Учреждение образования

«Белорусский государственный технологический университет»

**Лабораторная работа №4**

**Динамическое программирование**

Выполнила:

Студент 2 курса 7 группы ФИТ

Кореневский К.Р.

**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 300 символов и длиной 250. Ниже на рисунке 1 представлен код генерации строк и на рисунке 2 результат.

#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

int main()

{

setlocale(LC\_ALL, "rus");

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

char x[301], y[251];

srand(time(0));

for (int i = 0; i < 300; i++)

{

x[i] = ('a' + rand() % ('z' - 'a'));

std::cout << x[i];

}

std::cout << std::endl;

for (int i = 0; i < 250; i++)

{

y[i] = ('a' + rand() % ('z' - 'a'));

std::cout << y[i];

}

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

std::cout << std::endl;

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

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

for (int i = 0; i < 250; i++)

{

if (i < 13) { t1 = clock(); levenshtein\_r(i, x, i, y); t2 = clock(); }

t3 = clock(); levenshtein(i, x, i, y); t4 = clock();

std::cout << std::right << std::setw(2) << i << "/" << std::setw(3) << i

<< " ";

if (i < 13) std::cout << std::left << std::setw(10) << (t2 - t1);

else std::cout << std::left << std::setw(10) << "null";

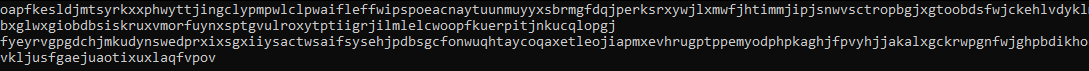
std::cout << " " << std::setw(10) << (t4 - t3) << std::endl;

}

system("pause");

return 0;

}



**Задание 2**. Вычислить двумя способами (рекурсивно и с помощью динамического программирования) – дистанцию Левенштейна для , где - длина строки - строка состоящая из первых символов строки.

#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

using namespace std;

int main()

{

setlocale(LC\_ALL, "rus");

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

char x[25] = "abcdefghdd", y[] = "xyabcdeshrtrh";

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

std::cout << std::endl;

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

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

<< std::endl;

int u = 1;

for (int i = 6; 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();

cout << right << setw(2) << 1 << "/" << setw(2) << u << " " << left << setw(10) << (t2 - t1) << " " << setw(10) << (t4 - t3) << endl;

if (u > 0) {

if (u == 20) { u = u + 5; }

if (u == 15) { u = u + 5; }

if (u == 10) { u = u + 5; }

if (u == 5) { u = u + 5; }

if (u == 2) { u = u + 3; }

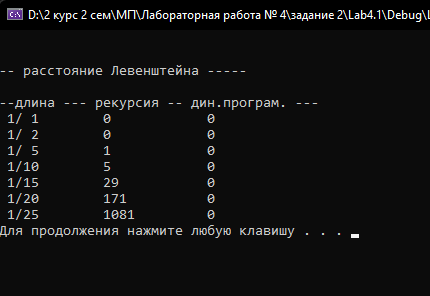
if (u == 1) { u = u + 1; }

}

}

system("pause");

return 0; }



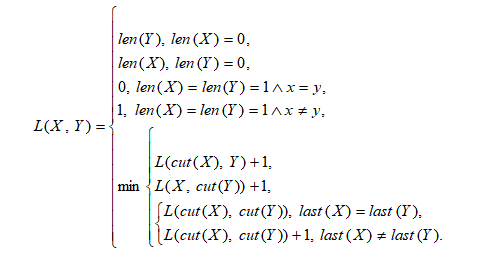
**Задание 3**. Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от k.

На графике, представленном на рисунке 6 можно заметить, что выполнение с помощью динамического алгоритма, вычисления производятся в разы быстрее, чем с помощью рекурсивного алгоритма.

**Задание 4.** Реализовать вручную пример вычисления дистанции Левен-штейна при помощи рекурсивного алгоритма (в соответствии с вариантом)

Рекурсивный алгоритм – это алгоритм, решающий задачу путем сведения ее к решению одной или нескольких таких же задач, но в сокращенном их варианте.

Дистанция Левенштейна (расстояние Левенштейна, редакционное расстояние, дистанция редактирования) определяется между двумя строками и равна минимальному количеству операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.





=5

*=4*



*=4*

*=3*

=3

*=2*

=2

*=1*

=3

*=2*

=2

*=1*

=1

*=1*



#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

int main()

{

setlocale(LC\_ALL, "rus");

char X[] = "кол", Y[] = "столб";

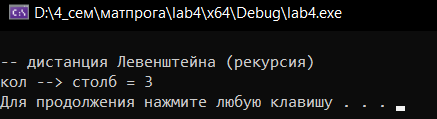
std::cout << std::endl << "-- дистанция Левенштейна (рекурсия)";

std::cout << std::endl << X << " --> " << Y << " = "

<< levenshtein\_r(sizeof(X) - 1, X, sizeof(Y) - 1, Y) << std::endl;

system("pause");

return 0; }

**

**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.

QVTWNHO

RQTWYK

1. С7,6 = max(c7,5 , c6,6);
2. С7,5 = max(c7,4 , c6,5);
3. С6,6 = max(c6,5 , c5,6);
4. С7,4 = max(c7,3 , c6,4);
5. С6,5 = max(c6,4 , c5,5);
6. С5,6 = max(c5,5 , c4,6);
7. С7,3 = max(c7,2 , c6,3);
8. С6,4 = max(c6,3 , c5,4);
9. С5,5 = max(c5,4 , c4,5);
10. С4,6 = max(C4,5 , c3,6);
11. С7,2 = max(c7,1 , c6,2);
12. С6,3 = max(c6,2 , c5,3);
13. С5,4 = max(c5,3 , c4,4);
14. С4,5 = max(c4,4 , c3,5);
15. С3,6 = max(C3,5 , c2,6);
16. С7,1 = max(c7,0 , c6,1) = C6,1;
17. С6,2 = max(c6,1 , c5,2);
18. С5,3 = max(c5,2 , c4,3);
19. С4,4 = C3,3 + 1;
20. С3,5 = max(c3,4 , c2,5);
21. С2,6 = max(C2,5 , c1,6);
22. С6,1 = max(C6,0 , c5,1) = C5,1;
23. С5,2 = max(c5,1 , c4,2);
24. С4,3 = max(c4,2 , c3,3);
25. С3,3 = C2,2 + 1;
26. С3,4 = max(c3,3 , c2,4);
27. С2,5 = max(c2,4 , c1,5);
28. С1,6 = max(c1,5 , c0,6) = C1,5;
29. С5,1 = max(C5,0 , c4,1) = C4,1;
30. С4,2 = max(C4,1 , c3,2);
31. С3,3 = max(C3,2 , c2,3) ;
32. С2,2 = max(C2,1 , c1,2) ;
33. С2,4 = max(C2,3 , c1,4) ;
34. С1,5 = max(C1,4 , c0,5) = C1,4;
35. С4,1 = max(C4,0 , c3,1) = C3,1;
36. С3,2 = max(C3,1 , c2,2) ;
37. С2,3 = max(C2,2 , c1,3) ;
38. С2,1 = max(C2,0 , c1,1) = C1,1;
39. С1,4 = max(C1,3 , c0,4) = C1,3;
40. С3,1 = max(C3,0 , c2,1) = C2,1;
41. С1,3 = max(C1,2 , c0,3) = C1,2;
42. С1,1 = max(C1,0 , c0,1);
43. С1,2 = C0,1 + 1;
44. С2,2 = max(1, 1) ;
45. С3,2 = max(1, 1) ;
46. С3,3 = C2,2 + 1 = 2;
47. С4,2 = max(2, 1) ;
48. С4,3 = max(1, 2) ;
49. С4,4 = C3,3 + 1 = 3;

С7,6 = C4,4 = 3;

Выполнив сравнительный анализ времени затраченного на решение задачи о наибольшей общей подпоследовательности для двух методов решения, можно заметить, что динамический алгоритм работает быстрее, однако, в данном ситуации оба метода затрачивают для выполнения минимально времени.

#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

int main()

{

setlocale(LC\_ALL, "rus");

char X[] = "ABCDFGI", Y[] = "EATUFI";

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

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

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

int s = lcs(

sizeof(X) - 1, // длина последовательности X

"ABCDFGI", // последовательность X

sizeof(Y) - 1, // длина последовательности Y

"EATUFI" // последовательность Y

);

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

//DYNAMIC

char z[100] = "";

char x[] = "QVTWNHO",

y[] = "RQTWYK";

int l = lcsd(x, y, z);

std::cout << std::endl

<< "-- наибольшая общая подпоследовательость - LCS(динамическое"

<< " программирование)" << std::endl;

std::cout << std::endl << "последовательость X: " << x;

std::cout << std::endl << "последовательость Y: " << y;

std::cout << std::endl << " LCS: " << z;

std::cout << std::endl << " длина LCS: " << l;

std::cout << std::endl;

system("pause");

return 0;

}

На рисунке представлен график зависимости от суммарной длинны двух подстрок и необходимом времени выполнения при помощи динамического алгоритма:

**Алгоритм прохождения:**

1. Создаем двумерный массив размером (len(QVTWNHO) + 1) на (len(RQTWYK) + 1), где каждый элемент равен 0.

2. Проходим по каждой строке и столбцу массива, начиная с первого и до последнего.

3. Если значение строки или столбца равно 0, то элемент массива на пересечении этой строки и столбца остается равным 0.

4. Если символ строки и столбца равны, то элемент массива на пересечении этой строки и столбца равен значению элемента на предыдущей диагонали плюс 1, т.е. с\_(𝑛,𝑝) = c\_(𝑛−1,𝑝−1) + 1.

5. Если символ строки и столбца не равны, то элемент массива на пересечении этой строки и столбца равен максимуму между значением элемента на предыдущей строке и значением элемента на предыдущем столбце, т.е. c\_(𝑛,𝑝) = max( с\_(𝑛,𝑝"−1" ), c\_(𝑛−1,𝑝)).

6. После прохода по всем элементам массива, наибольшая общая подпоследовательность будет равна значению элемента на пересечении последней строки и последнего столбца.

**Вывод:** динамический подход к решению задач позволяет выполнять их значительно быстрее, чем рекурсивный, особенно это будет заметно при решении задач с большим объёмом информации, вызвано это тем, что данные кэшируются, а не вычисляются каждый раз заново. Так же, в результате лабораторной работы я сделал следующие выводы:

1. Динамическое программирование - это мощный метод решения задач, позволяющий эффективно решать широкий спектр задач, которые не могут быть решены простыми алгоритмами.

2. Метод динамического программирования заключается в разбиении сложной задачи на более простые подзадачи, решение которых затем комбинируется в общее решение задачи.

3. Решение задач методом динамического программирования отличается высокой скоростью выполнения благодаря использованию кэширования вычислений и быстрой обработке данных.

4. В ходе выполнения лабораторной работы был исследован один из наиболее популярных методов динамического программирования - расстояние Левенштейна. Результаты экспериментов показали, что данная техника может быть очень эффективна для решения задач, связанных с обработкой текстовых данных.