r) {
            r = i + z[i] - 1;
            l = i;
        }
    }
    return z;
}

int main() {
    std::ios_base::sync_with_stdio(false);
        std::cin.tie(0);
    std::string s;
    std::string p;
    std::cin >> s;
```

```
    std::cin >> p;
    std::vector<long long> z = z_algorithm(p + s);
    long long n = p.size();
    for (long long i = 1; i < z.size(); ++i) {
        if (z[i] >= n && i - n >= 0) {
            std::cout << i - n << '\n';
        }
    }
}
```

Дневник отладки

После отправки решения в чекер была обнаружена ошибка. В ходе ручного тестирования была выяснена причина ошибки: при выводе результата не было проверки на выход индекса за пределы массива входных данных:

- Строка программы с ошибкой:

```
if (z[i] >= n) {
```

- Исправленная строка:

```
if (z[i] >= n && i - n >= 0) {
```

Тест производительности

В алгоритме мы делаем столько же действий, сколько раз сдвигается правая граница z-блока, поэтому асимптотика алгоритма - *O(n)*.

Для лучшей наглядности приведём таблицу, в которой написанный алгоритм сравнивается с двумя встроенными алгоритмами поиска строк, доступными в стандартной библиотеке языка C++ (std::search()).

Количество символов в тексте	z_algorithm(), мс	std::default_searcher, мс	std::boyer_moore_searcher, мс
100000	3240	1858	5892
1000000	26573	17369	58879
10000000	268407	170710	587061
100000000	2797324	1712765	5883362

Как показано в таблице, время выполнения алгоритма демонстрирует практически линейную зависимость от количества символов в тексте. Это соответствует ожидаемой сложности нашего алгоритма. Однако, проведенное сравнение показывает, что алгоритмы, реализованные в std::search(), значительно превосходят по скорости написанный в данной работе алгоритм, что говорит о несовершенстве этого алгоритма, даже принимая во внимание эффективный расчёт z-функции.

Ниже приведена программа benchmark.cpp, использовавшаяся для определения времени работы функций:

```
#include <iostream>
#include <random>
#include <string>
#include <algorithm>
#include <chrono>
#include <fstream>
#include <cassert>

#include "main.cpp"

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution int_dist(100, 1000);
    std::uniform_int_distribution short_dist(1, 10);
    std::uniform_int_distribution<> char_dist(0, 25);
    for (size_t k = 100000; k <= 100000000; k *= 10) {
        std::string s;
        std::string p;
        int n = int_dist(gen);
        for (size_t i = 0; i < n; ++i) {
            p += static_cast<char>('a' + char_dist(gen));
        }
        for (size_t i = 0; i < k; ++i) {
            int flag = short_dist(gen);
            if (flag == 10) {
                s += p;
                i += n;
            }
            s += static_cast<char>('a' + char_dist(gen));
        }
        std::cout << k << " & ";

        auto start1 = std::chrono::high_resolution_clock::now();
        std::vector<size_t> res1;

        std::vector<long long> z = z_algorithm(p + s);
        for (long long i = 1; i < z.size(); ++i) {
            if (z[i] >= n && i - n >= 0) {
```

```
            res1.push_back(i - n);
        }
    }
    auto finish1 = std::chrono::high_resolution_clock::now();
    auto duration1 = std::chrono::duration_cast<std::chrono::microsecon
    std::cout << duration1.count() << "␣&␣";

    auto start2 = std::chrono::high_resolution_clock::now();
    std::vector<size_t> res2;
    auto it1 = s.begin();
    while (it1 != s.end()) {
        const std::default_searcher searcher(p.begin(), p.end());
        it1 = std::search(it1, s.end(), searcher);
        if (it1 != s.end()) {
            res2.push_back(it1 - s.begin());
            it1 += n;
        }
    }
    auto finish2 = std::chrono::high_resolution_clock::now();
    auto duration2 = std::chrono::duration_cast<std::chrono::microsecon
    std::cout << duration2.count() << "␣&␣";

    auto start3 = std::chrono::high_resolution_clock::now();
    std::vector<size_t> res3;
    auto it2 = s.begin();
    while (it2 != s.end()) {
        const std::boyer_moore_searcher searcher(p.begin(), p.end());
        it2 = std::search(it2, s.end(), searcher);
        if (it2 != s.end()) {
            res3.push_back(it2 - s.begin());
            it2 += n;
        }
    }
    auto finish3 = std::chrono::high_resolution_clock::now();
    auto duration3 = std::chrono::duration_cast<std::chrono::microsecon
    std::cout << duration3.count() << "␣\\" << "\\";

    assert(res1 == res2);
    assert(res1 == res3);

    std::cout << '\n';
```

```
    }
    return 0;
}
```

Выводы

В ходе выполнения данной работы были изучены строковые алгоритмы, в частности алгоритм z-блоков. На практике алгоритм z-блоков может применяться для решения различных задач анализа данных. Однако для реальных задач существуют более быстрые алгоритмы, которые превосходят z-блоки по производительности. Алгоритм, использованный в работе, носит образовательный характер и помогает глубже понять принципы работы строковых алгоритмов и обработку префиксов строк.