Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу

«Дискретный анализ»

Тема работы

“Строковые алгоритмы”

Студент : Ю.И. Катаев
Группа : М8О-210Б-21
Преподаватель : Н.К. Макаров
Оценка : _____
Дата : _____
Подпись : _____

Москва, 2023

1. Постановка задачи

Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск одного образца основанный на построении Z-блоков.
Вариант алфавита: Числа в диапазоне от 0 до 2^{32} - 1.

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

2. Описание

Z-функция от строки S и позиции x — это длина максимального префикса подстроки, начинающейся с позиции x в строке S, который одновременно является и префиксом всей строки S.

Значение Z-функции от первой позиции не определено, поэтому его обычно приравнивают к нулю или к длине строки.

Чтобы получить эффективный алгоритм, будем вычислять значения z[i] по очереди — от i = 1 до n - 1, и при этом постараемся при вычислении очередного значения z[i] максимально использовать уже вычисленные значения.

Назовем для краткости подстроку, совпадающую с префиксом строки s, отрезком совпадения. Например, значение искомой Z-функции z[i] — это длиннейший отрезок совпадения, начинающийся в позиции i (и заканчиваться он будет в позиции i + z[i] - 1). Для этого будем поддерживать координаты [l; r] самого правого отрезка совпадения, т.е. из всех обнаруженных отрезков будем хранить тот, который оканчивается правее всего. В некотором смысле, индекс r — это такая граница, до которой наша строка уже была просканирована алгоритмом, а всё остальное — пока ещё не известно.
Тогда если текущий индекс, для которого мы хотим посчитать очередное значение Z-функции, — это i, мы имеем один из двух вариантов:

i > r — т.е. текущая позиция лежит за пределами того, что мы уже успели обработать.

Тогда будем искать z[i] тривиальным алгоритмом, т.е. просто пробуя значения z[i] = 0, z[i] = 1, и т.д. Заметим, что в итоге, если z[i] окажется > 0, то мы будем обязаны обновить

координаты самого правого отрезка [l; r] — т.к. i + z[i] - 1 гарантированно окажется больше r.

i ≤ r — т.е. текущая позиция лежит внутри отрезка совпадения [l; r].

Тогда мы можем использовать уже подсчитанные предыдущие значения Z-функции, чтобы проинициализировать значение z[i] не нулём, а каким-то возможно большим числом.
Для этого заметим, что подстроки s[l … r] и s[0 … r – l] совпадают. Это означает, что в качестве начального приближения для z[i] можно взять соответствующее ему значение из отрезка s[0 … r – l] , а именно, значение z[i - l].
Однако значение z[i – l] могло оказаться слишком большим: таким, что при применении его к позиции i оно "вылезет" за пределы границы r. Этого допустить нельзя, т.к. про символы правее r мы ничего не знаем, и они могут отличаться от требуемых.
В качестве начального приближения для z[i] возьмем min(r – i + 1, z[i - l]), а дальше продолжим заполнять тривиальным алгоритмом, потому что после границы r может обнаружится продолжение отрезка совпадения.

Таким образом, для поиска подстроки в строке с помощью Z-функции будем использовать строку s = pattern + $ + text, т.е. к образцу припишем текст через символ-разделитель (который не встречается нигде в самих строках).
Посчитаем для полученной строки Z-функцию. Тогда для любого i в отрезке [0; n - 1] по соответствующему значению z[i + m + 1] можно понять, входит ли образец в текст, начиная с позиции i: если это значение Z-функции равно m, то входит, иначе — нет.

3. Разбор программы

Функция	Описание	Time complexity	Space complexity
void ScanPattern(Vector& pattern)	Считывает посимвольно входной паттерн, пропуская ненужные символы.	O(K), где K - кол-во символов	O(M), где M - длина паттерна
void ScanText(Vector& text)	Считывает посимвольно входной текст, пропуская ненужные символы.	O(K), где K - кол-во символов	O(N), где N - длина текста
void Concatenation(Vector& pattern_and_text, Vector& pattern, Vector& text)	Конкатенирует паттерн и текст через разделитель.	O(N + M)	O(N + M)
uint64_t min(uint64_t a, uint64_t b)	Возвращает минимум из двух чисел	O(1)	O(1)

uint64_t* zFunction(Vector& input_string)	Возвращает массив Z - функции, построенный на основе строки “паттерн#текст”	O(N + M)	O(N + M)
void FindSubstrings(Vector& pattern_and_text, uint64_t* z_array, uint64_t len_pattern)	Находит номер строки и порядковый номер слова в строке, начиная с которого в тексте совпал паттерн.	O(N + M)	O(1)

4. Исходный код

```
#include <iostream>
#include <stdio.h>

using namespace std;

struct Data {
    int64_t value;
    int64_t string_number;
    int64_t word_number;
};

class Vector {
    Data* array_;
    uint64_t size_;
    uint64_t capacity_;

public:
    Vector(uint64_t capacity = 1) {
        capacity_ = capacity;
        size_ = 0;
        array_ = new Data[capacity_];
    }

    void AddMemory() {
        Data* temp_array = array_;
        array_ = new Data[capacity_ * 2];
        for (int i = 0; i < capacity_; ++i) {
            array_[i] = temp_array[i];
        }
        capacity_ *= 2;
        delete[] temp_array;
    }
```

```
    void PushBack(Data elem) {
        if (size_ == capacity_) {
            AddMemory();
        }
        array_[size_] = elem;
        ++size_;
    }

    Vector& operator=(const Vector& right) {
        //проверка на самоприсваивание
        if (this == &right) {
            return *this;
        }
        array_ = right.array_;
        capacity_ = right.capacity_;
        size_ = right.size_;
        return *this;
    }

    Data& operator[](uint64_t index) {
        return array_[index];
    }

    uint64_t Length() {
        return size_;
    }

    uint64_t Capacity() {
        return capacity_;
    }

    ~Vector() {
        delete[] array_;
    }

};

void ScanPattern(Vector& pattern) {
    int64_t number;
    bool flag;
    char symbol = ' ';
    while(symbol != '\n') {
```

```
        number = 0;
        flag = false;
        symbol = getchar();
        while (symbol != ' ' && symbol != '\n' && symbol != '\t' && symbol !=
'\r') {
            flag = true;
            number = number * 10 + (symbol - '0');
            symbol = getchar();
        }
        if (flag) {
            Data inserted_data = {number, 0, 0};
            pattern.PushBack(inserted_data);
        }
    }
}

void ScanText(Vector& text) {
    int64_t number;
    bool flag;
    int64_t string_number = 1, word_number = 1;
    char symbol = ' ';
    while (symbol != EOF) {
        number = 0;
        flag = false;
        symbol = getchar();
        while (symbol != ' ' && symbol != '\n' && symbol != EOF && symbol !=
'\t' && symbol != '\r') {
            flag = true;
            number = number * 10 + (symbol - '0');
            symbol = getchar();
        }
        if (flag) {
            Data inserted_data = {number, string_number, word_number};
            text.PushBack(inserted_data);
            ++word_number;
        }

        if (symbol == '\n') {
            word_number = 1;
            ++string_number;
        }
    }
```

```
}

void Concatenation(Vector& pattern_and_text, Vector& pattern, Vector& text) {
    for (uint64_t i = 0; i < pattern.Length(); ++i) {
        pattern_and_text.PushBack(pattern[i]);
    }
    Data sentinel = {-1, -1, -1};
    pattern_and_text.PushBack(sentinel);
    for (uint64_t i = 0; i < text.Length(); ++i) {
        pattern_and_text.PushBack(text[i]);
    }
}

int min(uint64_t a, uint64_t b) {
    if (a > b) {
        return b;
    }
    return a;
}

uint64_t* zFunction(Vector& input_string) {
    uint64_t string_size = input_string.Length();
    uint64_t* z_array = new uint64_t[string_size];
    for (int i = 0; i < string_size; ++i) {
        z_array[i] = 0;
    }

    uint64_t left = 0, right = 0;
    for (uint64_t i = 1; i < string_size; ++i) {
        if (i <= right) {
            z_array[i] = min(z_array[i-left], z_array[right-i+1]);
        }
        while(i + z_array[i] < string_size && input_string[i +
z_array[i]].value == input_string[z_array[i]].value) {
            ++z_array[i];
        }
        if (z_array[i] + i - 1 > right) {
            left = i;
            right = z_array[i] + i-1;
        }
    }
    return z_array;
```

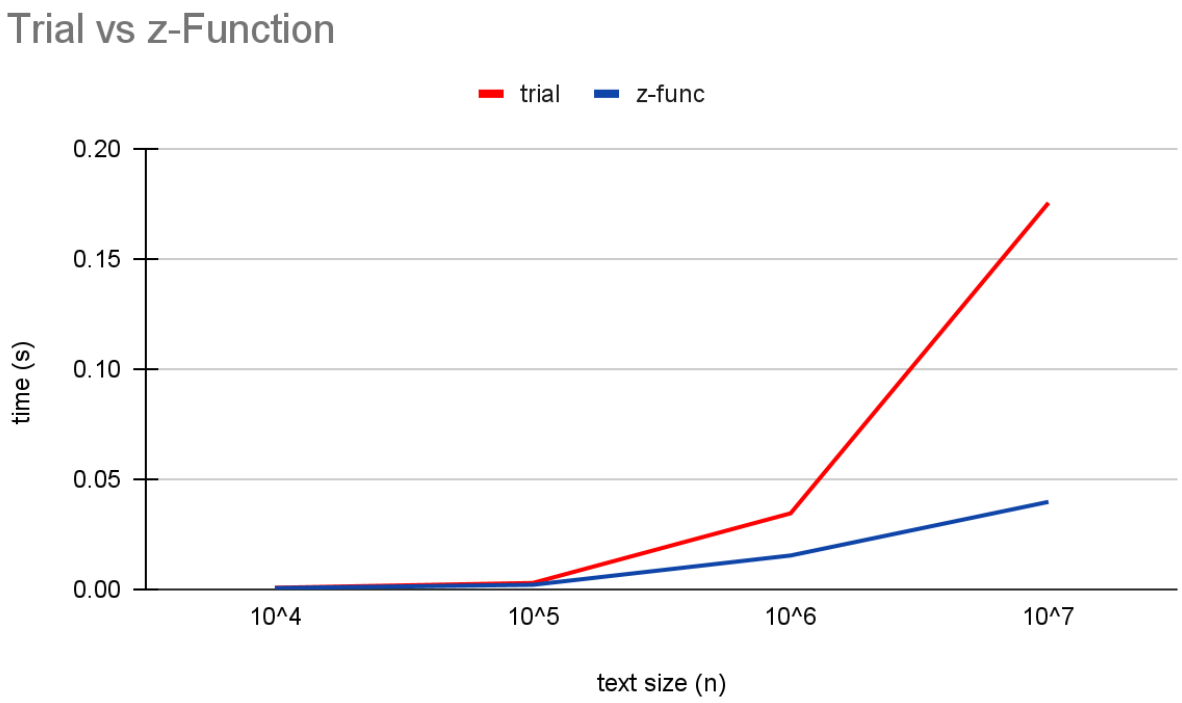
```
}

void FindSubstrings(Vector& pattern_and_text, uint64_t* z_array, uint64_t
len_pattern) {
    uint64_t index = 0;
    while (pattern_and_text[index].value != -1) {
        ++index;
    }
    for (int i = index+1; i < pattern_and_text.Length(); ++i) {
        if (z_array[i] == len_pattern) {
            cout << pattern_and_text[i].string_number << ", " <<
pattern_and_text[i].word_number << "\n";
        }
    }
}

int main() {
    Vector pattern, text;
    ScanPattern(pattern);
    ScanText(text);

    Vector pattern_and_text(pattern.Length() + text.Length() + 1);
    Concatenation(pattern_and_text, pattern, text);

    uint64_t* z_array = zFunction(pattern_and_text);
    FindSubstrings(pattern_and_text, z_array, pattern.Length());

    delete[] z_array;
    return 0;
}
```

5. Демонстрация работы программы

```
➜  src git:(main) g++ main.cpp
➜  src git:(main) cat test.txt
11 45 11 45 90
0011 45 011 0045 11 45 90    11
45 11 45 90
➜  src git:(main) ./a.out < test.txt
1, 3
1, 8
```

6. Сравнение работы алгоритмов сортировки

Кол-во слов в паттерне (m)	Кол-во слов в тексте (n)	Эффективная Z – функция O(m+n) (s)	Наивный алгоритм O(m*n) (s)
100	10^4	0.0005927	0.0007257
100	10^5	0.0020764	0.0028998
100	10^6	0.0153696	0.0344997
100	10^7	0.0197	0.1753209



7. Выводы

В ходе выполнения данной лабораторной работы я изучил алгоритмы поиска подстроки в строке и реализовал поиск одного образца, основанный на построении Z-блоков. Этот алгоритм имеет сложность O(m+n), где n – длина текста и m – длина паттерна, что позволяет фактически линейно производить поиск подстроки в строке.

Z-функция - простой, понятный и удобный алгоритм, который позволяет оптимизировать наивный квадратичный алгоритм до линейного.