МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Построение и анализ алгоритмов»
Тема: Ахо-Корасик

Студент гр. 9383 _____ Хотяков Е.П.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмом Ахо-Корасик, реализовать алгоритм на одном из языков программирования.

Задание.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, $1 \leq |T| \leq 100000$).

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p_1,\ldots p_n\}$ $1 \leq |p_i| \leq 75$

Все строки содержат символы из алфавита {A,C,G,T,N}

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

NTAG

3

TAGT

TAG

T

Sample Output:

2 2

2 3

Задание 2.

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Например, образец ab??c?ab??c? с джокером ?? встречается дважды в тексте xabvccbababcaxxabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A,C,G,T,N}

Вход:

Текст (T, $1 \leq |T| \leq 100000$)

Шаблон (P, $1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Основные теоретические положения.

Чтобы говорить об алгоритме необходимо ввести ряд понятий:

Бор — структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

Бор для набора образцов {he,she,his,hers}:

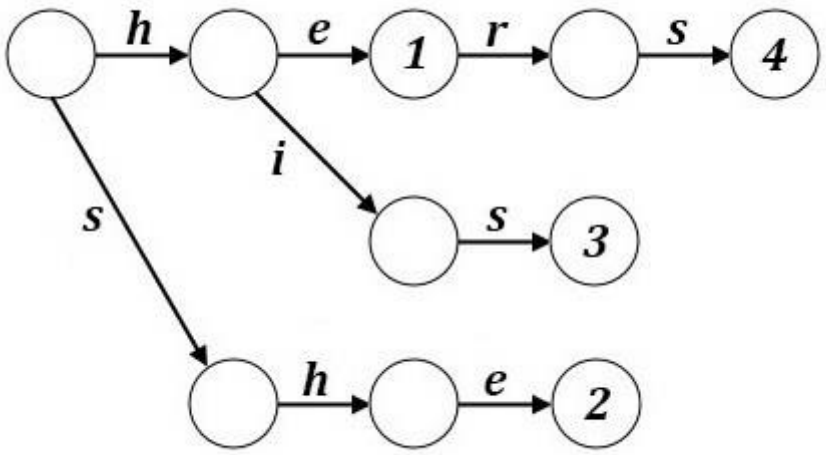

Рисунок 1 – пример бора для набора образцов

Корень обозначается символом, которого нет в алфавите(у меня это ‘.’)

Видно, что некоторые вершины имеют число – это обозначает, что вершина является *терминальной* (конечной) для определенного образца.

Назовем *суффиксной ссылкой* вершины *v* указатель на вершину *u*, такую что строка *u* — наибольший собственный суффикс строки *v*, или, если такой вершины нет в боре, то указатель на корень. В частности, ссылка из корня ведет в него же.

Например, для бора:

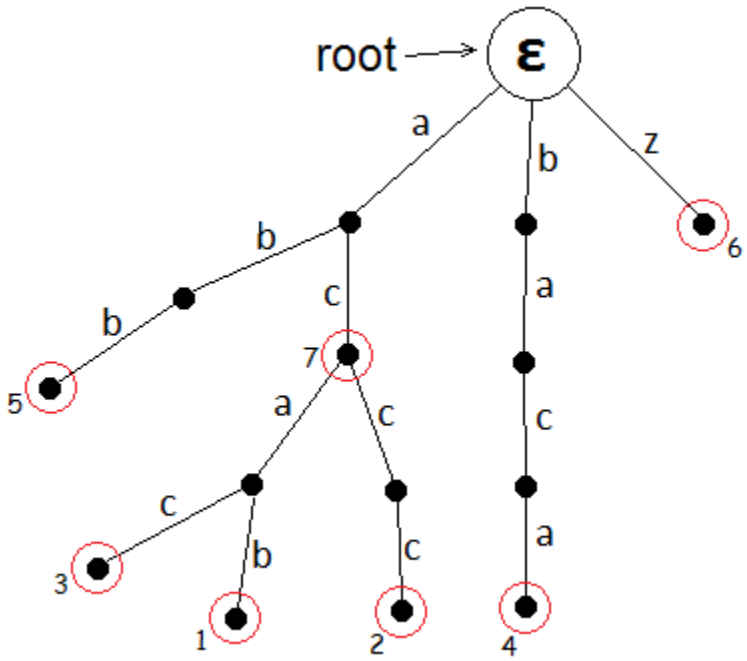

Рисунок 2 – пример бора

Суффиксные ссылки будут выглядеть следующим образом:

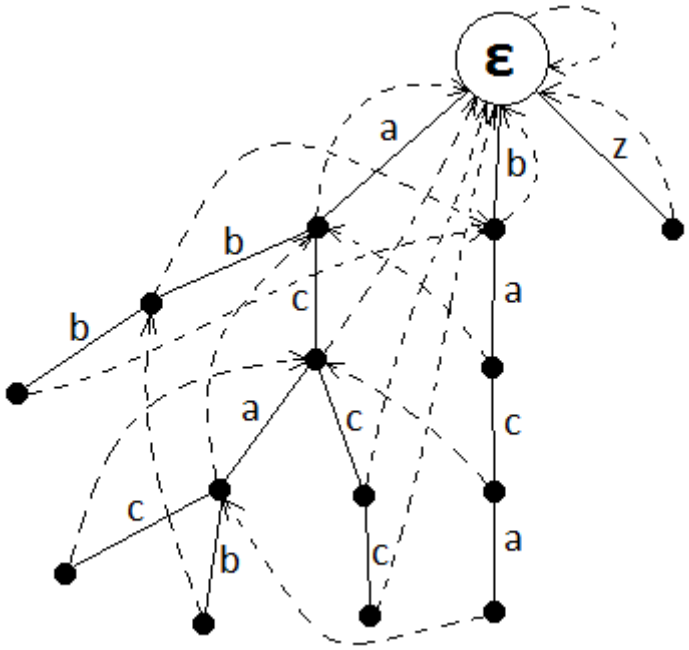

Рисунок 3 – пример суффиксных ссылок в боре

Наша задача — построить *конечный детерминированный автомат*. Состояние автомата — это какая-то вершина бора. Переход из состояний осуществляется по двум параметрам — текущей вершине *v* и символу *ch*. по которому нам надо сдвинуться из этой вершины. Необходимо найти вершину

u, которая обозначает наидлиннейшую строку, состоящую из суффикса строки *v* (возможно нулевого) + символа *ch*. Если такого в боре нет, то идем в корень.

Описание алгоритма.

Алгоритм состоит из трех этапов: построение бора, добавление суффиксных ссылок и самого прохода с помощью автомата.

1. Построение бора.

Бор строится из подстрок, переданных для поиска. Рассматривая каждый образец последовательно добавляем вершины в бор.

- если по текущему символу подстроки можно перейти в следующую вершину – переходим
- если ребра с текущим символом нет – создаем ребро, добавляя вершину и переходим в нее

2. Преобразование бора, добавлением суффиксных ссылок для каждой вершины.

При построении следует помнить:

- ссылка из корня идет в корень

- ссылка из потомка корня идет в корень

- ссылка из любой другой вершины с ребром e ищется следующим образом:

- смотрим на родителя
- пытаемся пройти по ребру e
- если не получается поднимаемся выше и повторяем
- если уперлись в корень и так и не нашли ребро – что ж, ссылка пойдет в корень
- иначе запоминаем ссылку на ту вершину, в которую попали, переходя по ребрам

Также еще одним этапом является построение конечной ссылки. Для этого:

- ходим по ссылкам, пока не упремся в терминальную – конечную вершину
- нашли – отлично, запоминаем
- не нашли и уперлись в корень – что ж, ссылка пуста

3. Сам алгоритм.

- пытаемся перейти в автомате по ребру с текущим рассматриваемым символом
- если такое ребро есть – переходим
- если такого ребра нет – переходим по суффиксной ссылке (если были в корне, то перейдем в него же) и повторяем с первой точки
- если попали в терминальную вершину – то ура, нашли вхождение – запоминаем
- проходим цепочку конечных ссылок, запоминая результаты

Для решения задания с джокером используется тот же алгоритм с некоторыми изменениями:

- выделяем максимальные подстроки, не содержащие джокера из заданного шаблона
- для каждого такого образца запоминаем смещение
- внутри алгоритма создаем дополнительный массив, заполненный 0, длинна которого равна длине текста
- запускаем поиск по тексту
- если нашли подстроку, то увеличиваем на 1 значение ячейки, которое соответствует разности между номером начального символа образца в тексте и смещением образца (если их несколько – то проделать со всеми)

Таким образом получится, что шаблонная подстрока будет начинаться в тех местах текста, для которых соответствующая ячейка массива содержит количество образцов с учетом кратности.

Сложность алгоритма Ахо-Корасика: O(M*k+N+t), где M – размер бора, k – размер алфавита, t - количество всех возможных вхождений всех строк-образцов, а N – длина исходной строки.

Описание функций и структур данных.

struct BohrNode – для хранения бора и работы с ним.

Его поля:

char symbol - хранит ребро для перехода в следующую вершину (вес ребра – символ char)

int terminalNodeNumber - хранит номер образца соответствующий конечной (терминальной вершине)

*std::vector< BohrNode *> next* - хранит ссылки на детей вершины

Bor parentNod;* - хранит ссылку на родителя

*BohrNode *suffixLink* - хранит суффиксную ссылку

*BohrNode *suffixLinkEnd* - хранит конечную ссылку

Функции и методы:

BohrNode *findNext(char nextSymbol) – метод для поиска возможного перехода к сыну текущего узла.

BohrNode *buildBohr(std::vector<std::pair<std::string, int>> &stringArray) – функция для построения Бора. Функция возвращает корень бора.

void addSinglePrefix(BohrNode *&node) – функция добавляет префиксную ссылку для переданного узла.

void addPrefixes(BohrNode *root) – функция добавляет все префиксные ссылки и конечные префиксные ссылки для всего Бора.

std::vector<int> AhoCorasick(BohrNode *root, std::string &text, std::vector<std::pair<std::string, int>> &stringArray) – функция реализующая алгоритм Ахо-Корасика.

Тестирование.

Задание 1.

Входные данные:

NTAG

3

TAGT

TAG

T

Выходные данные:

2 2

2 3

Входные данные:

ABCDEF

5

A

B

CD

CDEF

DC

Выходные данные:

1 1

2 2

3 3

3 4

Задание 2.

Входные данные:

ACTANCA

A$$A$

$

Выходные данные:

1

Входные данные:

ACTANCAACTAN

A$$A$

$

Выходные данные:

1

4

8

Выводы.

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Ахо-Корасик, который находит все вхождения подстрок шаблонов в заданный текст. Также был реализован поиск подстроки с джокером с использованием данного алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл task1.cpp:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

struct BohrNode
{
    char symbol;
    std::vector<BohrNode *> next;
    BohrNode *suffixLink = nullptr;
    BohrNode *suffixLinkEnd = nullptr;
    BohrNode *parentNode = nullptr;
    int terminalNodeNumber;

    BohrNode(char newSymbol, BohrNode *parent)
    {
        symbol = newSymbol;
        terminalNodeNumber = -1;
        suffixLink = nullptr;
        suffixLinkEnd = nullptr;
        parentNode = parent;
    }

    ~BohrNode()
    {
        for (auto i = next.begin(); i != next.end(); i++)
            delete *i;
    }

    BohrNode *findNext(char nextSymbol)
    {
```

```
            for (auto i = next.begin(); i != next.end(); i++)
            {
                if (nextSymbol == (*i)->symbol)
                {
                    return *i;
                }
            }
            return nullptr;
        }
    };

    BohrNode *buildBohr(std::vector<std::string> &stringArray)
    {
        BohrNode *root = new BohrNode('.', nullptr); //символ стартовой
вершины
        BohrNode *curNode;
        BohrNode *tmp = nullptr;
        for (int i = 0; i < stringArray.size(); i++)
        {
            curNode = root;
            for (int j = 0; j < stringArray[i].size(); j++)
            {
                tmp = curNode->findNext(stringArray[i][j]);
                if (tmp)
                    curNode = tmp;
                else
                {
                    curNode->next.push_back(new BohrNode(stringArray[i][j],
curNode));
                    curNode = curNode->next.back();
                }
            }
            curNode->terminalNodeNumber = i;
        }
        return root;
    }
```

```
void addSinglePrefix(BohrNode *&node)
{
    BohrNode *tmp;
    if (node->symbol == '.')
    {
        node->suffixLink = node;
        return;
    }
    BohrNode *curNode = node->parentNode;
    if (curNode->symbol == '.')
    {
        node->suffixLink = curNode;
        return;
    }
    curNode = curNode->suffixLink;
    do
    {
        tmp = curNode->findNext(node->symbol);
        if (tmp)
        {
            node->suffixLink = tmp;
            while (tmp->terminalNodeNumber == -1)
            {
                if (tmp->symbol == '.')
                    return;
                tmp = tmp->suffixLink;
            }
            node->suffixLinkEnd = tmp;
            return;
        }
        curNode = curNode->suffixLink;
    } while (curNode->symbol != '.');
    node->suffixLink = curNode; //Сейчас curNode - корень
}

void addPrefixes(BohrNode *root)
```

```
    {
        std::queue<BohrNode *> queue;
        BohrNode *curNode = root;
        for (auto i = curNode->next.begin(); i != curNode->next.end(); i++)
        {
            queue.push(*i);
        }
        curNode = queue.front();
        while (!queue.empty())
        {
            addSinglePrefix(curNode);
            queue.pop();
            for (auto i = curNode->next.begin(); i != curNode->next.end();
i++)
            {
                queue.push(*i);
            }
            curNode = queue.front();
        }
    }

    std::vector<std::string> readData()
    {
        int N;
        std::vector<std::string> stringArray;
        std::cin >> N;
        stringArray.resize(N);
        for (int i = 0; i < N; i++)
            std::cin >> stringArray[i];
        return stringArray;
    }

    std::vector<std::pair<int, int>> AhoCorasick(BohrNode *root, std::string
&text, std::vector<std::string> stringArray)
    {
        BohrNode *curNode = root;
        BohrNode *tmp;
```

```
        std::vector<std::pair<int, int>> result;
        for (int i = 0; i < text.length(); i++)
        {
            tmp = curNode->findNext(text[i]);
            if (tmp)
            {
                curNode = tmp;
            }
            while (!tmp)
            {
                if (curNode->symbol == '.')
                {
                    tmp = curNode;
                    break;
                }
                curNode = curNode->suffixLink;
                tmp = curNode->findNext(text[i]);
                if (tmp)
                {
                    curNode = tmp;
                    break;
                }
            }
            if (curNode->terminalNodeNumber != -1)
            {
                result.push_back(std::make_pair<int, int>(i -
stringArray[curNode->terminalNodeNumber].size() + 1, curNode-
>terminalNodeNumber + 1));
            }
            tmp = curNode->suffixLinkEnd;
            while (tmp)
            {
                result.push_back(std::make_pair<int, int>(i -
stringArray[tmp->terminalNodeNumber].size() + 1, tmp->terminalNodeNumber +
1));
                tmp = tmp->suffixLinkEnd;
            }
        }
```

```
    return result;
}

bool pairCmp(std::pair<int, int> a, std::pair<int, int> b)
{
    if (a.first == b.first)
        return a.second < b.second;
    return a.first < b.first;
}

void printResult(std::vector<std::pair<int, int>> &result)
{
    sort(result.begin(), result.end(), pairCmp);
    for (auto i = result.begin(); i != result.end(); i++)
    {
        std::cout << i->first + 1 << ' ' << i->second << '\n';
    }
}

int main()
{
    std::string text;

    std::cin >> text;
    std::vector<std::string> stringArray = readData();
    BohrNode *root = buildBohr(stringArray);
    root->suffixLink = root;
    addPrefixes(root);
    std::vector<std::pair<int, int>> result = AhoCorasick(root, text,
stringArray);
    printResult(result);
    return 0;
}
```

Файл task2.cpp:

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

```
#include <algorithm>
#include <queue>

struct BohrNode
{
    char symbol;
    std::vector<BohrNode *> next;
    BohrNode *suffixLink = nullptr;
    BohrNode *suffixLinkEnd = nullptr;
    BohrNode *parentNode = nullptr;
    std::vector<std::pair<int, int>> terminalNodeNumber; //будет хранить смещения текущей строки. А количество элементов массива - это кратность подстроки

    BohrNode(char newSymbol, BohrNode *parent)
    {
        symbol = newSymbol;
        suffixLink = nullptr;
        suffixLinkEnd = nullptr;
        parentNode = parent;
    }

    ~BohrNode()
    {
        for (auto i = next.begin(); i != next.end(); i++)
            delete *i;
    }

    BohrNode *findNext(char nextSymbol)
    {
        for (auto i = next.begin(); i != next.end(); i++)
        {
            if (nextSymbol == (*i)->symbol)
            {
                return *i;
            }
        }
```

```
            return nullptr;
        }
    };

    BohrNode *buildBohr(std::vector<std::pair<std::string, int>>
&stringArray)
    {
        BohrNode *root = new BohrNode('.', nullptr); //символ стартовой
вершины
        BohrNode *curNode;
        BohrNode *tmp = nullptr;
        for (int i = 0; i < stringArray.size(); i++)
        {
            curNode = root;
            for (int j = 0; j < stringArray[i].first.size(); j++)
            {
                tmp = curNode->findNext(stringArray[i].first[j]);
                if (tmp)
                    curNode = tmp;
                else
                {
                    curNode->next.push_back(new
BohrNode(stringArray[i].first[j], curNode));
                    curNode = curNode->next.back();
                }
            }
            curNode-
>terminalNodeNumber.push_back(std::make_pair(stringArray[i].second, i));
        }
        return root;
    }

    void addSinglePrefix(BohrNode *&node)
    {
        BohrNode *tmp;
        if (node->symbol == '.')
        {
            node->suffixLink = node;
```

```
        return;
    }
    BohrNode *curNode = node->parentNode;
    if (curNode->symbol == '.')
    {
        node->suffixLink = curNode;
        return;
    }
    curNode = curNode->suffixLink;
    tmp = curNode->findNext(node->symbol);
    while (!tmp)
    {
        if (curNode->symbol == '.')
        {
            tmp = curNode;
            break;
        }
        curNode = curNode->suffixLink;
        tmp = curNode->findNext(node->symbol);
    }
    node->suffixLink = tmp;
    BohrNode *tmp2 = tmp;
    while (tmp2->terminalNodeNumber.empty())
    {
        if (tmp2->symbol == '.')
            break;
        tmp2 = tmp2->suffixLink;
    }
    if (!tmp2->terminalNodeNumber.empty())
    {
        node->suffixLinkEnd = tmp2;
    }
}

void addPrefixes(BohrNode *root)
{
```

```
            std::queue<BohrNode *> queue;
            BohrNode *curNode = root;
            for (auto i = curNode->next.begin(); i != curNode->next.end(); i++)
            {
                queue.push(*i);
            }
            curNode = queue.front();
            while (!queue.empty())
            {
                addSinglePrefix(curNode);
                queue.pop();
                for (auto i = curNode->next.begin(); i != curNode->next.end();
i++)
                {
                    queue.push(*i);
                }
                curNode = queue.front();
            }
        }


        std::vector<int> AhoCorasick(BohrNode *root, std::string &text,
std::vector<std::pair<std::string, int>> &stringArray)
        {
            BohrNode *curNode = root;
            BohrNode *tmp;
            std::vector<int> result(text.size());
            for (int i = 0; i < text.length(); i++)
            {
                tmp = curNode->findNext(text[i]);
                if (tmp)
                {
                    curNode = tmp;
                }
                while (!tmp)
                {
                    if (curNode->symbol == '.')
                    {
```

```
                    tmp = curNode;
                    break;
                }
                curNode = curNode->suffixLink;
                tmp = curNode->findNext(text[i]);
                if (tmp)
                {
                    curNode = tmp;
                    break;
                }
            }
            if (!curNode->terminalNodeNumber.empty())
            {
                for (int j = 0; j < curNode->terminalNodeNumber.size(); j++)
                {
                    if (curNode->terminalNodeNumber[j].first <= i)
                        result[i - stringArray[curNode-
>terminalNodeNumber[j].second].first.size() + 1 - curNode-
>terminalNodeNumber[j].first]++;
                }

                //result.push_back(std::make_pair<int, int>(i -
stringArray[curNode->terminalNodeNumber].size() + 1, curNode-
>terminalNodeNumber + 1));
            }
            tmp = curNode->suffixLinkEnd;
            while (tmp)
            {
                for (int j = 0; j < curNode->terminalNodeNumber.size(); j++)
                {
                    if (curNode->terminalNodeNumber[j].first <= i)
                        result[i - stringArray[tmp-
>terminalNodeNumber[j].second].first.size() + 1 - tmp-
>terminalNodeNumber[j].first]++;
                }
                // result.push_back(std::make_pair<int, int>(i -
stringArray[tmp->terminalNodeNumber].size() + 1, tmp->terminalNodeNumber +
1));
                tmp = tmp->suffixLinkEnd;
```

```
            }
        }
        return result;
    }

    void separateJokerString(std::string stringWithJoker, char separator,
std::vector<std::pair<std::string, int>> &stringArray)
    {
        int j = 0;
        stringArray.push_back(std::make_pair("", 0));
        for (int i = 0; i < stringWithJoker.length(); i++)
        {
            if (stringWithJoker[i] == separator &&
!stringArray[j].first.empty())
            {
                stringArray.push_back(std::make_pair("", 0));
                j++;
            }
            else if (stringWithJoker[i] == separator)
                continue;
            else if (stringArray[j].first.empty())
            {
                stringArray[j].first += stringWithJoker[i];
                stringArray[j].second = i;
            }
            else
                stringArray[j].first += stringWithJoker[i];
        }
        if (stringArray.back().first.empty())
            stringArray.pop_back();
    }

    void printResult(std::vector<int> &result, int length, int wordsNumber)
    {
        int resultL = result.size();
        for (int i = 0; i < resultL; i++)
        {
```

```
        if (result[i] == wordsNumber && i + length <= resultL)
            std::cout << i + 1 << '\n';
    }
}

int main()
{
    std::string text;
    char separator;
    std::string stringWithJoker;
    std::vector<std::pair<std::string, int>> stringArray;
    std::cin >> text >> stringWithJoker >> separator;
    separateJokerString(stringWithJoker, separator, stringArray);
    BohrNode *root = buildBohr(stringArray);
    root->suffixLink = root;
    addPrefixes(root);
    std::vector<int> result = AhoCorasick(root, text, stringArray);
    printResult(result, stringWithJoker.length(), stringArray.size());
    delete root;
    return 0;
}
```