**Лекция 7**

**Динамическое программирование**

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***.

На рис. 1 приведен пример реализации алгоритма вычисления члена последовательности Фибоначчи методом динамического программирования.

// - Динамическое прогрммирование

// -- вычисление n-го члена ряда Фибоначчи

**int fib(int n)**

**{**

**static int \*f = new int[n+1];**

**int rc = 0;**

**if (f[n] > 0) rc = f[n];**

**else if (n == 0) rc = (f[n] = 0);**

**else if (n == 1 || n == 2) rc = (f[n] = 1);**

**else rc = (f[n] = fib(n-1)+fib(n-2));**

**return rc;**

**}**

Рис. 1. Функция вычисления члена последовательности Фибоначчи, реализующая алгоритм динамического программирования

Обратите внимание, что новой реализацией функции **fib** для хранения промежуточных значений используется статический массив. Применение массива позволяете избежать повторных рекурсивных вычислений, которые были в реализации, представленной в пред. лекции.

Заметим, что задачи вычисления факториала и наибольшего общего делителя не имеет смысла решать методом динамического программирования, так как при их рекурсивном решении не образуются перекрывающиеся подзадачи и, следовательно, нет необходимости запоминать промежуточные вычисления.

Следует отметить, что рассматриваемый здесь метод является частью более общей теории динамического программирования, основы которой разработаны Р. Беллманом. Эта теория исследует процесс пошагового решения задач оптимизации, в котором на каждом шаге из множества допустимых решений выбирается одно, оптимизирующее заданную целевую функцию.

**Решение задачи о рюкзаке**

Применение динамического программирования при решении оптимизационных задач обычно предполагает создание специальных таблиц для хранения промежуточных результатов.

На рис. 2 изображены таблицы, которые используются для решения задачи о рюкзаке методом динамического программирования. Векторы, определяющие размеры () и стоимости () типов предметов, а также величина, характеризующая вместимость рюкзака (), заданы в верхней части рисунка.

Рис. 2. Таблицы, используемые для хранения промежуточных результатов, при решении задачи о рюкзаке методом динамического программирования

Количество таблиц, которое необходимо построить для решения поставленной задачи, соответствует заданному в задаче  – количеству типов предметов (на рис. 2 ). Каждая таблица, кроме помеченной символом S, пронумерована и описывает все возможные случаи размещения в рюкзаке соответствующего типа предметов.

Например, таблица Т.0 описывает все возможные способы размещения в рюкзаке предметов, имеющих объем 85 и стоимость 150 за единицу веса, а таблица T.1 – все способы размещения предметов объемом 50 и стоимостью 25.

Все таблицы имеют одинаковую структуру и содержат по три столбца: неиспользованный объем места в рюкзаке, объем, занятый соответствующим типом, стоимость вещей в рюкзаке.

Например, первая строка таблицы T.0 соответствует случаю, когда в рюкзак не помещено ни одного предмета этого типа. Третья строка описывает случай, когда в пустой рюкзак помещено два предмета, которые заняли объем  Третий столбец третей стоки содержит стоимость двух предметов, равную  =  единиц.

Десятая строка таблицы T.1 соответствует случаю, когда в рюкзаке остался незанятый объем, равный 215, и туда поместили два предмета, имеющих объем по 50 единиц. При этом общая стоимость вещей в рюкзаке вычисляется как сумма стоимости предметов, размещенных ранее (таблица T.0) в объеме  и предметов, помещенных в данный момент.

Определить стоимость ранее размещенных предметов можно по таблице T.0. Для этого необходимо найти строку, которая соответствует остатку 215. Это вторая строка, так как она описывает размещение в пустом рюкзаке (300 единиц незанятого объема) одного предмета объемом 85 и стоимостью 12 750. Таким образом, стоимость предметов в рюкзаке, которая будет записана в десятой строке таблицы T.1, будет  

Построение таблиц осуществляется последовательно в порядке их нумерации. Каждая таблица строится на основе предшествующей. Таблица, помеченная символом S, имеет чисто техническое назначение и предназначена для построения таблицы T.0. Таблица S состоит из одной строки, в первом столбце этой строки указывается вместимость рюкзака, а два другие столбца заполняются нулями.

Таблица с номером  (далее просто таблица *k*) строится на основе таблицы  При этом будем считать, что таблица S имеет номер –1 (минус один).

Каждой строке  в таблице  соответствует  строк в таблице  где  – содержимое первого,  – второго,  – третьего столбца таблицы  а – объем предмета *k*-го типа. Строки   таблицы  соответствующие строке  таблицы  имеют следующий вид:  где – стоимость единицы объема предмета *k*-го типа.

В последней таблице (в нашем случае в таблице T.6) каждая строка соответствует одной из допустимых комбинаций предметов, помещенных в рюкзак. Для выбора комбинации, имеющей наибольшую стоимость, необходимо просмотреть всю таблицу и выбрать строку, в которой третий столбец имеет максимальное содержимое (на рис. 7.22 эта строка выделена, содержимое третьего столбца – 39 650). Далее следует восстановить путь построения этой строки, продвигаясь от последней таблицы к первой, используя значение третьего столбца в качестве ключа.

Для того чтобы определить строку в таблице T.5, необходимо вычислить значение ее третьего столбца. Оно равно разнице содержимого третьего столбца выделенной в таблице T.6 строки (39 650) и стоимости предметов, общий объем которых является значением второго столбца (это значение равно 0). Таким образом, разница состовлять  Искомая в таблице T.5 будет иметь значение 37 650 в третьем столбце (на рис. 7.22 эта строка выделена).

Отыскав в таблице T.5 необходимую строку, следует подобным способом дойти до таблицы T.0 (на рис. 7.22 все выбранные строки выделены). Разделив содержимое выделенных строк на объем соответствующего таблице типа предмета, получим их количество. Таким образом, решением данной задачи будет вектор  каждая компонента которого – количество предметов соответствующего типа.

На рис. 3 и 4 представлена функция **knapsack\_d**, реализующая алгоритм динамического программирования решения задачи о рюкзаке.

// - Knapsack\_d.h

//-- алгоритм динамического программирования решения задачи о рюкзаке

**int knapsack\_d(**

**int V,** // вместимость рюкзака

**int n,** // количество предметов

**int v[],** // вес предметов

**int c[],** // стоимость предметов

**int m[]** // результат

**);**

Рис. 3. Прототип функции **knapsack\_d**

// - Knapsack\_d.cpp (начало)

**#include "stdafx.h"**

**#include <memory.h>**

**#include "Knapsack.h"**

**#define NINF 0x80000000**

**struct Table** // таблица

**{**

**int n;** // максимальное количество строк

**int cn;** // текущее количество строк

**struct Element** // строка

**{**

**int V;** // остаток вместимости

**int v;** // вес предметов

**int c;** // стоимость предметов

**void set(int V = 0, int v = 0, int c = 0)**

**{**

**this->V = V; this->v = v; this->c = c;**

**};**

**Element(int V = 0, int v = 0, int c = 0) {this->set(V, v, c);};**

**int remainV() {return (this->V - this->v);};**

**} \*e;** // массив строк таблицы

**Table(int n = 0)**

**{**

**this->e = new Element[this->n = n];**

**this->cn = 0;**

**};**

**};**

Рис. 4. Реализация функции **knapsack\_d**

// - Knapsack\_d.cpp (окончание)

**void insert(Table& t, int V, int v, int C,int c)** // добавить строки

**{**

**for (int i = 0; i < V/v+1; i++) t.e[t.cn++].set(V, i\*v, C+i\*c\*v);**

**};**

**Table\* create (Table s, int v, int c)** // создать следующую таблицу

**{**

**Table \*rc ;**

**int tsize = 0;**

**for (int i = 0; i < s.cn; i++)**

**tsize+=(s.e[i].remainV()/v+1);**

**rc = new Table(tsize);**

**for (int i = 0; i < s.cn; i++)**

**insert(\*rc,s.e[i].remainV(), v, s.e[i].c, c);**

**return rc;**

**};**

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

**{**

**int mmax = NINF, \*path = new int[n];**

**Table s(1), \*\*t = new Table\* [n];**

**insert(s,V, V+1,0,0);**

**t[0] = create(s, v[0], c[0]);**

**for (int i = 1; i < n; i++) t[i] = create(\*t[i-1], v[i],c[i]);**

**mmax = NINF;**

**for (int j = 0; j < t[n-1]->cn; j++)**

**if (t[n-1]->e[j].c > mmax)**

**{**

**mmax = t[n-1]->e[j].c;**

**path[n-1] = j;**

**m[n-1] = t[n-1]->e[j].v/v[n-1];**

**};**

**for(int i = n-2; i >= 0; i--)**

**{**

**mmax = NINF;**

**for (int j = 0; j < t[i]->cn; j++)**

**if (t[i]->e[j].c > mmax &&**

**t[i]->e[j].remainV() == t[i+1]->e[path[i+1]].V)**

**{**

**mmax = t[i]->e[j].c;**

**path[i] = j;**

**m[i] = t[i]->e[j].v/v[i];**

**};**

**};**

**return t[n-1]->e[path[n-1]].c;**

**};**

Рис. 4. Окончание

Прототип функции **knapsack\_d** ничем не отличается от прототипа функции **knapsack\_r**,описанной ыше, поэтому не будем останавливаться на описании параметров.

Реализация функции **knapsack\_d** претерпелазначительные изменения, обусловленные необходимостью сохранять промежуточные результаты вычислений. Для хранения промежуточных результатов применяется структура **Table** (структура моделирует таблицу в описанном выше алгоритме), представляющую собой коллекцию структур **Element** (строки таблицы). Конструктор структуры **Table** имеет единственный параметр, задающий количество элементов типа **Element** в коллекции.

Для работы со структурами **Table** и **Element** служат вспомогательные функции **create** и **insert**. Функция **create** создает (и возвращает) структуру **Table** на основе предшествующей структуры **Table**, заданной первым параметром. Второй и третий параметры задают размер и стоимость предмета, для которого строится таблица. Для вставки строк в созданную таблицу (структуру **Table**) используется функция **insert**.

Функция **insert** имеет пять параметров: **t** (таблица, в которую добавляются строки), **V** (остаток неиспользованного объема в рюкзаке), **v** (объем размещаемого предмета), **C** (стоимость предметов в занятом объеме рюкзака), **с** (стоимость размещаемого предмета).

Условно весь текст функции **knapsack\_d** можно разбить на две части: сначала с помощью функции **create** создается массив таблиц (структур **Table**), а далее обратным ходом формируется решение.

Если в программе, приведенной на рис. 7 (см. пред. лекц.) заменить вызов функции **knapsack\_r** на **knapsack\_d**, проинициализировать переменную **V**, массивы **v** и **c** соответствующими значениями из условия задачи на рис. 2, а также правильно задать количество предметов (изменить макрос **NN**), то результат выполнения этой программы будет таким же, как на рис. 5.

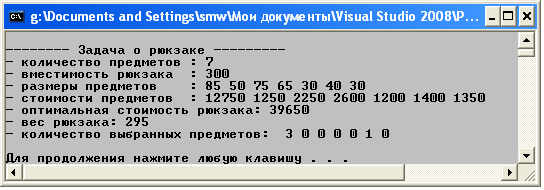


Рис. 7.5. Результат решения задачи о рюкзаке

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

На рис. 6, 7 представлена функция **OptimalMD**, которая реализует алгоритм динамического программирования для решения задачи о расстановке скобок.

//- MultiMatrix.h

**int OptimalMD**(

**int n,** // [in] количество матриц

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

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

**);**

Рис. 6. Прототип функции OptimalMD

// - MultyMatrix.cpp

**int OptimalMD(int n, const int c[], int\* s)**

**{**

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

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

**int \*M = new int[n\*n], j = 0, q = 0;**

**for (int i = 1; i <= n; i++) OPTIMALM\_M(i,i) = 0;**

**for (int l = 2; l <= n; l++)**

**{**

**for (int i = 1; i <= n-l+1; i++)**

**{**

**j = i+l-1;**

**OPTIMALM\_M(i,j) = INFINITY;**

**for (int k = i; k <= j-1; k++)**

**{**

**q = OPTIMALM\_M(i,k) + OPTIMALM\_M(k+1,j)+c[i-1]\*c[k]\*c[j];**

**if (q < OPTIMALM\_M(i,j))**

**{**

**OPTIMALM\_M(i,j) = q; OPTIMALM\_S(i,j)= k;**

**}**

**}**

**}**

**}**

**return OPTIMALM\_M(1,n);**

**#undef OPTIMALM\_M**

**#undef OPTIMALM\_S**

**};**

Рис. 7. Реализация функции OptimalMD

Функция **OptimalMD** имеет три параметра: **n** (общее количество перемножаемых матриц), **M** (массив размерностью **n**, содержащий размерности матриц), а также выходной параметр **s** (двумерный массив размерностью **n**, содержащий точки разрыва).

Структура функции **OptimalMD** практически идентична функции **OptimalM**, описанной в подразделе 7.1.2. Главное отличие – применение массива **M** (работа с ним осуществляется с помощью макроса **OPTIMAL\_M**) для хранения промежуточных вычислений.

**Решение задачи о наибольшей общей подпоследовательности**

На рис. 8 и 9 представлена функция **lcsd**, реализующая алгоритм динамического программирования для решения задачи о наибольшей общей подпоследовательности (LCS).

**//- LCH.h**

**int lcsd(**

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

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

**char z[]** // наибольшая общая подпоследовательность

**);**

Рис. 8. Прототип функции lcsd

//- LCH.cpp

**#include "stdafx.h"**

**#include "LCS.h"**

**#define LCS\_C(x1,x2) (C[(x1)\*(leny+1)+(x2)])**

**#define LCS\_B(x1,x2) (B[(x1)\*(leny+1)+(x2)])**

**#define LCS\_X(i) (x[(i)-1])**

**#define LCS\_Y(i) (y[(i)-1])**

**#define LCS\_Z(i) (z[(i)-1])**

**enum Dart{TOP,LEFT,LEFTTOP};**

**void getLCScontent( int lenx, int leny, const char x[],**

**const Dart\* B,**

**int n, int i, int j, char z[])**

**{**

**if ((i > 0 && j > 0 && n > 0 ))**

**{**

**if (LCS\_B(i,j) == LEFTTOP)**

**{**

**getLCScontent(lenx, leny,x, B, n-1, i-1, j-1, z);**

**LCS\_Z(n) = LCS\_X(i);**

**LCS\_Z(n+1) = 0;**

**}**

**else if (LCS\_B(i,j)== TOP)**

**getLCScontent(lenx, leny,x, B, n, i-1, j, z);**

**else getLCScontent(lenx, leny,x, B, n, i, j-1, z);**

**}**

**};**

**int lcsd(const char x[], const char y[], char z[])**

**{**

**int n;**

**int lenx = strlen(x), leny = strlen(x),**

**\*C = new int[(lenx+1)\*(leny+1)];**

**Dart\* B = new Dart[(lenx+1)\*(leny+1)];**

**memset(C,0,sizeof(int)\*(lenx+1)\*(leny+1));**

**for (int i = 1; i <= lenx; i++)**

**for(int j = 1; j <= leny; j++)**

**if (LCS\_X(i) == LCS\_Y(j))**

**{LCS\_C(i,j) = LCS\_C(i-1,j-1)+1;**

**LCS\_B(i,j) = LEFTTOP;}**

**else if (LCS\_C(i-1,j) >= LCS\_C(i, j-1))**

**{**

**LCS\_C(i,j) = LCS\_C(i-1, j);**

**LCS\_B(i,j) = TOP;**

**}**

**else**

**{**

**LCS\_C(i,j) = LCS\_C(i, j-1);**

**LCS\_B(i,j) = LEFT;**

**}**

**getLCScontent(lenx, leny, x, B, LCS\_C(lenx,leny), lenx, leny, z);**

**return LCS\_C(lenx,leny);**

**}**

**#undef LCS\_Z**

**#undef LCS\_C**

**#undef LCS\_B**

**#undef LCS\_X**

**#undef LCS\_Y**

Рис. 9. Реализация функции **lcsd**

Функция **lcsd** имеет три параметра: **x** (символьная строка, интерпретируемая как первая заданная последовательность), **y** (символьная строка, интерпретируемая как вторая заданная последовательность) и возвращаемый параметр **z** (символьная строка, интерпретируемая как LCS двух последовательностей, заданных первыми двумя параметрами). Функция возвращает длину LCS.

Для построения наибольшей общей подпоследовательности в функции **lcsd** используются два массива **С** и **B**.Массивы моделирует матрицы размерностью  где  – размерности соответственно первой и второй последовательностей. Номера строк матриц, начиная с первой, соответствуют номерам элементов первой последовательности, а номера столбцов, тоже начиная с первого, – номерам элементов второй последовательности. Дляудобства работы с массивами как с матрицами применяются два макроса: **LCS\_C** и **LCS\_B**.

Массив **C** предназначен для вычисления длины LCS. Элементы массива **C** – числа. Вычисление элементов **C** осуществляется пошагово, начиная с северо-западного угла матрицы. В результате вычислений в последней строке последнего столбца матрицы **C** (в юго-восточном углу) находится длина LCS.

Массив **B** предназначен для построения массива LCS (возвращаемый параметр **z**). Массив **B** моделирует матрицу, элементами которой могут быть три типа символов, обозначающих направления: **TOP** (вверх), **LEFT** (налево), **LEFTTOP** (налево и вверх). Заполнение матрицы **B** осуществляетсяодновременнозаполнением матрицы **C** и в том же порядке.

На рис. 10 изображены матрицы С и **B** после завершения работы алгоритма функции **lcsd** для двух последовательностей B, D, C, A, B, A и A, B, C, B, D, A, B.



Рис. 10. Заполненные матрицы **C** и **B**

Функция **lcsd** вызывает вспомогательную рекурсивную функцию **getLCScontent**,предназначенную для формирования строки LSC.

Функция имеет восемь параметров: **lenx** (длина первой последовательности), **leny** (длина второй последовательности), **x** (символьная строка, интерпретируемая как первая заданная последовательность), **B** (сформированный массив **B**), **n** (длина LCS), **i** (номер строки текущего элемента матрицы **B**), **j** (номер столбца текущего элемента матрицы **B**), а также возвращаемый параметр **z** (символьная строка, интерпретируемая как LCS двух последовательностей).

Функция **getLCScontent** выбирает элементы массива **B**,начиная с последнего элемента последней строки, используя значения выбранных элементов массива как указатель направления перехода к следующему элементу. На рис. 10 выбранные элементы массива **B** выделены. Выбранные элементы массива **B**,значение которых **LEFTTOP** (на рис. 10 – диагональная стрелка), соответствуют элементам последовательностей, вошедших в LSC.

**Вычисление дистанции Левенштейна**

На рис. 11, 12 представлена функция **levenshtein**, реализующая алгоритм динамического программирования вычисления дистанции Левенштейна.

// - Levenshtein.h

// -- дистанции Левенштeйна (динамическое программирование)

**int levenshtein(**

**int lx,** // длина слова x

**сonst char x[],** // слово длиной lx

**int ly,** // длина слова y

**const char y[]** // слово y

**);**

Рис. 11. Прототип функции **levenshtein**

**#include "stdafx.h"**

**#include <iomanip>**

**#include <algorithm>**

**#include "Levenshtein.h"**

**#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);**

**}**

Рис. 12. Реализация функции **levenshtein**

Функция **levenshtein** имеет тот же набор параметров, что и функция **levenshtein\_r**,описанная в пред. лекции. Более того, реализации этих двух функций практически одинаковы. Основное отличие – применение в функции **levenshtein** массива **d** (для работы с ним используется макрос **DD**) для хранения результатов промежуточных вычислений.

Интерес представляет сравнительный анализ скорости вычисления дистанции Левенштейна функциями **levenshtein\_r** (рекурсивный алгоритм) и **levenshtein** (алгоритм динамического программирования) сделанный для строк различной длины с помощью программы, представленной на рис. 13. В программе поочередно вызываются функции **levenshtein\_r** и **levenshtein** с одинаковыми значениями параметров и замеряется продолжительность выполнения этих функций.

// --- main

// вычисление дистанции (расстояния) Левенштейна

**#include "stdafx.h"**

**#include <algorithm>**

**#include <iostream>**

**#include <ctime>**

**#include <iomanip>**

**#include "Levenshtein.h"**

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

**{**

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

**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;**

**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)<<(t2-t1)**

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

**}**

**system("pause");**

**return 0;**

**}**

Рис. 13. Программа для сравнительного анализа продолжительности выполнения функций **levenshtein\_r** и **levenshtein**

На рис. 14 представлен результат выполнения программы, приведенной на рис. 13. Продолжительность выполнения функций в распечатке приведена в миллисекундах.

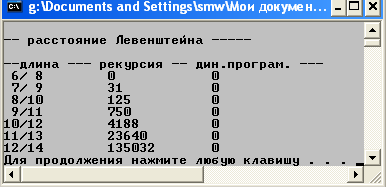


Рис. 14. Результат выполнения программы, представленной на рис. 13

Рис. 14 демонстрирует подавляющее превосходство алгоритма, основанного на динамическом программировании. При последнем просчете, в котором вычислялась дистанция Левенштейна между строками длиной 12 и 14 символов, рекурсивный алгоритм затратил на вычислении более 2 мин, а алгоритм, основанный на динамическом программировании, – менее 0,001 с.