Министерство Образования Республики Беларусь

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

**СТРУКТУРЫ ДАННЫХ**

**Запись/чтение массива структур в файл.**

**Лабораторная работа № 8, 9.**

Студент:

1курса 2 группы 1 подгруппы

Марушко Тимофей Фёдорович

Проверяющий:

Белодед Николай Иванович

**ОГЛАВЛЕНИЕ**

[**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 3**](#_Toc193833060)

[**1. ФИЛЬТРАЦИЯ ДАННЫХ 3**](#_Toc193833061)

[**2. АЛГОРИТМ КНУТА-МОРРИСА-ПРАТТА 6**](#_Toc193833062)

[**3. АЛГОРИТМ РАБИНА-КАРПА 9**](#_Toc193833063)

[**4. ИСПОЛЬЗОВАНИЕ STRING::FIND() 12**](#_Toc193833064)

[**РЕШЕНИЕ ОСНОВНЫХ ЗАДАЧ 14**](#_Toc193833065)

[**РЕШЕНИЕ ДОПОЛНИТЕЛЬНЫХ ЗАДАЧ 17**](#_Toc193833066)

Теоретические сведения

1. Фильтрация данных

**Фильтрация данных** — это процесс поиска и извлечения нужных данных из набора данных. Фильтрация может осуществляться на основе различных критериев, таких как значения полей, дата, время и т.д.

Для фильтрования данных в файле необходимо выполнить следующие шаги:

1. Открыть файл с исходными данными для чтения.
2. Прочитать данные из файла и сохранить их в переменной.
3. Применить фильтр на основе заданных критериев.
4. Записать отфильтрованные данные в новый файл.

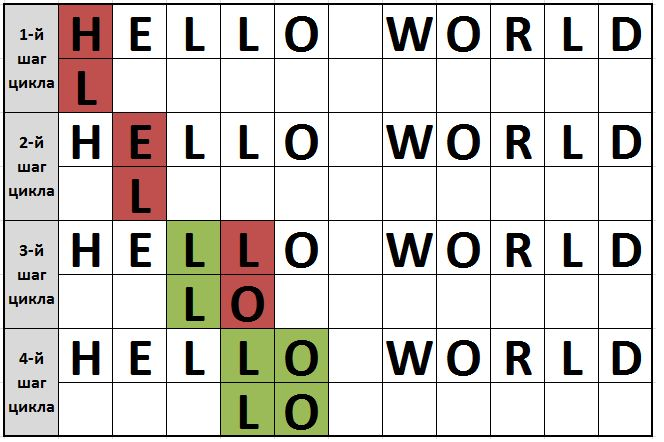
Далее мы разберём на примерах, как может выглядеть алгоритм поиска подстроки в строке. Примеры будут основываться на функциях стандартных библиотек, ведь именно в таких функциях и проявляются все удобства написания программ. А вот классический разбор алгоритма, основанный на циклах и сравнениях, также достаточно примечателен. Поэтому мы его рассмотрим в этом же уроке.

Сам алгоритм в принципе очень прост. Есть две строки. Например "Hello world"  и "lo".

Работать будем в два цикла:

1. Первый будет выполнять проход по всей строке, и искать местоположение **первой буквы** искомой строки ( "lo" ).
2. Второй, начиная с найденной позиции первой буквы – сверять, какие буквы стоят после неё и сколько из них подряд совпадают.

Проиллюстрируем поиск подстроки в строке:



На первых двух итерациях цикла сравниваемые буквы не будут совпадать (выделено красным). На третьей итерации искомая буква (первый символ искомого слова) совпала с символом в строке, где происходит поиск. При таком совпадении в работу включается второй цикл.

Он призван отсчитывать количество символов после первого в искомой строке, которые будут совпадать с символами в строке исходной. Если один из следующих символов не совпадает – цикл завершает свою работу. Нет смысла гонять цикл впустую, после первого несовпадения, так как уже понятно, что искомого тут нет.

На третьей итерации совпал только первый символ искомой строки, а вот второй уже не совпадает. Придется первому циклу продолжить работу. Четвертая итерация дает необходимые результаты – совпадают все символы искомой строки с частью исходной строки. А раз все символы совпали – подстрока найдена. Работу алгоритма можно закончить.

Сколько раз искомое слово встречается в строке и в каких местах? Именно это и призван контролировать третий параметр – int n – номер вхождения в стро-ку. Если поставить туда единицу – он найдет первое совпадение искомого. Если двойку, он заставит первый цикл пропустить найденное первое, и искать второе. Если тройку – искать третье и так далее. С каждым найденным искомым словом, этот счетчик вхождений уменьшается на единицу.

Посмотрим, как выглядит классический код поиска подстроки в строке в С++:

// Функция для поиска подстроки в строке

// + поиск позиции, с которой начинается подстрока

int pos(const char\* s, const char\* c, int n)

{

int i, j; // Счетчики для циклов

int lenC, lenS; // Длины строк

//Находим размеры строки исходника и искомого

for (lenC = 0; c[lenC]; lenC++);

for (lenS = 0; s[lenS]; lenS++);

for (i = 0; i <= lenS - lenC; i++) // Пока есть возможность поиска

{

for (j = 0; s[i + j] == c[j]; j++); // Проверяем совпадение посимвольно

// Если посимвольно совпадает по длине искомого

// Вернем из функции номер ячейки, откуда начинается совпадение

// Учитывать 0-терминатор ( '\0' )

if (j == lenC && !(n - 1))

return i;

if (j == lenC)

if (n - 1)

n--;

else

return i;

}

//Иначе вернем -1 как результат отсутствия подстроки

return -1;

}

int main()

{

const char\* s = "parapapa";

const char\* c = "pa";

….

}



Два цикла выполняют каждый свою задачу. Один топает по строке в надежде найти “голову” искомого слова (первый символ). Второй выясняет, есть ли после найденной “головы” “тело” искомого. Причем проверяет, не лежит ли это “тело” в конце строки. Т.е. не является ли длина найденного слова на единицу больше длины искомой строки, если учитывать, что в эту единицу попадает нулль-терминатор ( '\0' ).

Мы видим, что программа нашла начало подстроки **pa** в ячейках символьного массива с индексом 0 и 4. Но почему? Ведь в слове **parapapa 3**таких подстроки. Все дело в '\0' .

В целом смысл самого алгоритма на этом заканчивается. Больше никаких сложностей кроме нуля в конце строки нет.

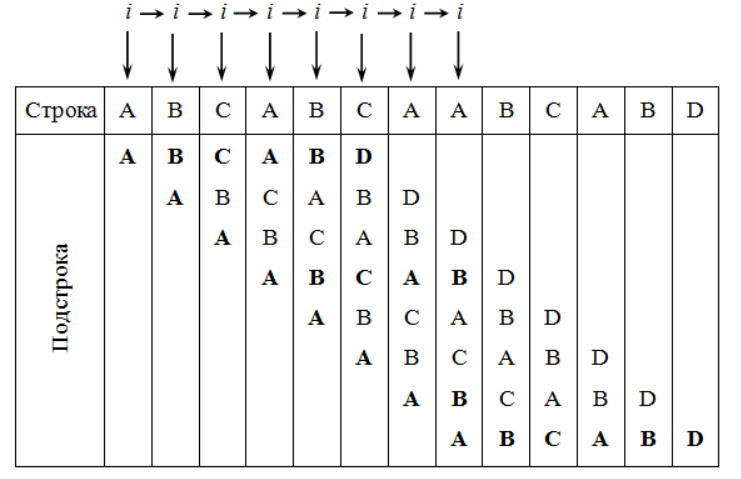
1. Алгоритм Кнута-Морриса-Пратта

Кнута-Морриса-Пратта (KMP) - это алгоритм поиска подстроки в строке, который использует метод сравнения шаблона с текстом, но в отличие от простого метода сравнения, KMP использует информацию о предыдущих сравнениях для ускорения процесса.

Алгоритм KMP основан на использовании префикс-функции, которая вычисляется для шаблона. **Префикс-функция для шаблона** - это массив значений, где pi[i] - это максимальная длина суффикса шаблона p[0...i], который одновременно является его префиксом. Массив pi можно вычислить за линейное время O(m), где m - длина шаблона.

Затем, используя префикс-функцию, KMP алгоритм сравнивает шаблон с текстом, начиная с начала строки. Если текущий символ текста не соответствует текущему символу шаблона, то алгоритм сдвигает шаблон на количество символов, равное разности текущей позиции и значения префикс-функции для предыдущей позиции. Это позволяет избежать повторных сравнений символов, которые уже были проверены и не соответствуют искомой подстроке.

Алгоритм продолжает сравнивать символы шаблона с текстом, пока не будет найдено первое вхождение подстроки в строку или пока не достигнут конец текста.



Асимптотическая сложность алгоритма Кнута-Морриса-Пратта - O(n+m), где n - длина текста, а m - длина шаблона. Он обладает лучшей производительностью, чем метод посимвольного сравнения, особенно для длинных строк и подстрок.

Для более подробного и углубленного изучения формулы и нюанмсов алгоритма КМП: [MAXimal :: algo :: Префикс-функция. Алгоритм Кнута-Морриса-Пратта (e-maxx.ru)](https://e-maxx.ru/algo/prefix_function)

Пример поиска подстроки в строке с использованием алгоритма Кнута-Морриса-Пратта:

// Функция вычисления префикс-функции для шаблона

vector<int> compute\_prefix\_function(const string& pattern) {

int m = pattern.length();

vector<int> pi(m); // массив для хранения значений префикс-функции

pi[0] = 0; // префикс-функция для первого символа всегда равна 0

int k = 0; // длина максимального суффикса, совпадающего с префиксом

for (int i = 1; i < m; i++) {

while (k > 0 && pattern[k] != pattern[i]) {

k = pi[k - 1]; // корректируем значение k, используя префикс-функцию для предыдущего символа

}

if (pattern[k] == pattern[i]) {

k++; // увеличиваем значение k, если символы совпадают

}

pi[i] = k; // сохраняем значение префикс-функции для i-го символа шаблона

}

return pi; // возвращаем массив значений префикс-функции

}

// Функция поиска подстроки в тексте с использованием алгоритма КМП

int kmp\_search(const string& pattern, const string& text) {

int m = pattern.length(); // длина шаблона

int n = text.length(); // длина текста

vector<int> pi = compute\_prefix\_function(pattern); // вычисляем префикс-функцию для шаблона

int k = 0; // длина максимального суффикса, совпадающего с префиксом шаблона

for (int i = 0; i < n; i++) {

while (k > 0 && pattern[k] != text[i]) {

k = pi[k - 1]; // корректируем значение k, используя префикс-функцию для предыдущего символа шаблона

}

if (pattern[k] == text[i]) {

k++; // увеличиваем значение k, если символы совпадают

}

if (k == m) { // если k равно длине шаблона, то мы нашли подстроку

return i - m + 1; // возвращаем позицию, с которой начинается подстрока

}

}

return -1;

}

int main() {

setlocale(LC\_ALL, "rus");

string pattern = "abc"; // задаем шаблон

string text = "abababc"; // задаем текст

….

}

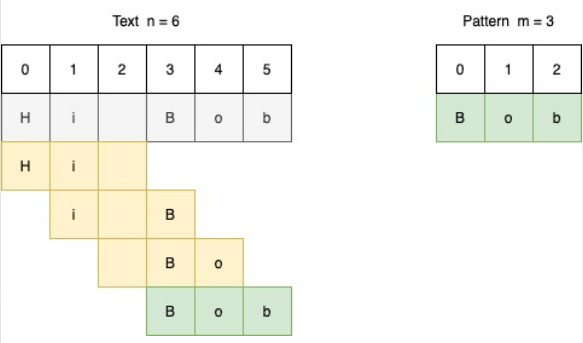


1. Алгоритм Рабина-Карпа

Алгоритм Рабина-Карпа (RK) — это алгоритм поиска подстроки в строке, который основан на хэшировании строк. Для каждого окна в строке, которое имеет длину подстроки, вычисляется хэш-значение. Затем сравнивается хэш-значение окна со значением хэша подстроки. Если они совпадают, то выполняется дополнительная проверка на совпадение каждого символа.

Наивный алгоритм поиска строки сравнивает заданный шаблон со всеми позициями в тексте. Это приводит к далеко не идеальной сложности времени исполнения O(nm), где n = длина текста, а m = длина шаблона.

Алгоритм Рабина-Карпа совершенствует этот подход за счёт того, что сравнение хешей двух строк выполняется за линейное время: для поиска совпадения это гораздо эффективнее, чем сравнение отдельных символов этих строк. Таким образом, алгоритм показывает лучшее время исполнения O(n+m).



С применением хеш-функции связаны два нюанса.

Во-первых, алгоритм хорош настолько, насколько хороша его хеш-функция. Если при использовании хеш-функции имеют место многочисленные ложные срабатывания, то сравнение символов будет выполняться слишком часто. В этом случае очень сложно считать этот метод более эффективным, чем наивный алгоритм.

Во-вторых, каждый раз, когда подстрока проходит по тексту, вычисляется новый хеш, что крайне неэффективно, ведь в этом случае производительность такая же, если не хуже, как у наивного алгоритма.

Обе эти проблемы решаются с помощью полиномиального хеша с операциями сложения и умножения. И хотя это не какой-то эксклюзив алгоритма Рабина-Карпа, которого нет в других алгоритмах, здесь он работает так же хорошо.

Алгоритм Рабина-Карпа работает следующим образом:

1. Вычисляем хэш-значение для подстроки, которую мы ищем.
2. Вычисляем хэш-значения для всех подстрок строки, которые имеют ту же длину, что и искомая подстрока.
3. Сравниваем хэш-значения подстрок. Если хэш-значения совпадают, то выполняем дополнительную проверку на совпадение каждого символа.

Для вычисления хэш-значения используется функция хэширования, которая преобразует строку в число. Обычно используется хэш-функция, которая основана на арифметических операциях над ASCII-кодами символов строки. Однако, хэш-функция должна быть такой, чтобы вероятность коллизий (ситуация, когда две разные строки имеют одно и то же хэш-значение) была минимальной.

Стоит отметить, что при использовании алгоритма Рабина-Карпа возможны ложные срабатывания, когда хэш-значения двух разных строк совпадают. Для уменьшения вероятности ложных срабатываний можно использовать несколько хэш-функций или дополнительно проверять совпадение каждого символа при совпадении хэш-значений.

Алгоритм Рабина-Карпа может быть эффективен при поиске нескольких подстрок в одной строке, так как при этом можно повторно использовать уже вычисленные хэш-значения.

Пример программы на C++, которая ищет подстроку в строке с помощью алгоритма Рабина-Карпа:

#include <iostream>

#include <string>

#include <vector>

using namespace std;

// Хэш-функция, которая суммирует ASCII-коды символов

int hash\_function(string str) {

int hash\_value = 0;

for (int i = 0; i < str.length(); i++) {

hash\_value += (int)str[i];

}

return hash\_value;

}

// Функция для поиска подстроки в строке с помощью алгоритма Рабина-Карпа

vector<int> rabinkarp(string str, string substr) {

vector<int> matches; // массив для хранения индексов найденных подстрок

int substr\_hash = hash\_function(substr); // хэш-значение искомой подстроки

for (int i = 0; i <= str.length() - substr.length(); i++) {

string current\_substr = str.substr(i, substr.length());

int current\_hash = hash\_function(current\_substr); // хэш-значение текущей подстроки

if (current\_hash == substr\_hash && current\_substr == substr) {

matches.push\_back(i); // добавляем индекс найденной подстроки в массив

}

}

return matches;

}

int main() {

setlocale(LC\_ALL, "rus");

string str = "Hello, world!";

string substr = "wor";

vector<int> matches = rabinkarp(str, substr);

if (matches.size() == 0) {

cout << "Подстрока не найдена" << endl;

}

else {

cout << "Подстрока найдена в следующих позициях: ";

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

cout << matches[i] << " ";

}

cout << endl;

}

return 0;

}



Однако в виду возможных неточностей при поиске выполнение поиска при помощи этого алгоритма в лабораторной работе мы не требуем.

1. Использование string::find()

Класс string в С++ снабжен методом find(), возвращающий номер ячейки, с которого начинается тело искомой строки в исходной строке.

Функция find() возвращает индекс первого вхождения подстроки или отдельного символа в строке:

#include <iostream>

#include <string>

int main()

{

std::string text{ "A friend in need is a friend indeed." };

std::cout << text.find("ed") << std::endl; // 14

std::cout << text.find("friend") << std::endl; // 2

std::cout << text.find('d') << std::endl; // 7

std::cout << text.find("apple") << std::endl; // 18446744073709551615

}

Если строка или символ не найдены (как в примере выше в последнем случае), то возвращается специальная константа std::string::npos, которая представляет очень большое число. И при поиске мы можем проверять результат функции find() на равенство этой константе.

Функция find() принимает вторым параметром номер символа, с которого начать поиск. Т.е. найдя первое вхождение, его значение увеличивается на единицу и find() продолжает поиск со следующего символа.

#include <iostream>

#include <string>

using namespace std;

int main()

{

string s = "parapapa";

int i = 0;

for (i = s.find("pa", i++); i != string::npos; i = s.find("pa", i + 1))

cout << i << endl;

}



В качестве третьего параметра передается количество символов из искомой строки, которые программа будет искать в тексте:

#include <iostream>

#include <string>

using namespace std;

int main()

{

string text{ "A friend in need is a friend indeed." };

string word{ "endless" };

// поиск с 10-го индекса 3 первых символов слова "endless", то есть "end"

cout << text.find("endless", 10, 3) << endl;

}



Функция **rfind**() работает аналогично функции find(), принимает те же самые параметры, только ищет подстроку в обратном порядке - с конца строки:

#include <iostream>

#include <string>

int main()

{

std::string text{ "A friend in need is a friend indeed." };

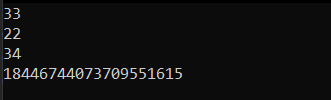
std::cout << text.rfind("ed") << std::endl; // 33

std::cout << text.rfind("friend") << std::endl; // 22

std::cout << text.rfind('d') << std::endl; // 34

std::cout << text.rfind("apple") << std::endl; // 18446744073709551615

}



Пара функций - *find\_first\_of()* и *find\_last\_of()* позволяют найти соответственно первый и последний индекс любого из набора символов.

Если нам надо найти позиции символов, которые НЕ представляют любой символ из набора, то мы можем использовать функции *find\_first\_not\_of()* (первая позиция) и *find\_last\_not\_of()* (последняя позиция).

Решение ОСНОВНЫХ ЗАДАЧ

**Условие:**

Для вашей БД реализовать 3 вида поиска: стандартный алгоритм поиска подстроки (в лр. рассматривался как первый алгоритм), поиск подстроки с помощью алгоритма Кнута-Морриса-Пратта, поиск подстроки с использованием string::find(). Поиск производить по одному типу данных, например, по цене.

**Решение:**

*#include* <iostream>

*#include* <fstream>

*#include* <vector>

*#include* <string>

*#include* <sstream>

*using* *namespace* std;

*// Разделение текста на предложения*

vector<*string*> splitIntoSentences(*const* *string&* text) {

    vector*<string>* sentences;

*stringstream* ss(text);

*string* sentence;

*while* (getline(ss, sentence, '.')) {

*if* (*!*sentence.empty()) {

            sentences.push\_back(sentence + '.');

        }

    }

*return* sentences;

}

*// Наивный поиск*

*bool* naiveSearch(*const* *string&* text, *const* *string&* word) {

*size\_t* len *=* word.length();

*for* (*size\_t* i *=* 0; i *<=* text.length() *-* len; i*++*) {

*if* (text.substr(i, len) == word) *return* *true*;

    }

*return* *false*;

}

*// Поиск с использованием string::find()*

*bool* findSearch(*const* *string&* text, *const* *string&* word) {

*return* text.find(word) *!=* *string*::npos;

}

*// Функция для построения префикс-функции КМП*

vector<*int*> computeKMPTable(*const* *string&* pattern) {

    vector*<int>* table(pattern.length(), 0);

*int* j *=* 0;

*for* (*size\_t* i *=* 1; i *<* pattern.length(); i*++*) {

*while* (j *>* 0 *&&* pattern[i] *!=* pattern[j]) {

            j *=* table[j *-* 1];

        }

*if* (pattern[i] *==* pattern[j]) {

            j*++*;

            table[i] *=* j;

        }

    }

*return* table;

}

*// Алгоритм Кнута-Морриса-Пратта (КМП)*

*bool* kmpSearch(*const* *string&* text, *const* *string&* pattern) {

    vector*<int>* table *=* computeKMPTable(pattern);

*int* j *=* 0;

*for* (*size\_t* i *=* 0; i *<* text.length(); i*++*) {

*while* (j *>* 0 *&&* text[i] *!=* pattern[j]) {

            j *=* table[j *-* 1];

        }

*if* (text[i] *==* pattern[j]) {

            j*++*;

*if* (j *==* pattern.length()) *return* *true*;

        }

    }

*return* *false*;

}

*string* highlightWord(*const* *string&* sentence, *const* *string&* word) {

*string* result *=* sentence;

*size\_t* pos *=* result.find(word);

*while* (pos *!=* *string*::npos) {

        result.insert(pos, "|");

        result.insert(pos *+* word.length() *+* 1, "|");

        pos *=* result.find(word, pos *+* word.length() *+* 2);

    }

*return* result;

}

*void* menu(*const* vector<*string*>*&* sentences) {

*while* (*true*) {

        cout << "\nEnter a word to search (or 'exit' to quit): ";

*string* word;

        cin >> word;

*if* (word == "exit") *break*;

        cout << "\nChoose search algorithm:\n";

        cout << "1. Naive Search\n";

        cout << "2. string::find()\n";

        cout << "3. Knuth-Morris-Pratt (KMP)\n";

        cout << "Enter choice: ";

*int* choice;

        cin >> choice;

        cout << "\nResults:\n";

*int* index *=* 1;

*for* (*const* *string&* sentence : sentences) {

*bool* found *=* *false*;

*switch* (choice) {

*case* 1: found *=* naiveSearch(sentence, word); *break*;

*case* 2: found *=* findSearch(sentence, word); *break*;

*case* 3: found *=* kmpSearch(sentence, word); *break*;

*default*: cout << "Invalid choice.\n"; *return*;

            }

*if* (found) {

                cout << index << ". " << highlightWord(sentence, word) << "\n";

            }

            index*++*;

        }

    }

}

*int* main() {

*ifstream* file("input\_text.txt");

*string* text((istreambuf\_iterator<*char*>(file)), istreambuf\_iterator<*char*>());

    vector*<string>* sentences *=* splitIntoSentences(text);

    menu(sentences);

*return* 0;

}

Решение Дополнительных задач

**Условие:**

Напишите программу, которая принимает на вход две строки и находит все вхождения первой строки во вторую строку. Выведите на экран позиции начала каждого вхождения.

**Решение:**

*// Функция для поиска всех вхождений подстроки*

vector<*int*> findAllOccurrences(*const* *string&* pattern, *const* *string&* text) {

    vector*<int>* positions;

*size\_t* pos *=* text.find(pattern);

*while* (pos *!=* *string*::npos) {

        positions.push\_back(pos);

        pos *=* text.find(pattern, pos *+* 1); *// Ищем следующее вхождение*

    }

*return* positions;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*case* 1: {

            vector*<int>* positions *=* findAllOccurrences(str1, str2);

*if* (positions.empty()) cout << "No occurrences found.\n";

*else* {

                cout << "Occurrences found at positions: ";

*for* (*int* pos : positions) cout << pos << " ";

                cout << endl;

            }

*break*;

        }

**Условие:**

Измените свою программу для поиска подстроки в строке так, чтобы она искала только первое вхождение. Если подстрока не найдена, программа должна выводить сообщение об этом.

**Решение:**

*// Функция для поиска первого вхождения подстроки*

*int* findFirstOccurrence(*const* *string&* pattern, *const* *string&* text) {

*return* text.find(pattern);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*case* 2: {

*int* pos *=* findFirstOccurrence(str1, str2);

*if* (pos *==* *string*::npos) cout << "Substring not found.\n";

*else* cout << "First occurrence at position: " << pos << endl;

*break*;

        }

**Условие:**

Напишите программу, которая находит наиболее длинную общую подстроку двух заданных строк. Выведите найденную подстроку на экран.

**Решение:**

*// Функция для поиска самой длинной общей подстроки*

*string* longestCommonSubstring(*const* *string&* str1, *const* *string&* str2) {

*int* len1 *=* str1.length(), len2 *=* str2.length();

    vector*<*vector*<int>>* dp(len1 *+* 1, vector<*int*>(len2 *+* 1, 0));

*int* maxLength *=* 0, endIndex *=* 0;

*for* (*int* i *=* 1; i *<=* len1; i*++*) {

*for* (*int* j *=* 1; j *<=* len2; j*++*) {

*if* (str1[i *-* 1] *==* str2[j *-* 1]) {

                dp[i][j] *=* dp[i *-* 1][j *-* 1] *+* 1;

*if* (dp[i][j] *>* maxLength) {

                    maxLength *=* dp[i][j];

                    endIndex *=* i;

                }

            }

        }

    }

*return* str1.substr(endIndex *-* maxLength, maxLength); *// Возвращаем найденную подстроку*

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*case* 3: {

*string* lcs *=* longestCommonSubstring(str1, str2);

*if* (lcs.empty()) cout << "No common substring found.\n";

*else* cout << "Longest common substring: " << lcs << endl;

*break*;

        }