**Отчёт по лабораторным работам**

**Вариант 1**

**Лабораторная работа 4.**

**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

**Студент:**

**Адамович Карианна Павлловна, 2-3**

**Преподаватель:**

**Бурмакова Анастасия Владимировна**

**г.Минск, 2025**

**ЦЕЛЬ РАБОТЫ:** освоить общие принципы решения задач методом динамического программирования, сравнить полученные решения задач с рекурсивным методом.

**ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:**

***Задание 1.***

На языке С++ сгенерировать случайным образом строку букв латинского алфавита S1 длиной 300 символов и S2 длиной 200.

Для создания генератора случайных строк была создана функция - generateRandomString.

#include <iostream>

#include <string>

#include <cstdlib>

#include <ctime>

std::string generateRandomString(int length) {

const std::string alphabet = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

std::string result;

result.reserve(length);

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

result += alphabet[rand() % alphabet.size()];

}

return result;

}

int main() {

setlocale(LC\_ALL, "Russian");

std::srand(std::time(0));

// --- Задание 1: Генерация случайных строк ---

std::cout << "Задание 1: Генерация случайных строк\n";

std::string S1 = generateRandomString(300);

std::string S2 = generateRandomString(200);

std::cout << "Сгенерированная строка S1: " << S1 << "\n";

std::cout << "Сгенерированная строка S2: " << S2 << "\n";

return 0;

}

1)Объявление функции для генерации случайной строки

Функция generateRandomString(int length) генерирует случайную строку указанной длины из символов латинского алфавита (как строчных, так и прописных).

const std::string alphabet: Строка, содержащая все буквы латинского алфавита.

std::string result: Переменная для хранения сгенерированной строки.

result.reserve(length): Зарезервировать память для строки заданной длины.

2)Генерация случайных символов

Цикл for проходит по всем символам строки и добавляет случайный символ из строки alphabet.

rand() % alphabet.size(): Используется для генерации случайного индекса символа.

3)Основная функция main

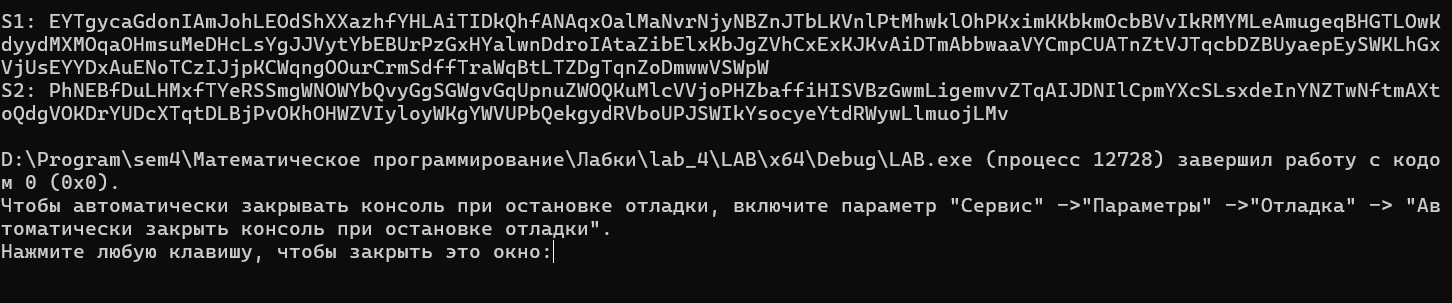
std::srand(std::time(0)): Инициализация генератора случайных чисел текущим временем.

std::string S1 = generateRandomString(300): Генерация строки S1 длиной 300 символов.

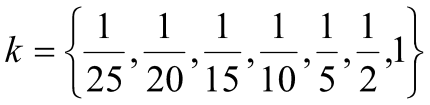
std::string S2 = generateRandomString(200): Генерация строки S2 длиной 200 символов.

std::cout << "S1: " << S1 << std::endl: Вывод строки S1 на экран.

std::cout << "S2: " << S2 << std::endl: Вывод строки S2 на экран.



***Задание 2.***

Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – дистанцию Левенштейна для , где - длина строки ,  - строка состоящая из первых  символов строки . (копии экрана и код вставить в отчет).

***Задание 3.***

Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от . (копии экрана и график вставить в отчет).

Для выполнения этого задания воспользуемся предоставленными в методическом пособии классами.

// - Levenshtein.h

// -- дистанции Левенштeйна (динамическое программирование)

int levenshtein(

int lx, // длина слова x

const char x[], // слово длиной lx

int ly, // длина слова y

const char y[] // слово y

);

// -- дистанции Левенштeйна (рекурсия)

int levenshtein\_r(

int lx, // длина строки x

const char x[], // строка длиной lx

int ly, // длина строки y

const char y[] // строка y

);

Функция levenshtein (Динамическое программирование):

Описание: Это оптимизированный подход, который использует таблицу (обычно двумерный массив) для сохранения промежуточных результатов. Он предотвращает многократное вычисление одинаковых подзадач, что существенно ускоряет выполнение.

Параметры:

lx: длина первой строки x.

x[]: массив символов, представляющий первую строку.

ly: длина второй строки y.

y[]: массив символов, представляющий вторую строку.

Функция levenshtein\_r (Рекурсия):

Описание: Решение на основе рекурсивного подхода, который повторно вычисляет подзадачи, вызывая сам себя.

Параметры:

Аналогичны функции levenshtein (lx, x[], ly, y[]).

// - Levenshtein.cpp

#include "stdafx.h"

#include <iomanip>

#include <algorithm>

#include "Levenshtein.h"

#define DD(i,j) d[(i)\*(ly+1)+(j)]

int min3(int x1, int x2, int x3)

{ return std::min(std::min(x1,x2),x3); }

int levenshtein(int lx, const char x[],int ly, const char y[])

{

int \*d = new int[(lx+1)\*(ly+1)];

for(int i = 0; i <= lx; i++) DD(i, 0) = i;

for(int j = 0; j <= ly; j++) DD(0, j) = j;

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

for (int j = 1; j <= ly; j++)

{

DD(i,j) = min3(DD(i-1, j) + 1, DD(i, j-1) + 1,

DD(i-1, j-1) + (x[i-1]==y[j-1]?0:1));

}

return DD(lx,ly);

}

int levenshtein\_r(

int lx, const char x[],

int ly, const char y[]

)

{

int rc = 0;

if (lx == 0) rc = ly;

else if (ly == 0) rc = lx;

else if (lx == 1 && ly == 1 && x[0] == y[0]) rc = 0;

else if (lx == 1 && ly == 1 && x[0] != y[0]) rc = 1;

else rc = min3(

levenshtein\_r(lx-1, x, ly, y)+1,

levenshtein\_r(lx, x, ly-1, y)+1,

levenshtein\_r(lx-1, x, ly-1, y)+(x[lx-1] == y[ly-1]?0:1)

);

return rc;

};

Функция levenshtein\_r (рекурсия)

Рекурсивно сравниваются последние символы строк и рассчитывается минимальная стоимость операций (вставка, удаление, замена).

В базовых случаях:

Если одна из строк пуста, расстояние равно длине другой строки. Если обе строки состоят из одного символа, возвращается 0 (если символы равны) или 1 (если они разные). Вызывает себя трижды для всех возможных преобразований.

Ключевая особенность:

Простота реализации, но отсутствие оптимизации (нет кеширования промежуточных результатов).

Производительность: Требует значительных ресурсов (особенно стековой памяти) для обработки большого количества рекурсивных вызовов.

Функция levenshtein (динамическое программирование)

Создается двумерная таблица d размером (lx+1)×(ly+1)(lx + 1) \times (ly + 1), где lxlx и lyly — длины строк x и y. Таблица заполняется минимальными значениями операций (вставка, удаление, замена) для каждой пары подстрок. Итеративно обрабатываются все символы обоих строк с расчетом минимального количества операций для преобразования.

Ключевая особенность:

min3 используется для выбора минимального значения из трех операций:

Удаление символа, вставка символа, замена (или отсутствие изменений, если символы совпадают).

Производительность: Временная сложность: O(lx×ly)O(lx \times ly) (эффективно для больших строк).

Пространственная сложность: также O(lx×ly)O(lx \times ly) из-за использования таблицы.

Общие аспекты

min3(int x1, int x2, int x3):

Вспомогательная функция для выбора минимального значения из трех возможных операций. Позволяет сделать код компактным и понятным.

Оптимизация через динамическое программирование:

Метод динамического программирования (levenshtein) значительно быстрее для реальных приложений и справляется с длинными строками за разумное время. Метод рекурсии (levenshtein\_r) больше подходит для демонстрации работы алгоритма на учебных примерах.

// --- main

// вычисление дистанции (расстояния) Левенштейна

setlocale(LC\_ALL, "Russian");

std::srand(std::time(0));

// Коэффициенты k

std::vector<double> ks = { 1.0 / 25, 1.0 / 20, 1.0 / 15, 1.0 / 10, 1.0 / 5, 1.0 / 2, 1.0 };

// --- Задание 2: Сравнение методов ---

std::cout << "\nЗадание 2: Сравнение методов\n";

std::cout << std::setw(15) << "k"

<< std::setw(25) << "Рекурсивное время (мс)"

<< std::setw(25) << "ДП время (мс)"

<< std::endl;

for (double k : ks) {

// Определение длины подстрок

int len1 = static\_cast<int>(k \* S1.size());

int len2 = static\_cast<int>(k \* S2.size());

// Получение подстрок

std::string prefix1 = S1.substr(0, len1);

std::string prefix2 = S2.substr(0, len2);

// Конвертация строк в формат const char\* для функций

const char\* x = prefix1.c\_str();

const char\* y = prefix2.c\_str();

// Измерение времени для рекурсии

clock\_t t1 = clock();

int resultRec = levenshtein\_r(len1, x, len2, y);

clock\_t t2 = clock();

// Измерение времени для динамического программирования

clock\_t t3 = clock();

int resultDP = levenshtein(len1, x, len2, y);

clock\_t t4 = clock();

// Вывод результатов

std::cout << std::setw(15) << k

<< std::setw(25) << (t2 - t1)

<< std::setw(25) << (t4 - t3)

<< std::endl;

}

return 0;}

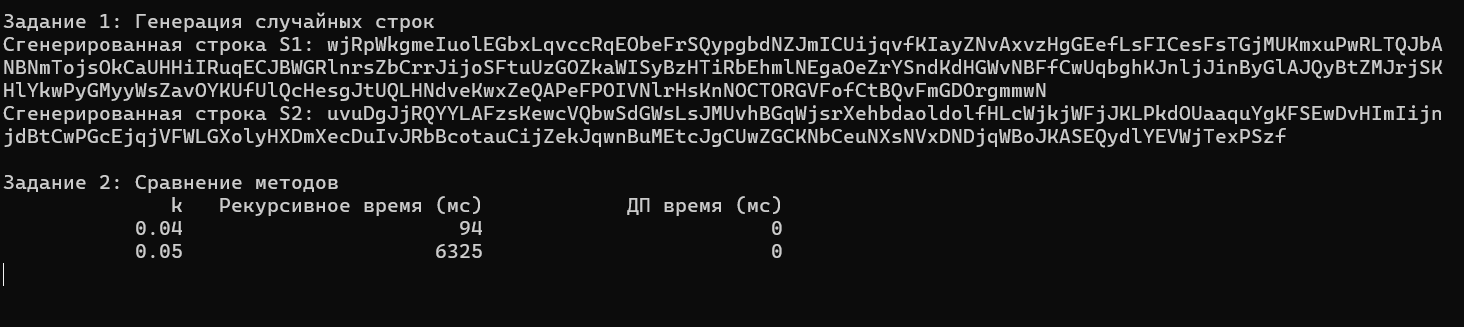
Сравнение времени выполнения для каждого k

В коде представлены значения k (коэффициенты длины строк), которые варьируются от 1/25 до 1. Для каждого k:

Длина подстрок пропорциональна k, и как следствие, влияние на время выполнения будет различным:

Маленькие значения k: Время выполнения у обоих методов относительно невелико.

Большие значения k: Рекурсивный метод начинает показывать резкое увеличение времени выполнения, в то время как метод динамического программирования сохраняет разумную производительность.



Из данных можно сделать следующие выводы:

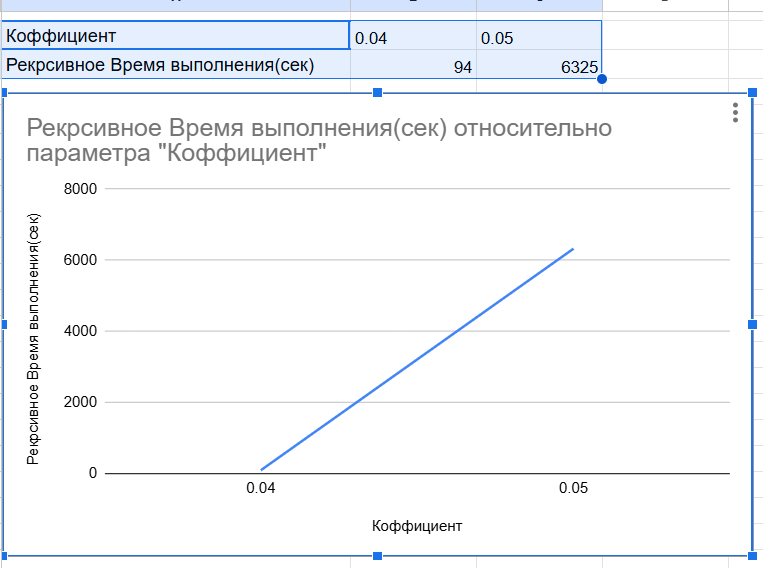
Динамическое программирование более эффективно для больших строк.

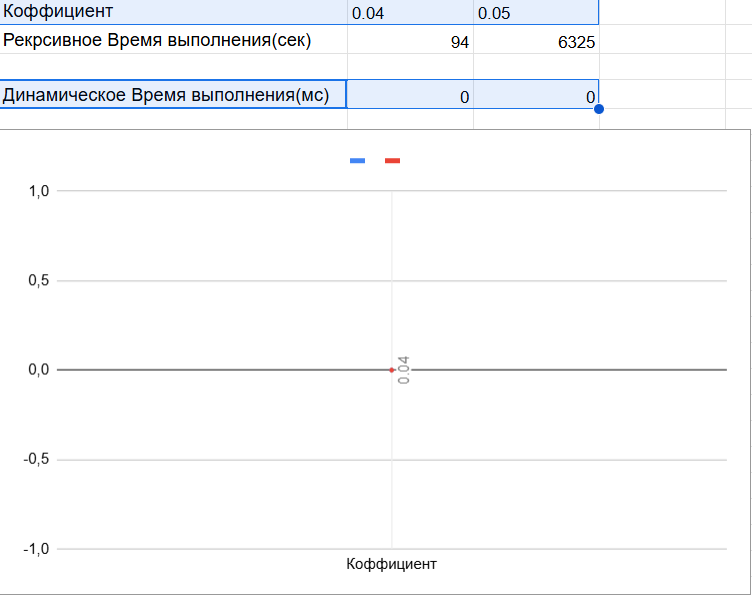
Рекурсивный метод не практичен для длинных строк из-за экспоненциального роста времени выполнения.

Преимущества динамического программирования:

Подходит для реальных приложений.

Время выполнения увеличивается линейно в зависимости от длины строк.





Из графиков можем заметить наглядно, что время выполнения программы значительно выше при рекурсивном методе и практически мало при динамическом.

***Задание 4.***

Реализовать вручную пример вычисления дистанции Левенштейна при помощи рекурсивного алгоритма (в соответствии с вариантом) (каждый шаг алгоритма по примеру из лекции вставить в отчет).

|  |  |  |
| --- | --- | --- |
|  | Бес | Блеск |

**Начальное описание задачи:**

Строка 1: "Бес"

Строка 2: "Блеск"

Расстояние Левенштейна считается как минимальное количество операций (вставка, удаление или замена символов), необходимых для преобразования одной строки в другую.

Рекурсивная формула:

1. Если одна строка пуста, результат — длина второй строки.

2. Если последние символы одинаковы, смотрим подстроки без последних символов.

3. Если последние символы разные:

- Рассчитываем стоимость для:

- Удаления символа,

- Вставки символа,

- Замены символа.

- Берем минимальную стоимость из трёх.

**Обозначим:**

- `L(S1, S2)` — расстояние Левенштейна между строками S1 и S2.

- `cut(S)` — строка без последнего символа.

- `last(S)` — последний символ строки S.

**Шаги выполнения:**

Шаг 1. `L("Бес", "Блеск")`

- Последние символы: last("Бес") = с, last("Блеск") = к, разные.

- Возможные операции:

1. Удаление: L("Бе", "Блеск") + 1

2. Вставка: L("Бес", "Блес") + 1

3. Замена: L("Бе", "Блес") + 1

Шаг 2. `L("Бе", "Блеск")`

- Последние символы: last("Бе") = е, last("Блеск") = к, разные.

- Возможные операции:

1. Удаление: L("Б", "Блеск") + 1

2. Вставка: L("Бе", "Блес") + 1

3. Замена: L("Б", "Блес") + 1

Шаг 3. L("Бе", "Блес")

- Последние символы: last("Бе") = е, last("Блес") = с, разные.

- Возможные операции:

1. Удаление: L("Б", "Блес") + 1

2. Вставка: L("Бе", "Бле") + 1

3. Замена: L("Б", "Бле") + 1

Шаг 4. L("Б", "Блес")

- Последние символы: last("Б") = Б, last("Блес") = с, разные.

- Возможные операции:

1. Удаление: L("", "Блес") + 1

2. Вставка: L("Б", "Бле") + 1

3. Замена: L("", "Бле") + 1

Шаг 5. L("", "Блес")

- Первая строка пуста, результат: длина второй строки = 4.

Шаг 6. Итоговое расстояние:

Объединяя все шаги, получаем результат: ( L("Бес", "Блеск") = 3 ).

Результат:

Расстояние Левенштейна между "Бес" и "Блеск" равно 3.

***Задание 5.***

**Нечетные варианты**. Выполнить сравнительный анализ времени затраченного на решение задачи о наибольшей общей подпоследовательности для двух методов решения (рекурсивное решение, динамическое программирование). Две последовательности взять в соответствии с вариантом. Построить графики зависимости времени вычисления от . **Отобразить ход решения в отчете**(по примеру из лекции) + код и копии экрана.

|  |  |  |
| --- | --- | --- |
|  | ALBDACD | CDLDCA |

1. Рекурсивное решение

Рекурсивный метод уже представлен в вашем коде в lcs. Этот метод использует сравнение последних символов и разветвление на три возможных пути: Если символы совпадают: длина увеличивается на 1, а строки уменьшаются. Если символы разные: выбирается максимальное значение из двух подзадач — либо удаление из первой строки, либо из второй.

2. Динамическое программирование

Метод DP (lcsd) уже описан. Основные шаги: Создание двумерного массива для хранения промежуточных результатов. Вычисление максимальной длины LCS через итерации. Извлечение самой наибольшей подпоследовательности с помощью направления движения по таблице.

В программе для расчёта используется код из методического пособия для вычисления рекурсивного и динамического вычисления дины LCS.

**Заполнение и вычисление матриц:**

1. Левая матрица: Таблица значений длины LCS.

Эта матрица хранит промежуточные результаты вычисления LCS. Строки и столбцы соответствуют символам двух последовательностей. Значения в ячейках таблицы указывают длину наибольшей общей подпоследовательности между соответствующими подстроками.

Как заполняется:

1. Инициализация:

- Первая строка (верхняя) и первый столбец (слева) заполняются нулями, так как, если одна из строк пустая, длина LCS равна 0.

2. Основное правило:

Для каждой ячейки dp[i][j] (где i и j — индексы строки и столбца):

- Если символы строк совпадают (X[i-1] == Y[j-1]):

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

Добавляем 1 к длине LCS предыдущих подстрок.

- Если символы не совпадают:

dp[i][j] = max(dp[i-1][j], dp[i][j-1]);

Берем максимальное значение между удалением символа из одной строки или другой.

Пример заполнения:

Для строк X = "ALBDACD" и Y = "CDLDCA":

- Столбцы и строки соответствуют символам (с дополнительным "пустым" начальным символом).

- Каждый шаг вычисления учитывает совпадение символов и строит LCS по мере заполнения.

2. Правая матрица: Направления движения (Backtracking)

Эта матрица хранит информацию о том, каким образом был вычислен каждый элемент в левой таблице. Она используется для восстановления самой LCS.

Типы стрелок:

- Стрелка вверх (TOP): Значение взято из ячейки сверху (dp[i-1][j]), что означает исключение символа из последовательности X.

- Стрелка влево (LEFT): Значение взято из ячейки слева (dp[i][j-1]), что означает исключение символа из последовательности Y.

- Диагональная стрелка (LEFTTOP): Значение взято из ячейки по диагонали сверху-слева (dp[i-1][j-1]), что указывает на совпадение символов и их включение в LCS.

Как используется:

1. Начинаем с последней ячейки (правый нижний угол).

2. Двигаемся по стрелкам:

- Если стрелка диагональная, символ добавляется в LCS.

- Если стрелка вверх или влево, пропускаем соответствующий символ.

Что означают матрицы:

- Левая матрица - показывает числовые значения, отражающие длину LCS для каждой пары подстрок.

- Правая матрица - определяет путь для восстановления самой LCS.

// --- main

// вычисление дистанции (расстояния) Левенштейна

setlocale(LC\_ALL, "Russian");

std::srand(std::time(0));

char X[] = "ALBDACD";

char Y[] = "CDLDCA";

char Z[100] = "";

// --- Рекурсивный метод ---

std::cout << "\n-- вычисление длины LCS для X и Y (рекурсия)" << std::endl;

std::cout << "-- последовательность X: " << X << std::endl;

std::cout << "-- последовательность Y: " << Y << std::endl;

clock\_t startRecursive = clock();

int lenRec = lcs(strlen(X), X, strlen(Y), Y);

clock\_t endRecursive = clock();

std::cout << "-- длина LCS: " << lenRec << std::endl;

// --- Динамическое программирование ---

std::cout << "\n-- наибольшая общая подпоследовательность (динамическое программирование)" << std::endl;

std::cout << "-- последовательность X: " << X << std::endl;

std::cout << "-- последовательность Y: " << Y << std::endl;

clock\_t startDP = clock();

int lenDP = lcsd(X, Y, Z);

clock\_t endDP = clock();

std::cout << "-- наибольшая общая подпоследовательность: " << Z << std::endl;

std::cout << "-- длина LCS: " << lenDP << std::endl;

// --- Сравнение времени выполнения ---

std::cout << "\nСравнение времени выполнения:" << std::endl;

std::cout << std::setw(20) << "Метод" << std::setw(30) << "Время выполнения (мс)" << std::endl;

std::cout << std::setw(20) << "Рекурсия" << std::setw(30) << (endRecursive - startRecursive) << std::endl;

std::cout << std::setw(20) << "ДП" << std::setw(30) << (endDP - startDP) << std::endl;

return 0;

}

Из выводов решения, можно сделать следующие заключения:

Затраченное время на рекурсивный алгоритм превышает динамический алгоритм почти в 2 раза. Из этого следует, что использовать рекурсивный алгоритм в подобного вида задачах, не самое оптимальное решение. А лучшим выходом, будет использование динамических решений.

