**Московский авиационный институт (национальный исследовательский университет)**

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

**Лабораторная работа №4**

**по предмету "Дискретный анализ"**

Студент: Елистратова П.А.

Преподаватель: Макаров Н.К.

Группа: М8О-210Б-21

Дата:

Оценка:

Подпись:

Оглавление

[Цель работы 3](#__RefHeading___Toc3871_94863195)

[Постановка задачи 3](#__RefHeading___Toc3873_94863195)

[Описание 3](#__RefHeading___Toc1694_1904022149)

[Реализация 3](#__RefHeading___Toc1694_19040221493)

[Код программы 4](#__RefHeading___Toc1694_190402214931)

[Вывод 7](#__RefHeading___Toc422_3869341128)

# **Цель работы**

Реализовать алгоритм поиска построки в строке на Си++.

# **Постановка задачи**

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.

# **Описание**

Для строки S и позиции i > 0 определим:

**Zi(S)** — длина наибольшей подстроки S, которая начинается в i и совпадает с префиксом S.

**Z-блок** в i, для которого Zi > 0 — интервал, начинающийся в i и кончающийся в позиции i + Zi - 1.

**ri** — крайний правый конец Z-блоков, начинающихся не позднее позиции i, то есть наибольшее значение i + Zi — 1 по всем 0 < j ≤ i, для которых Zj > 0.

# **Реализация**

Считаем сначала заданный текст, а затем паттерн. Добавим наш текст в вектор к паттерну, разделив их символом «$». Идея алгоритма заключается в том чтобы посчитать Z-функцию для объединённых паттерна и текста, и тогда если какое-то Zi равно длине паттерна, значит с позиции i мы имеем полное вхождение нашего паттерна в текст.

Алгоритм вычисления Z-функции следующий:

Будем поддерживать координаты [left; right] самого правого Z-блока. Тогда если текущий индекс, для которого мы хотим посчитать очередное значение Z-функции, — это i, мы имеем один из двух вариантов:

i > right— т.е. текущая позиция лежит за пределами того, что мы уже успели обработать. Тогда будем вычисляем функцию тривиальным алгоритмом.

i ≤ right — т.е. текущая позиция лежит внутри отрезка совпадения [left; right] . Тогда мы можем использовать уже подсчитанные предыдущие значения Z-функции. Дальше снова действуем тривиальным алгоритмом — потому что после границы r могло обнаружиться продолжение Z-блока.

# **Код программы**

**lab4.cpp**

#include <iostream>

#include <stdio.h>

#include <string.h>

template <typename T>

class VectorClass

{

T\* data;

int capacity;

int current;

public:

VectorClass()

{

data = new T[1];

capacity = 1;

current = 0;

}

~VectorClass()

{

delete[] data;

}

void push(T value)

{

if (current == capacity)

{

T\* temp = new T[2 \* capacity];

for (int i = 0; i < capacity; i++)

{

temp[i] = data[i];

}

delete[] data;

capacity \*= 2;

data = temp;

}

data[current] = value;

++current;

}

void push(T value, int index)

{

if (index == capacity)

{

push(value);

}

else

{

data[index] = value;

}

}

T get(int index)

{

return data[index];

}

int size()

{

return current;

}

};

int\* ZFunction(VectorClass<char> &s, int\* Z)

{

int n = s.size();

int left = 0;

int right = 0;

for (int i = 1; i < n; ++i)

{

if (i < right)

{

if (right - i < Z[i - left])

{

Z[i] = right - i;

}

else

{

Z[i] = Z[i - left];

}

}

while (s.get(Z[i]) == s.get(i + Z[i]))

{

Z[i] += 1;

}

if (i + Z[i] > right)

{

left = i;

right = i + Z[i];

}

}

return Z;

}

int main()

{

VectorClass<char> P;

VectorClass<char> T;

char symbol;

while ('\n' != (symbol = getchar()))

{

T.push(symbol);

}

while ('\n' != (symbol = getchar()))

{

P.push(symbol);

}

int n = P.size();

int m = T.size();

P.push('$');

for (int i = 0; i < T.size(); ++i)

{

P.push(T.get(i));

}

int Z[n + m + 1];

memset(Z, 0, (n + m + 1) \* sizeof(int));

ZFunction(P, Z);

for (int i = 0; i < n + m + 1; ++i)

{

if (Z[i] == n)

{

std::cout << i - n - 1 << "\n";

}

}

return 0;

}

**Пример работы**

polina@pelis:~/diskran/lab4$ cat input.txt

abacaba

ab

polina@pelis:~/diskran/lab4$ g++ lab4.cpp

polina@pelis:~/diskran/lab4$ ./a.out < input.txt

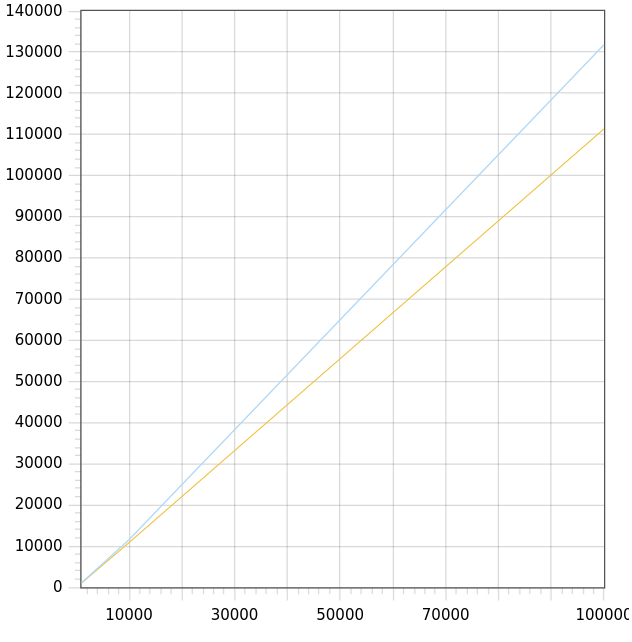
0

4

**Сравнения производительности**

Сравним алгоритм поиска подстроки в строке с помощью Z-блоков с наивным алгоритмом. По оси X — количество входных символов, по оси Y — время в мс.

Голубой график — наивный алгоритм, оранжевый график — с использованием Z-функции.

Как мы видим, алгоритм поиска подстроки в строке с помощью Z-блоков работает быстрее. Конечно, потому что не приходится повторно сравнивать символы в случаях, когда для ускорения расчёта Z-функции мы можем использовать уже вычисленные значения.

# **Вывод**

Выполнив данную лабораторную работу, я реализовала алгоритм поиска подстроки в строке. Построение Z-функции работает за O(n+m), где n — длина паттерна, а m — длина текста, а поиск всех вхождений работает за O(m). Это гораздо эффективнее наивного алгоритма, который в худшем случае работает за O(n\*m). Также положительной чертой этого алгоритма, является то, что он относительно прост для понимания.