**Лабораторная работа 4**

**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

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

Как уже отмечалось, в некоторых случаях рекурсивное решение может оказаться очень трудоемким, так как в процессе его выполнения одна и та же задача может решаться несколько раз. Простой анализ примера вычисления дистанции Левенштейна позволяет выявить такие повторные вычисления. Например, функция  вычислялась в шагах 7, 8, 7, 16 алгоритма.

Если в процессе выполнения рекурсивного алгоритма одна и та же задача решается несколько раз, то говорят, что алгоритм содержит ***перекрывающиеся подзадачи***. Например, задача вычисления функции , рассмотренная выше, является перекрывающейся подзадачей задачи вычисления функции 

Метод решения задачи оптимизации, реализующей рекурсивный алгоритм с перекрывающимися подзадачами, в котором каждая такая подзадача решается один раз, а ее результат сохраняется для последующего применения, называется ***динамическим программированием***

**Задание 1.** На языке С++ сгенерировать случайным образом строку букв латинского алфавита  длиной  символов и длиной .

Генератор представляет собой функцию getString с 1 параметром, в котором указывается размер строки. Функция возвращает указатель на начало этой строки в куче.

char\* getString(int size)

{

srand(time(NULL));

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

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

{

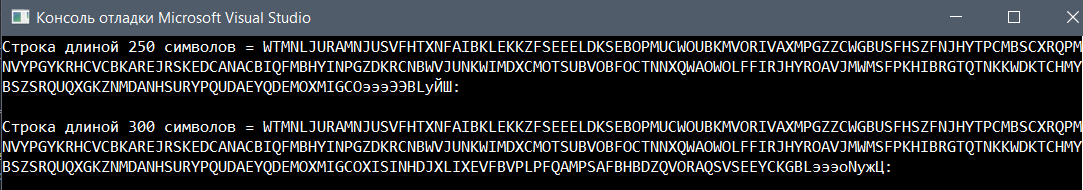
str[i] = 65 + rand() % 26;

}

return str;

}

Результат:



**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).

Дистанция Левенштейна — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций, необходимых для превращения одной последовательности символов в другую.

Рекурсивно данное вычисление производится с помощью функции

levenshtein\_r, которая принимает как параметры две строки х и у, а также их длины lx и ly соответственно.

Динамически данное вычисление производится с помощью функции

levenshtein, которая принимает как параметры две строки х и у, а также их длины lx и ly соответственно.

Обе функции возвращают количество правок, которые нужно сделать, для эквивалентности строк.

Результат:



**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).

Метод динамического программирования значительно эффективнее рекурсивного метода, т.к. выполняется намного быстрее.

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).

Найти расстояние Левенштейна между словами «Акр» и «Якорь».



1. L(“Акр”, “Якорь”) = min

2. L(“Ак”, “ Якорь”) = min

3. L(“Акр”, “Якор”) = min

4. L(“Ак”, “Якор”) = min

5. L(“А”, “Якорь”) = min

L(“”,” Якорь ”) = 5

L(“”,”Якор”)=4

6. L(“А”, “Якор”) = min

L(“”,”Якор”) = 4

L(“”,”Яко”)=3

7. L(“Акр”, “Яко”) = min

8. L(“Ак”, “Яко”) = min

9. L(“А”, “Яко”) = min

L(“”,”Яко”) = 3

L(“”,”Як”) = 2

10. L(“Акр”, “Як”) = min

11. L(“Ак”, “Як”) = min

12. L(“А”, “Як”) = min

L(“”,”Як”) = 2

L(“”,”Я”) = 1

13. L(“Акр”, “Я”) = min

L(“Акр”,””) = 3

L(“Ак”,””) = 2

14. L(“Ак”, “Я”) = min

L(“Ак”,””) = 2

L(“А”,””) = 1

15. L(“А”, “Я”) = min

L(“”,”Я”) = 1

L(“А”,””) =1

L(“”,””) = 0

16 L(“А”, ”Я”) = min(2,2,1) = 1

17 L(“Ак”, ”Я”) = min (2, 3, 2) = 2

18 L(“Акр”, ”Я”) = min (3, 4, 3) = 3

19 L(“А”, “Як”) = min (3, 2, 2) = 2

20 L(“Ак”, ”Як”) = min (3, 3, 1) = 1

21 L(“Акр”, ”Як”) = min (2, 4, 3) = 2

22 L(“А”, ”Яко”) = min (4, 3, 3) = 3

23 L(“Ак”, “Яко”) = min (4, 2, 3) = 2

24 L(“Акр”, “Яко”) = min (3, 3, 1) = 1

25 L(“А”, ”Якор”) = min(5, 4, 4) = 4

26 L(“А”, “ Якорь”) = min (6, 5, 5) = 5

27 L(“Акр”, “Яко”) = min (4, 2, 3) = 2

28 L(“Ак”, “Якор”) = min (3, 3, 4) = 3

29 L(“Акр”, “Якор”) = min (4, 2, 3) = 2

30 L(“Ак”, “ Якорь”) = min (4, 4, 3) = 3

31 L (“Акр”, “ Якорь”) = min (3, 4, 3) = 3

**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.

Принцип расстановки скобок по итоговой матрице:

Скобки расставляются по принципу «сначала внешние – затем внутренние». Имеется 6 матриц, вот их размерность:

А1=20,

А2=15,

А3=30,

А 4 =53,

А 5 =10,

А 6 =20.

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

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

Точку разрыва между первой и пятой матрицей определяет элемент (2,4). Он равен 4. Следовательно разрыв будет после четвертой матрицы.

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

Далее берем элемент (2,4) и получаем, что он равен 2. Следовательно получаем:

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

Это выражение и есть конечное.

Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 27550.

