Учреждение образования

«Белорусский государственный технологический университет»

Факультет информационных технологий

Лабораторная работа 4

Динамическое программирование

Выполнил:

Студент 2 курса 7 группы

Колядко Яна Дмитриевна

Минск 2022

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

Задание 1.

На языке С++ сгенерировать случайным образом строку букв латинского алфавита длиной символов и длиной.

Task1.cpp

char\* getString(int size)

{

srand(time(NULL));

char\* str = new char[size]{};

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

{

str[i] = 65 + rand() % 26;

}

return str;

}

Source.cpp

char\* str250 = getString(250);

char\* str300 = getString(300);

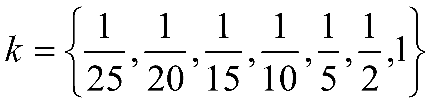
cout << "Строка длиной 250 символов = " << str250 << endl << endl;

cout << "Строка длиной 300 символов = " << str300 << endl << endl;

**Изображение выглядит как текст

Автоматически созданное описание**

Задание 1.

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).

Task2.cpp

#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;

};

Source.cpp

clock\_t t1 = 0, t2 = 0, t3, t4;

char x[] = "abcdefghklmnoxm", y[] = "xyabcdefghomnkm";

int lx = sizeof(x) - 1, ly = sizeof(y) - 1;

std::cout << std::endl << "-- расстояние Левенштейна -----" << std::endl;

std::cout << std::endl << "--длина --- рекурсия -- дин.програм. ---" << std::endl;

for (int i = 8; i < std::min(lx, ly); i++)

{

t1 = clock();

levenshtein\_r(i, x, i - 2, y);

t2 = clock();

t3 = clock();

levenshtein(i, x, i - 2, y);

t4 = clock();

std::cout << std::right << std::setw(2) << i - 2 << "/" << std::setw(2) << i

<< " " << std::left << std::setw(10) << ((double)(t2 - t1)) / ((double)CLOCKS\_PER\_SEC)

<< " " << std::setw(10) << ((double)(t4 - t3)) / ((double)CLOCKS\_PER\_SEC) << std::endl;

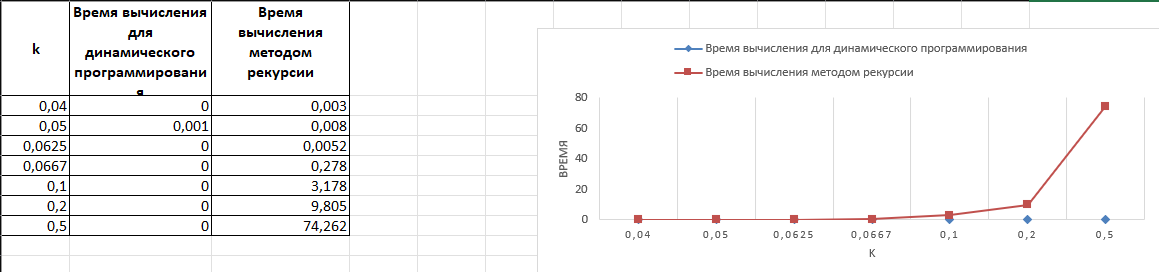
}

**Изображение выглядит как стол

Автоматически созданное описание**

**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).

****

Как видно из задания номер 2 и графиков, при больших значениях k, а соответственно, при небольшой длине строк, метод динамического программирования является выигрышным вариантом по сравнению с методом рекурсии. Это происходит по той причине, что в методе ДП мы должны рассмотреть полиноминальное количество вариантов, пока не найдем решение, а в методе рекурсии перебор является экспоненциальным.

**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).

Вариант 6.

|  |  |
| --- | --- |
| Сом | Домик |

L(“сом”, “домик”)

1. L (“сом”, “домик”) =
2. L (“со”, “домик”) =
3. L (“сом”, “доми”) =
4. L (“со”, “доми”) =
5. L (“с”, “домик”) =

L (“”, “домик”)=5, L (“”, “доми”)=4

1. L (“с”, “доми”) =

L (“”, “доми”)=4, L (“”, “дом”)=3

1. L (“сом”, “дом”) =
2. L (“со”, “дом”) =
3. L (“сом”, “до”) =
4. L (“сом”, “д”) =

L (“сом”, “”)=3, L (“со”, “”)=2

1. L (“с”, “дом”) =

L (“”, “дом”)=3, L (“”, “до”)=2

1. L (“со”, “до”) =
2. L (“с”, “до”) =

L (“”, “до”)=2, L (“с”, “д”)=1, L (“”, “д”)=1

1. L (“со”, “д”) =

L (“со”, “”)=2, L (“с”, “”)=1

1. L (“с”, “д”) =

L (“”, “д”)=1, L (“со”, “”)=2, L (“с”, “”)=1

1. L (“с”, “д”)=min(2, 2, 1)=1
2. L (“со”, “д”)=min(2, 3, 2 )=2
3. L (“с”, “до”)=min(3, 2, 2)=2
4. L (“со”, “до”)=min(3, 3, 1)=1
5. L (“с”, “дом”)=min(4, 3, 3)=3
6. L (“сом”, “д”)=min(3, 4, 3)=3
7. L (“сом”, “до”)=min(2, 4, 3)=2
8. L (“со”, “дом”)=min(4, 2, 3)=2
9. L (“сом”, “дом”)=min(3, 3, 1)=1
10. L (“с”, “доми”)=min(5, 4, 4)=4
11. L (“с”, “домик”)=min(6, 5, 5)=5
12. L (“со”, “доми”)=min(5, 3, 4)=3
13. L (“сом”, “доми”)=min(4, 2, 3)=2
14. L (“со”, “домик”)=min(6, 4, 5)=4
15. L (“сом”, “домик”)=min(5, 3, 4)=3

Таким образом дистанция Левенштейна для данных строк равна 3.

Изображение выглядит как текст

Автоматически созданное описание

**Задание 5.**

**Четные варианты**. Выполнить сравнительный анализ времени, затраченного на решение задачи об оптимальной расстановке скобок при умножении нескольких матриц для двух методов решения (рекурсивное решение, динамическое программирование). Размерность матриц взять в соответствии с вариантом.

int Mc[N + 1] = { 9, 12, 20, 23, 30, 40, 51 }, Ms[N][N], r = 0, rd = 0;

memset(Ms, 0, sizeof(int) \* N \* N);

r = OptimalM(1, N, N, Mc, OPTIMALM\_PARM(Ms));

std::cout << std::endl;

std::cout << std::endl << "-- расстановка скобок (рекурсивное решение) "

<< std::endl;

std::cout << std::endl << "размерности матриц: ";

for (int i = 1; i <= N; i++) std::cout << "(" << Mc[i - 1] << "," << Mc[i] << ") ";

std::cout << std::endl << "минимальное количество операций умножения: " << r;

std::cout << std::endl << std::endl << "матрица S" << std::endl;

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

{

std::cout << std::endl;

for (int j = 0; j < N; j++) std::cout << Ms[i][j] << " ";

}

std::cout << std::endl;

memset(Ms, 0, sizeof(int) \* N \* N);

rd = OptimalMD(N, Mc, OPTIMALM\_PARM(Ms));

std::cout << std::endl

<< "-- расстановка скобок (динамичеое программирование) " << std::endl;

std::cout << std::endl << "размерности матриц: ";

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

std::cout << "(" << Mc[i - 1] << "," << Mc[i] << ") ";

std::cout << std::endl << "минимальное количество операций умножения: "

<< rd;

std::cout << std::endl << std::endl << "матрица S" << std::endl;

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

{

std::cout << std::endl;

for (int j = 0; j < N; j++) std::cout << Ms[i][j] << " ";

}

std::cout << std::endl << std::endl;

Изображение выглядит как текст

Автоматически созданное описание

Принцип расстановки скобок по итоговой матрице:

Скобки расставляются по принципу «сначала внешние – затем внутренние». Имеется 6 матриц.

Найдем элемент (1,6) в матрице S, он равен 5. Это означает, что точка разрыва между 1-ой и 6-ой матрицей находится после 5-ой матрицы. Что позволяет расставить скобки следующим образом:

(A1\*A2\*A3\*A4\*A5)\*A6

Точку разрыва между первой и пятой матрицей определяет элемент (1,5). Он равен 4. Следовательно разрыв будет после четвертой матрицы.

((A1\*A2\*A3\*A4)\*A5)\*A6

Далее берем элемент (1,4) и получаем, что он равен 3. Следовательно получаем:

(((A1\*A2\*A3)\*A4)\*A5)\*A6

И на последнем шаге мы возьмем элемент (1,3), и он равен 2:

(((A1\*A2)\*A3)\*A4)\*A5)\*A6

Это выражение и есть конечное.

Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 41670.

**Вывод:** в данной лабораторной работе были освоены общие принципы решения задач методом динамического программирования, сравнены полученные решения задач с рекурсивным методом.