Белорусский государственный технологический университет

Кафедра Информационных Систем и Технологий

**Лабораторная работа №4**

**Динамическое программирование**

Выполнилстудент 2 курса 7 группы

Бобрович Глеб Сергеевич

Минск 2021

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

**Условие:**

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

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main()  {  printf("GENERATE 300 SYMBOLS\n\n");  int i;  srand(time(NULL));  unsigned char c;  for (i = 0; i <= 300; ++i)  {  c = (rand() % ('z' - 'a' + 1)) + 'a';  printf("%c", c);  }  printf("\n\nGENERATE 250 SYMBOLS\n\n");  srand(time(NULL));  unsigned char cC;  for (i = 0; i <= 300; ++i)  {  cC = (rand() % ('z' - 'a' + 1)) + 'a';  printf("%c", cC);  }  return 0;  } |
|  |

**Задание 2.** Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – дистанцию Левенштейна для k=0,04; 0,05; 0,07; 0,1; где - длина строки ,  - строка состоящая из первых  символов строки . (копии экрана и код вставить в отчет).

|  |
| --- |
| // - Levenshtein.h  // -- дистанции Левенштeйна (динамическое программирование)  int levenshtein(  int lx, // длина слова x  const char x[], // слово длиной lx  int ly, // длина слова y  const char y[] // слово y  );    // -- 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  );  // - Levenshtein.cpp  #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);  }  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;  };  // --- 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 << x << "->" << y << " = " << levenshtein\_r(sizeof(x)-1, x, sizeof(y)-1, y) << 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;  } |
|  |

**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).



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. При кол-ве букв 60 и 50 рекурсивный метод проработал всю ночь, но так и не выдал решения.

**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).

1. **Вычисление дистанции Левенштейна**



 – количество символов в заданной строке. Например, 

 – заданная строка без последнего символа. Например, 

 – последний символ заданной строки. Например, 

|  |  |
| --- | --- |
| эхо | Хорёк |

Пусть необходимо вычислить L (“эхо”, “хорёк”) Тогда имеем следующую последовательность шагов:

1. L (“эхо”, “хорёк”) =
2. L (“эх”, “хорёк”) =
3. L (“эхо”, “хорёк”) =
4. L (“эх”, “хорё”) =
5. L (“э”, “хорёк”) =

L (“”, “хорёк”)=5, L (“”, “хорё”)=4

1. L (“э”, “хорё”) =

L (“”, “хорё”)=4, L (“”, “хор”)=3

1. L (“эхо”, “хор”) =
2. L (“эх”, “хор”) =
3. L (“эхо”, “хо”) =
4. L (“эхо”, “х”) =

L (“эхо”, “”)=3, L (“эх”, “”)=2

1. L (“э”, “хор”) =

L (“”, “хор”)=3, L (“”, “хо”)=2

1. L (“эх”, “хо”) =

L (“э”, “х”)=1

1. L (“э”, “хо”) =

L (“”, “хо”)=2, L (“э”, “х”)=1, L (“”, “х”)=1

1. L (“эх”, “х”) =

L (“э”, “х”)=1, L (“эх”, “”)=2, L (“э”, “”)=1

1. L (“эх”, “х”)=min(2, 2, 1)=1
2. L (“э”, “хо”)=min(3, 2, 2 )=2
3. L (“эх”, “хо”)=min(3, 2, 2)=2
4. L (“э”, “хор”)=min(4, 3, 3)=3
5. L (“эхо”, “х”)=min(2, 4, 3)=2
6. L (“эхо”, “хо”)=min(3, 3, 2)=2
7. L (“эх”, “хор”)=min(4, 3, 3)=3
8. L (“эхо”, “хор”)=min(4, 2, 3)=2
9. L (“э”, “хорё”)=min(5, 4, 4)=4
10. L (“э”, “хорёк”)=min(6, 5, 5)=5
11. L (“эх”, “хорё”)=min(5, 4, 4)=4
12. L (“эхо”, “хорё”)=min(5, 3, 4)=3
13. L (“эх”, “хорёк”)=min(6, 5, 5)=5
14. L (“эхо”, “хорёк”)=min(6, 4, 5)=4

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | \* | Х | хо | хор | хорё | хорёк |
| \* | 0 | 1 | 2 | 3 | 4 | 5 |
| Э | 1 | 1 | 2 | 3 | 4 | 5 |
| Эх | 2 | 1 | 2 | 3 | 4 | 4 |
| Эхо | 3 | 2 | 2 | 2 | 3 | 4 |

Таким образом дистанция Левенштейна для данных строк равна 4.

**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. Объяснить в отчете принцип расставления скобок по итоговой матрице + код + копии экрана.

|  |  |  |
| --- | --- | --- |
| эхо | хорёк | 8\*11, 11\*19, 19\*22, 22\*29, 29\*39, 39\*50 |

|  |
| --- |
| #define OPTIMALM\_PARM(x) ((int\*)x) // для представления 2мерного массива  int OptimalM( // рекурсия  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  );  int OptimalMD( // динамическое программирование  int n, // [in] количество матриц  const int c[], // [in] массив размерностей  int\* s // [out] результат: позиции скобок  );  // расстановка скобок (рекурсия)  #include"stdafx.h"  #include<memory.h>  #include"MultiMatrix.h"  #define INFINITY 0x7fffffff  #define NINFINITY 0x80000000  int OptimalM(int i, int j, int n, constint 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  };  // --- MultyMatrix.cpp (продолжение)  // расстановка скобок (динамическое программирование)  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  };  // --- main  // расстановка скобок  #include"stdafx.h"  #include<cmath>  #include<memory.h>  #include<iostream>  #include"MultiMatrix.h"// умножение матриц  #define N 6  int main()  {  int Mc[N+1] = {8, 11, 19, 22, 29, 39, 50 }, 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;  memset(Ms,0,sizeof(int)\*N\*N);  rd = OptimalMD(N, Mc, OPTIMALM\_PARM(Ms));  std::cout<<std::endl  <<"-- расстановка скобок (динамичеое программирование) "<< 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: "<< rd;  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<<std::endl;  system("pause");  return 0;  } |
|  |

Принцип расстановки скобок по итоговой матрице:

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

А1=8,

А2=11,

А3=19,

А 4 =22,

А 5 =29,

А 6 =39,

Найдем элемент (1,6) в матрице S, он равен 5. Это означает, что точка разрыва между 1-ой и 6-ой матрицей находится после 5-ой матрицы. Что позволяет расставить скобки следующим образом:

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

Точку разрыва между первой и пятой матрицей определяет элемент (1,5). Он равен 4. Следовательно разрыв будет после четвертой матрицы.

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

Далее берем элемент (1,4) и получаем, что он равен 3. Следовательно получаем:

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

И на последнем шаге мы возьмем элемент (1,3) и он равен 2:

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

Это выражение и есть конечное.

Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 34678.

**Вывод:** освоили общие принципы решения задач методом динамического программирования, сравнили полученные решения задач с рекурсивным методом. Мы выяснили, что метод динамического программирования справляется со строками длиной 300 и 250 символов за 97 единиц тактового времени, в то время как рекурсивный метод уже на 14 символах считает примерно 3 секунды (3367 единиц тактового времени). Таким образом для решения задачи о нахождении расстояния Левенштейна динамическое программирование будет более эффективным методом, чем рекурсивный метод.