**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++)

{

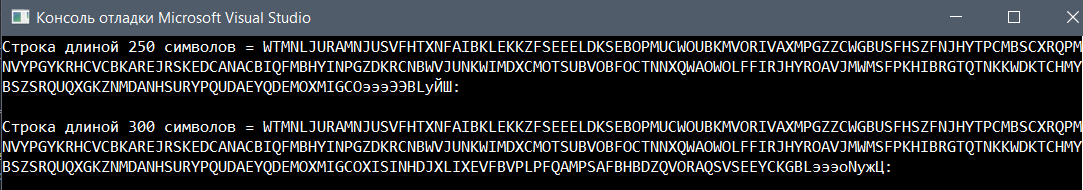
str[i] = 65 + rand() % 26;

}

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).

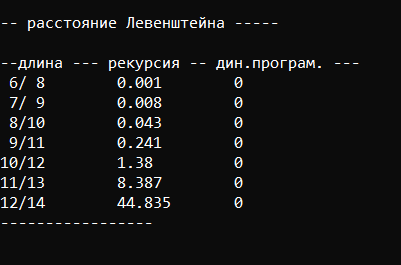
Дистанция Левенштейна — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций, необходимых для превращения одной последовательности символов в другую.

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 динамического программирования 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).

Задание варианта:

|  |  |  |
| --- | --- | --- |
| Вариант | Задание 4 | |
| 9 | Вол | Колун |

Найти расстояние Левенштейна между словами «Вол» и «Колун».

1. L(“вол”, “колун”) =   
    min(L(“во”, “колун”) + 1,   
    L(“вол”, “колу”) + 1,   
    L(“во”, “колу”) + 1) = min(5, 3, 4) [этап 1];  
     
   1 этап
2. L(“во”, “ колун”) =   
    min(L(“в”, “колун”) + 1,   
    L(“во”, “колу”) + 1,   
    L(“в”, “колу”) + 1) = min(5, 4, 5) = 4 [этап 2];
3. L(“вол”, “колу”) =   
    min(L(“во”, “колу”) + 1,   
    L(“вол”, “кол”) + 1,   
    L(“во”, “кол”) + 1) = min(4, 2, 3) = 2 [этап 3];
4. L(“во”, “колу”) =   
    min(L(“в”, “колу”) + 1,   
    L(“во”, “кол”) + 1,   
    L(“в”, “кол”) + 1) = min(5, 3, 4) =3 [этап 4];  
     
   2 этап
5. L(“в”, “колун”) =   
    min(L(“”, “колун”) + 1,   
    L(“в”, “колу”) + 1,   
    L(“”, “колу”) + 1);  
   min(6, 5 , 5) = 5;
6. L(“в”, “колу”) =   
    min(L(“”, “колу”) + 1,   
    L(“в”, “кол”) + 1,   
    L(“”, “кол”) + 1);  
   min(5, 4 , 4) = 4;
7. L(“в”, “кол”) =   
    min(L(“”, “кол”) + 1,   
    L(“в”, “ко”) + 1,   
    L(“”, “ко”) + 1);  
   min(4, 3 , 3) = 3;
8. L(“в”, “ко”) =   
    min(L(“”, “ко”) + 1,   
    L(“в”, “к”) + 1,   
    L(“”, “к”) + 1);  
   min(3, 2 , 2) = 2;
9. L(“в”, “к”) =   
    min(L(“”, “к”) + 1,   
    L(“в”, “”) + 1,   
    L(“”, “”) + 1);  
   min(2, 2, 1) = 1;  
     
   3 этап
10. L(“во”, “колу”) =   
     min(L(“в”, “колу”) + 1,   
     L(“во”, “кол”) + 1,   
     L(“в”, “ко”) + 1) = (4, 3, 3) = 3;
11. L(“во”, “кол”) =   
     min(L(“в”, “кол”) + 1,   
     L(“во”, “ко”) + 1,   
     L(“в”, “ко”) + 1);  
    min(3, 2 , 2) = 2;
12. L(“во”, “ко”) =   
     min(L(“в”, “ко”) + 1,   
     L(“во”, “к”) + 1,   
     L(“в”, “к”) );  
    min(2, 3 , 1) = 1;
13. L(“во”, “к”) =   
     min(L(“в”, “к”) + 1,   
     L(“во”, “”) + 1,   
     L(“в”, “”) + 1 );  
    min(2, 3 , 2) = 2;  
      
    4 этап
14. L(“вол”, “кол”) =   
     min(L(“во”, “кол”) + 1,   
     L(“вол”, “ко”) + 1,   
     L(“во”, “ко”) ) = min(4, 3, 1) = 1;  
      
    Результат:
15. Дистанция Левенштейна для слов «Вол» и «Колун»: 3.

**Задание 5.**

**Нечетные варианты (9)**. Две последовательности взять в соответствии с вариантом. **Отобразить ход решения в отчете**.

Задача на вычисление наибольшей общей подпоследовательности.

Заполнение матрицы 1:

1. Первые строка и столбец матрицы заполняются нулями.
2. Каждый элемент последовательности так, что если символы равны для позиции (i, j), то мы прибавляем 1. Иначе вычисляем максимум для соседей слева и сверху.
3. Элемент в правом нижнем углу показывает длину наибольшей общей подпоследовательности.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | K | I | B | Y | S | V |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| H | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| D | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| S | 0 | 0 | 0 | 1 | 1 | 2 | 2 |
| A | 0 | 0 | 0 | 1 | 1 | 2 | 2 |
| V | 0 | 0 | 0 | 1 | 1 | 2 | 3 |

Заполнение матрицы 2:

1. Если символы для позиции (i, j) совпадают, то стрелка указывает в левый верхний угол.
2. Если значение от соседа слева больше, чем от соседа сверху, то стрелка указывает влево.
3. Остальные все стрелки указывают вверх.
4. Последовательность получается ходом стрелок с правого нижнего угла путём добавления символов в последовательность при наличии стрелки влево-вверх во время добавления.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | K | I | B | Y | S | V |
|  |  |  |  |  |  |  |  |
| T |  | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| B |  | ↑ | ↑ | \ | ← | ← | ← |
| H |  | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| D |  | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| S |  | ↑ | ↑ | ↑ | ↑ | \ | ← |
| A |  | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| V |  | ↑ | ↑ | ↑ | ↑ | ↑ | \ |