**Лекция 5**

**Рекурсивные алгоритмы**

Многие оптимизационные алгоритмы основаны на принципе разбиения основной задачи на подзадачи, каждая из которых повторяет основную, но входные их данные таковы, что область допустимых решений становится меньше.

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

Неразрывно с понятием рекурсивного алгоритма связано понятие рекурсивной функции. Существует два определения этого понятия.

Первое определение рекурсивной функции относится к теории вычислимости и является синонимом понятия вычислимой функции, т. е. функции, для вычисления значения которой можно указать алгоритм.

Второе определение, которое и будет использоваться здесь, происходит из области теории программирования.

***Рекурсивная функция*** – это функция, которая вызывает саму себя.

Рекурсивный алгоритм может быть записан в виде рекурсивной функции. Классическими примерами рекурсивных функций являются функции для вычисления факториала, чисел Фибоначчи и наибольшего общего делителя с помощью алгоритма Эвклида (рис. 1).

//- Рекурсивные функции

//-- вычисление факториала числа

**unsigned int fac(unsigned int x)** // A000142 в OEIS

**{ return (x == 0)?1:x\*fac(x-1); };**

//--вычисление наибольшего общего делителя

**unsigned int gcd(int m, int n)** // Эвклид (III в до н.э.)

**{ return (n == 0)?m:gcd(n, m%n); };**

// -- вычисление n-го члена ряда Фибоначчи(1170-1250**)**

**unsigned int fib(int n)** // A000045 в OEIS

**{ return (n < 1)?0:((n == 1)?1:fib(n-1)+fib(n-2));};**

1. Примеры простейших рекурсивных функций

Рекурсивную функцию всегда можно преобразовать в цикл, и, наоборот любой цикл можно представить в виде рекурсивной функции. На рис. 2 приведен пример функции, вычисляющей факториал числа с помощью цикла.

//- Вычисление факториала числа

**unsigned int fac(unsigned int x)**

**{**

**int rc = 1;**

**for (int i = 1; i<=x; i++) rc \*= i;**

**return rc;**

**};**

Рис. 2. Нерекурсивная функция вычисления факториала числа

Рекурсивная запись алгоритма, как правило, не дает выигрыша в скорости его работы. Скорее наоборот, так как вызов любой функции связан с сохранением и восстановлением контекста вызывающей функции, что является затратной по времени операцией. Кроме того, для хранения контекста операционной системой резервируется специальная секция памяти, называемая системным стеком. Если цепочка вызовов функций является длинной (иногда говорят о большой ***глубине рекурсии***), то это может привести к переполнению стека. Например, при вычислении факториала числа 25 глубина рекурсии достигает значения 24.

Часто рекурсивные функции, применяемые для решения оптимизационных задач, используют более одного рекурсивного вызова, каждый из которых работает приблизительно с половиной входных данных. Такую схему решения называют ***«разделяй и властвуй»***. На рис 3 представлен пример рекурсивной функции поиска максимального элемента в массиве, которая использует эту схема.

//- Поиск максимального элемента в массиве

**int mmax (** // поиск максимального элемента в диапазоне [s,e]

**const int m[],** // массив

**int s,** // индекс первого элемента

**int e** // индекс последнего элемента

**)**

**{**

**int rc, u, v;**

**if (s == e) rc = m[s];**

**else rc = (u = mmax(m, s, (s+e)/2))>(v = mmax(m, (s+e)/2+1,e))?u:v;**

**return rc;**

**};**

**int mmax (**// поиск максимального элемента в массиве

**const int m[],** // массив

**int n** // количество элементов

**)**

**{ return mmax(m, 0, n-1); };**

Рис. 3. Пример рекурсивной функции, использующей два рекурсивных вызова

**Решение задачи о рюкзаке**

В предыдущих лекциях была приведена упрощенная задача о рюкзаке. Здесь эта задача будет сформулирована в классическом виде и решена с помощью рекурсивного алгоритма.

Классическая формулировка задачи допускает выбор одинаковых предметов при размещении их в рюкзаке. Математическая модель классической задачи о рюкзаке выглядит следующим образом:

где  – неизвестные (количество предметов каждого типа), которые требуется найти;  – объемы предметов; – стоимость единицы объема каждого предмета;  – вместимость рюкзака;.

Решением задачи при такой постановке будет вектор  Каждый элемент  вектора может принимать целое значение из отрезка  При этом, если  то -й предмет не выбран, и если  то -х предметов в рюкзак помещено  единиц.

Обозначим  – отрезок вектора решения  Аналогичным образом введем отрезок  стоимостей. Тогда максимальная стоимость рюкзака  может быть записана в векторной форме: 

Заметим, что  где  – стоимость оптимальной комбинации первых  предметов;  – стоимость оптимальной комбинации последних  предметов. Поэтому можем записать следующую последовательность рекуррентных соотношений, позволяющую вычислить значение 

, ,

, ,

,

, …, , .

Схема решения задачи о рюкзаке с помощью рекурсивного алгоритма при    ,     изображена на рис. 4.



Рис. 4. Схема рекурсивного решения задачи коммивояжера

Все вершины, кроме корневой, дерева, приведенного на схеме, изображают этапы решения. Вершины помечены двумя числами: первое число – текущая стоимость рюкзака, второе – остаток неиспользованного объема рюкзака. Все этапы образуют три слоя, что определяет глубину рекурсии. Каждой вершине (этапу) соответствует вызов рекурсивной функции.

Вершины соединены дугами, указывающими связь между этапами решения. Каждая дуга имеет метку, обозначающую предположение, при котором решается очередной этап.

Для примера, рассмотрим маршрут, образованный дугами с метками   и 

Первый этап в этом маршруте отмечен меткой  означающей, что при размещении в рюкзаке двух предметов с номером 3 стоимость рюкзака станет  единиц, и при этом в рюкзаке останется  единиц объема.

Второй этап имеет метку . Решение на этом этапе осуществляется в предположении, что в рюкзаке два предмета с номером 3 и один предмет с номером 2. Поэтому  и .

На третьем этапе завершается формирование одного из допустимых решений. Неиспользованный остаток объема в  единиц позволяет поместить только один предмет первого типа. В окончательном решении, соответствующем этому маршруту, стоимость предметов, уложенных в рюкзак, равна 

Несложно заметить, что разобранный маршрут не соответствует оптимальному решению. Оптимальным будет решение , а соответствующая ему стоимость –  Этапы этого решения на рис. 7.4 обозначены закрашенными овалами.

Из схемы, приведенной на рис. 4, видно, что в рекурсивном решении, как и в случае с использованием генератора, осуществляется полный перебор допустимых решений.

На рис. 5 и 6 представлена функция **knapsack\_r**, реализующая рекурсивный алгоритм решения задачи о рюкзаке.

// - Knapsack\_r.h

//-- рекурсивный алгоритм решения задачи о рюкзаке

**int knapsack\_r(**

**int V,** // вместимость рюкзака

**int n,** // количество предметов

**int v[],** // вес предметов

**int c[],** // стоимость предметов

**int m[]** //[out]результат

**);**

Рис.5. Прототип функции **knapsack\_r**

Функция **knapsack\_r** имеет четыре входных и один возвращаемый параметры: значение входных параметров задают условие задачи, а последний возвращаемый параметр **m** представляет собой массив размерностью **n** (общее количество предметов, заданное вторым параметром),содержащий количества выбранных предметов каждого типа.

// - Knapsack\_r.cpp

**#include "stdafx.h"**

**#include <memory.h>**

**#include "Knapsack.h"**

**#define NINF 0x80000000**

**int knapsack\_r(int V, int n, int v[], int c[], int m[])**

**{**

**int o = NINF, bo = 0, iv = 0, \*bm = new int[n-1], li = n-1;**

**if (li > 0)**

**for (int i = 0; i\*v[li] <= V;i++)** // пока решение допустимо

**{**

**bo = i\*c[li]\*v[li]+ knapsack\_r(V-i\*v[li], n-1, v, c, bm);**

**if (bo > o)**

**{**

**o = bo;** // максимальная стоимость

**m[li] = i;** // количество предметов

**memcpy(m,bm,li\*sizeof(int));**

**}**

**}**

**else**

**{**

**iv = V/v[li];** // округление

**o = c[li]\* iv\*v[li];** // стоимость

**m[li] = iv;** // количество

**}**

**delete[]bm;**

**return o;**

**}**

Рис. 6. Реализация рекурсивной функции **knapsack\_r**

Условно текст функции **knapsack\_r** можно разбить на два блока, соответствующих двум ветвям оператора **if-else**.

В первом блоке в цикле осуществляется рекурсивный вызов функции **knapsack\_r**. В цикле в порядке возрастания перебираются все допустимые для заданного неиспользованного объема рюкзака количества предметов одного типа. Вызов функции **knapsack\_r** в итерации цикла соответствует разветвлению в узлах графа на рис. 4.

Кроме массива **m**, содержащего количества предметов каждого типа в оптимальной загрузке рюкзака, функция в точку вызова возвращает суммарную стоимость этих предметов.

Во втором блоке обрабатывается так называемое ***дно рекурсии***. Дно рекурсии соответствует последнему вызову функции **knapsack\_r** (последнему этапу решения) в полной ветви графа на рис. 4.

На рис. 7 и 8 приведен пример вызова функции **knapsack\_r** для решения задачи о рюкзаке с такими же исходными данными, что для схемы, представленной на рис. 4.

// - Мain

// -- рекурсивный алгоритм решения задачи о рюказке

**#include "stdafx.h"**

**#include <iostream>**

**#include "Knapsack.h"**

**#define NN 3**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**int V = 120,** // вместимость рюкзака

**v[3] = {30, 50, 20},** // размер предмета каждого типа

**c[3] = {5, 7, 3};** // стоимость предмета каждого типа

**int m[3];** // количество предметов каждого типа

**int maxcc = knapsack\_r(**

**V,** // [in] вместимость рюкзака

**NN,** // [in] количество типов предметов

**v,** // [in] размер предмета каждого типа

**c,** // [in] стоимость предмета каждого типа

**m** //[out] количество предметов каждого типа

**);**

**std::cout<<std::endl<<"-------- Задача о рюкзаке --------- ";**

**std::cout<<std::endl<<"- количество предметов : "<< NN;**

**std::cout<<std::endl<<"- вместимость рюкзака : "<< V;**

**std::cout<<std::endl<<"- размеры предметов : ";**

**for(int i = 0; i < NN; i++) std::cout<<v[i]<<" ";**

**std::cout<<std::endl<<"- стоимости предметов : ";**

**for(int i = 0; i < NN; i++) std::cout<<v[i]\*c[i]<<" ";**

**std::cout<<std::endl<<"- оптимальная стоимость рюкзака: " << maxcc;**

**std::cout<<std::endl<<"- вес рюкзака: ";**

**int s = 0; for(int i = 0; i < NN; i++) s+= m[i]\*v[i]; std::cout<<s;**

**std::cout<<std::endl<<"- количество выбранных предметов: ";**

**for(int i = 0; i < NN; i++) std::cout<<" "<<m[i];**

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

**system("pause");**

**return 0;**

**}**

Рис. 7.Пример вызова функции **knapsack\_r**

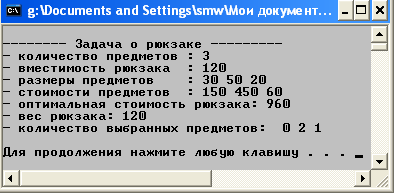


Рис. 8. Результат выполнения программы, представленной на рис. 7

**Решение задачи о расстановке скобок при перемножении матриц**

Вычислим количество операций умножения, которые необходимо выполнить при перемножении трех матриц   и  размерностью соответственно   и 

Заметим, что операция умножения матриц ассоциативна и следовательно, 

Несложно подсчитать, что количество операций умножения, которые необходимо выполнить при вычислении произведения  будет  а при вычислении  –  т.е. в 10 раз больше. Таким образом, расстановка скобок при перемножении большого количества матриц может существенно повлиять на трудоемкость вычисления результата.

Сформулируем задачу в общем виде.

Рассмотрим произведение  матиц    …,  имеющих размерности соответственно    …, 

Обозначим  – минимальное возможное количество операций умножения при вычислении произведения    Очевидно, что  Кроме того, будем считать, что 

Разобьем произведение  следующим образом  где  Индекс  будем называть точкой разрыва. Тогда , где  – количество операций умножения при перемножении матриц  и  Другими словами, оптимальная расстановка скобок между матрицами  и  сводится к поиску точки разрыва  при которой количество операций умножения матриц  и  минимально.

Запишем окончательное рекуррентное соотношение, позволяющее вычислить минимальное количество операций умножения, в следующем виде:



Определим для примера минимальное количество операций умножения для матриц   и  из рассмотренного выше примера:









На 9 и 10 представлена функция **OptimalM**, реализующая алгоритм поиска оптимальной расстановки скобок при перемножении нескольких матриц.

//- MultiMatrix.h

**#define OPTIMALM\_PARM(x) ((int\*)x)** // представлениe двумерного массива

**int OptimalM(**

**int i,** // [in] номер первой матрицы

**int j,** // [in] номер последней матрицы

**int n,** // [in] количество матриц

**const int c[],** // [in] массив размерностей

**int\* s** // [out] результат: позиции скобок

**);**

Рис. 9. Функция оптимальной расстановки скобок при перемножении матриц

// - MultyMatrix.cpp

**#include "stdafx.h"**

**#include <memory.h>**

**#include "MultyMatrix.h"**

**#define INFINITY 0x7fffffff**

**int OptimalM(int i, int j, int n, const int c[], int \*s)**

**{**

**#define OPTIMALM\_S(x1,x2) (s[(x1-1)\*n+x2-1])**

**int o =INFINITY, bo = INFINITY;**

**if (i < j)**

**{**

**for (int k = i; k < j;k++)**

**{**

**bo = OptimalM(i,k, n, c, s)+**

**OptimalM(k+1,j,n, c, s)+ c[i- 1]\*c[k]\*c[j];**

**if (bo < o){o = bo; OPTIMALM\_S(i,j) = k;}**

**}**

**}**

**else o = 0;**

**return o;**

**#undef OPTIMALM\_S**

**};**

Рис. 10. Реализация функции **OptimalM**

На рис. 11 приведен пример программы, вызывающей функцию **OptimalM** для оптимальной расстановки скобок при перемножении шести матриц, а на рис. 12 – результат выполнения этой программы.

// --- main

// расстановка скобок

**#include "stdafx.h"**

**#include <cmath>**

**#include <memory.h>**

**#include <iostream>**

**#include "MultyMatrix.h"**

**#define N 6**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**int Mc[N+1] = {30,35,15,5,10,20,25}, Ms[N][N], r = 0, rd = 0;**

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

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

**setlocale(LC\_ALL, "rus");**

**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;**

**system("pause");**

**return 0;**

**};**

Рис. 11. Пример вызова функции **OptimalM**

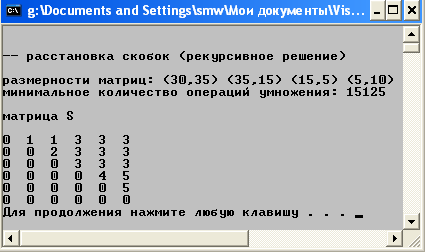


Рис. 12. Результат выполнения программы, представленной на рис. 11

Функция **OptimalM** имеет четыре входных параметра: **i** (номер первой матрицы), **j** (номер последней матрицы), **n** (общее количество перемножаемых матриц), **c** (массив размерностбю **n**, содержащий размерности матриц), а также один выходной параметр **s** (двумерный массив размерностью **n**, содержащий точки разрыва). Кроме того, к точке вызова возвращается минимальное количество операций умножения, необходимых для перемножения матриц заданной размерности.

Функция **OptimalM** является рекурсивной, так как она в процессе своей работы вызывает саму себя. Дно рекурсии достигается при совпадении значений двух первых параметров функции.

Расстановку скобок в заданной последовательности матриц можно осуществить с помощью двумерного массива **s**, возвращаемого функцией в последнем параметре. Поясним принцип использования этого массива для расстановки скобок при перемножении шести матриц       размерности которых заданы массивом **Mc** на рис. 7.11. Массив (матрица) **s**, который используется далее для расстановки скобок, является результатом работы программы (рис. 11) и выведен на рис. 12.

Скобки расставляются по принципу «сначала внешние – затем внутренние». Найдем элемент  в матрице **s** на рис. 12. Он равен 3. Это означает, что точка разрыва находится между первой и шестой матрицей после третьей матрицы, что позволяет расставить внешние скобки следующим образом: 

Точку разрыва между первой и третьей матрицей определяет элемент  а между четвертой и шестой –  После расстановки скобок выражение будет выглядеть следующим образом  Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 15 125.

**Решение задачи вычисления длины наибольшей общей подпоследовательности**

Пусть  – последовательность  символов . Последовательность  называется ***подпоследовательностью*** последовательности  если она может быть получена из последовательности  удалением ее некоторых элементов. Например, если  – последовательность символов  то последовательность  состоящая из символов  является подпоследовательностью  так как она может быть получена с помощью удаления элементов 

Подпоследовательность  называется ***общей подпоследовательностью*** последовательностей  и  если она является подпоследовательностью обеих последовательностей. Например, подпоследовательность  является общей подпоследовательностью последовательностей  и 

***Длиной последовательности*** (***подпоследовательности***) будем называть количество ее элементов. Например, длина последовательности  равна 6.

Подпоследовательность  называется ***наибольшей*** ***общей подпоследовательностью*** последовательностей  и  если  имеет наибольшую длину среди всех общих подпоследовательностей. Например, общая подпоследовательность  является наибольшей для последовательностей  и  В общем случае может быть несколько наибольших подпоследовательностей. Например,  тоже является наибольшей общей подпоследовательностью для последовательностей из последнего примера. Часто для обозначения наибольшей общей подпоследовательности используется сокращение **LCS** (longest common subsequence).

Рекурсивный алгоритм вычисления длины LCS  для двух последовательностей  и  основывается на трех следующих очевидных утверждениях, которые приводятся без доказательства:

1. Если  то  и  является LCS для  и 
2. Если  и  то  является LCS для  и 
3. Если  и  то  является LCS для  и 

Обозначив через  длину LCS для последовательностей  и  можем записать следующее рекуррентное соотношение:



Рассмотрим пример вычисления длины LCS для последовательностей  и  Вычисление осуществляется по шагам:

1. .
2. 
3. 
4. .
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 
13. 
14. 
15. 
16. 
17. 
18. 
19. 
20. 
21. 
22. 
23. 
24. 
25. 
26. 

Все шаги вычисления можно разбить на две группы: с 1 по 17 и с 18 по 26. Первая группа соответствует рекурсивному погружению, вторая – восходящему вычислению. Результатом вычисления является значение  равное длине LCS. Нетрудно убедиться, что для последовательностей  и существуют две LCS:  и  имеющих длину 3.

На рис.13 и 14 представлена рекурсивная функция **lcs**, вычисляющая длину LCS для двух заданных последовательностей, а на рис. 15 – пример программы, вызывающей эту функцию, и результат ее выполнения (рис. 16) соответственно.

// - LCS.h

// -- рекурсивное вычисление длины LCS

**int lcs (**

**int lenx,** // длина последовательности X

**const char x[],** // последовательность X

**int leny,** // длина последовательности Y

**const char y[]** // последовательность Y

**);**

Рис. 13. Прототип функции **lcs**, вычисляющей длину LCS для двух заданных последовательностей

// - LCS.cpp

// -- рекурсивное вычисление длины LCS

**#include "stdafx.h"**

**#include <algorithm>**

**#include "LCS.h"**

**int lcs (int lenx, const char x[],**

**int leny, const char y[])**

**{**

**int rc = 0;**

**if (lenx > 0 && leny > 0)**

**{**

**if (x[lenx-1] == y[leny-1]) rc = 1 + lcs(lenx-1, x,leny-1, y);**

**else rc = std::max(lcs(lenx, x,leny-1, y), lcs(lenx-1, x,leny, y));**

**}**

**return rc;** //длина LCS

**}**

Рис. 14. Реализация функции **lcs**

// - main

// -- вычисления длины LCS

**#include "stdafx.h"**

**#include <iostream>**

**#include "LCS.h"**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**char X[]="ALDC", Y[]="LADCM";**

**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: "<< X;**

**int s = lcs(**

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

**"ALDC",** // последовательность X

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

**"LADCM"** // последовательность Y

**);**

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

**system("pause");**

**return 0;**

**}**

Рис. 15. Пример вызова функции **lcs**

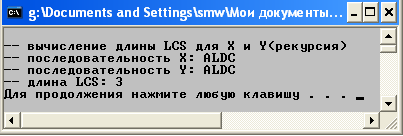


Рис. 16. Результат выполнения программы, представленной на рис. 15

Функция **lcs** имеет четыре параметра: **lenx** (длина первой последовательности), **x** (массив, содержащий символы первой последовательности), **leny** (длина второй последовательности), **x** (массив, содержащий символы второй последовательности). Функция возвращает длину LCS заданных последовательностей.

Функция **lcs** фактически повторяет запись рекуррентного соотношения, записанного выше, и поэтому не требует дополнительного пояснения. Обратите внимание, что исходные последовательности и результат совпадают с последовательностями и результатом вычисления предыдущего примера.

**Вычисление дистанции Левенштейна**

***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.

Впервые такой способ вычисления дистанции (меры различия) между двумя двоичными последовательностями предложил советский ученый-математик В. И. Левенштейн (род. 1935). Впоследствии способ распространился на последовательности произвольного алфавита.

Расстояние Левенштейна активно применяется для исправления ошибок в поисковых системах, в текстовых редакторах, а также в биоинформатике.

Пусть и  – две символьные строки, тогда для вычисления дистанции Левенштейна  между ними может быть использовано следующее рекуррентное соотношение:



В предыдущем выражении используются символы   и  Разъясним их смысл:

 – количество символов в заданной строке. Например, 

 – заданная строка без последнего символа. Например, 

 – последний символ заданной строки. Например, 

Поясним принцип применения этого рекуррентного соотношения на следующем примере.

Пусть необходимо вычислить  Тогда имеем следующую последовательность шагов:

1.  
2. 
3. 
4. 
5. 

1. 

1. 
2. 
3. 
4. 

1. 

1. 



1. 

1. 

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 
13. 
14. 

Шаги вычисления с 1 по 14 соответствуют рекурсивному погружению, а шаги с 15 по 28 – восходящему вычислению. Нетрудно убедиться, что для превращения слова «сор» в слово «спорт» достаточно удалить (или вставить) две буквы.

На рис. 17 и 18 представлена рекурсивная функция **levenshtein\_r**, вычисляющая дистанцию Левенштейна для двух строк, а на рис. 19 и 20 – пример программы, вызывающей эту функцию и результат ее выполнения соответственно.

// - Levenshtein.h

// -- 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

**);**

Рис. 17. Прототип функции  **levenshtein\_r**, вычисляющей дистанцию Левенштейна между двумя строками

// - Levenshtein.cpp

**#include "stdafx.h"**

**#include <algorithm>**

**#include "Levenshtein.h"**

**int min3(int x1, int x2, int x3)** // выбрать минимум из 3х

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

**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;**

**};**

Рис. 18. Реализация рекурсивной функции **levenshtein\_r**

// - main

**#include "stdafx.h"**

**#include <iostream>**

**#include "Levenshtein.h"**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**char X[] = "сор", Y[] = "спорт";**

**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;**

**}**

Рис. 19. Пример программы, вызывающей функцию **levenshtein\_r**

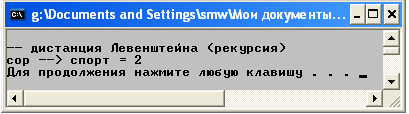


Рис. 20. Результат выполнения программы, представленной на рис. 19

Функция **levenshtein\_r** имеет четыре параметра: **lx** (длина первой строки), **x** (массив символов размерностью **lx**, содержащий символы первой строки), **ly** (длина второй строки), **y** (массив символов размерностью **ly**, содержащий символы второй строки). Фунция возвращает дистанцию Левенштейна между двуми заданными строками.

Реализация функции (рис. 17) практически повторяет рекуррентное соотношение, приведенное выше, поэтому не нуждается в дополнительном пояснении.

В программе, приведенной на рис. 18, вызывается функция **levenshtein\_r** для вычисления дистанции Левенштейна между строками «сор» и «спорт».

Результат выполнения программы (рис. 17) совпадает с результатом, полученным ранее вручную. Следует лишь обратить внимание, что в отличие от ручного просчета при выполнении функции **levenshtein\_r** некоторыеподзадачи решаются по несколько раз. Например, дистанция между строками «со» и «спор» вычисляется 3 раза, а это в каждом случае приводит к повторному вычислению дистанций между «с» и «спор», «со» и «спо», «с» и «спо» и т. д.

Следует отметить, что в информатике часто используется ***дистанция Дамерау – Левенштейна***, которая является модификацией дистанции Левенштейна и отличается применением еще одной операции преобразования – перестановки соседних символов.