# 实验二 动态规划算法

## 实验目的：

理解动态规划算法的思想

## 实验课时：

4学时

## 实验原理：

动态规划算法思想：把待求解问题分解成若干个子问题，先求解子问题，然后由这些子问题的解得到原问题的解，但动态规划求解过的子问题的结果会被保留下来，不像递归那样每个子问题的求解都要从头开始返回求解。动态规划求解问题的关键在于获得各个阶段子问题的递推关系式：

（1）分析原问题的最优解性质，刻画其结构特征

（2）递归定义最优值

（3）自底向上（由后向前）的方式计算最优值

（4）根据计算最优值时得到的信息，构造一个最优解。

## 实验题目：

### 基本题：

1、矩阵连乘问题 ：给定n个矩阵｛A1,A2,…,An｝，其中Ai与Ai+1是可乘的，i=1，2…，n-1。如何确定计算矩阵连乘积的计算次序，使得依此次序计算矩阵连乘积需要的数乘次数最少。如数据文件input.txt为：

6（矩阵个数）

30，35，15，5，10，20，25

代码如下所示。

package second;  
*//矩阵乘法*public class FirstExercise {  
 public static void main(String[] args){  
 FirstExercise mc = new FirstExercise();  
*// 矩阵个数为7* int n = 7;  
*// 矩阵的规模* int p[] = { 30, 35, 15, 5, 10, 20, 25 };  
*// 存储矩阵相乘的值* int m[][] = new int[n][n];  
*// 辅助信息* int s[][] = new int[n][n];  
 int l = p.length-1;  
 mc.*MatrixChain*(p, l,m, s);  
 for (int i = 1; i < n; i++) {  
 for (int j = 1; j < n; j++) {  
 System.*out*.printf("%-7d",m[i][j]); *//控制格式* }  
 System.*out*.println();  
 }  
  
 System.*out*.println();  
  
 for (int i = 1; i < n; i++) {  
 for (int j = 1; j < n; j++) {  
 System.*out*.printf("%-7d", s[i][j]);  
 }  
 System.*out*.println();  
 }  
  
 mc.*Traceback*( 1, 6, s);  
 }  
  
 public static void MatrixChain(int[] p,int n, int[][] m, int[][] s) {  
 for (int i = 1; i <= n; i++) {  
 m[i][i] = 0;  
 }

for(int r = 2;r <= n; r++ ) { *//矩阵连乘的个数* for(int i = 1; i <= n-r+1; i++) { *//从第一个矩阵开始相乘* int j = i+r-1;  
 m[i][j] = m[i+1][j] + p[i-1]\*p[i]\*p[j];  
 s[i][j] = i;  
 for(int k = i+1; k < j; k++) {  
 int t = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];  
 if(t < m[i][j]) {  
 m[i][j] = t;  
 s[i][j] = k;  
 }  
 }  
  
 }  
 }  
 }  
 public static void Traceback(int i, int j, int[][] s) {  
 if(i == j) return;  
 *Traceback*(i,s[i][j],s);  
 *Traceback*(s[i][j] + 1,j,s);  
 System.*out*.println("Multiply A" + i + ", " + s[i][j] + "and A" + (s[i][j] + 1) + "," + j);  
 }  
}

实验结果如图1所示，在矩阵乘法中，对问题进行分析，该问题符合最优子结构性质，且满足重复求解，所以采用分治法进行求解，自底向上逐步求解。算法的求解过程课拆解为，先求出2个矩阵相乘的最优值m[i][j]，然后求解3个矩阵相乘的最优值，以此类推，得到全部矩阵相乘的最优值。并且在求解的过程中，用二维矩阵s记录矩阵得到最优值时的矩阵序号，通过递归调用可得到最优相乘顺序。其中，m[i][j]为i到j矩阵的最优相乘次数，可以递归定义为：当i=j时，m[i][j]=0；当i<j时，m[i][j]=min{m[i][k]+m[k+1][j]+p[i-1][k][j]}，i<=k<j。

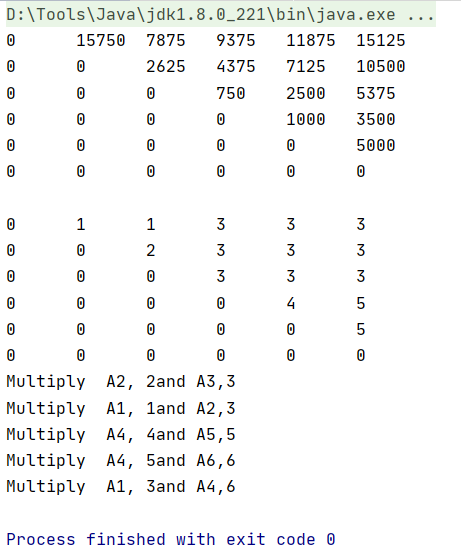


图1

2、最长公共子序列

给定2个序列X={x1,x2,…,xm}和Y={y1,y2,…,yn}，找出X和Y的最长公共子序列。

代码如下：

package second;  
*//最长公共子序列*public class SecondExercise {  
 public static void main(String[] args){  
 SecondExercise object=new SecondExercise();  
 char[] x={'A','A','G','A'};  
 char[] y={'A','T','A','G','C','G','T','C'};  
 System.*out*.println("x数组为：" );  
 for (int i = 0; i < x.length; i++){  
 System.*out*.print(x[i] + "\t");  
 }  
 System.*out*.println("\n y数组为：" );  
 for (int i = 0; i < y.length; i++){  
 System.*out*.print(y[i] + "\t");  
 }  
 *//定义一个数组用来记录c[i][j]的值是由哪一个子问题的解得到的* int [][] Rec=new int[x.length+1][y.length+1];  
 object.LcsLength(x,y,Rec);  
 }  
  
 public void LcsLength(char[] x,char[] y,int[][] Rec){  
 *//初始化* int [][]c=new int[x.length+1][y.length+1];  
 for(int i=0;i<=x.length;i++){  
 c[i][0]=0;  
 Rec[i][0]=0;  
 }  
 for(int j=0;j<=y.length;j++){  
 c[0][j]=0;  
 Rec[0][j]=0;  
 }  
 *//动态规划* for(int i=1;i<=x.length;i++){  
 for(int j=1;j<=y.length;j++){  
 if(x[i-1]==y[j-1]){  
 c[i][j]=c[i-1][j-1]+1;  
 *//当Rec为1时，表示Xi和Yi的最长公共子序列是由Xi-1和Yi-1的最长公共子序列在尾部加上Xi所得的子序列。* Rec[i][j]=1;  
 }else if(c[i-1][j]>=c[i][j-1]){  
 c[i][j]=c[i-1][j];  
 *//当Rec为2时，表示Xi和Yi的最长公共子序列与Xi-1和Yi的最长公共子序列相同。* Rec[i][j]=2;  
 }else{  
 c[i][j]=c[i][j-1];  
 *//当Rec为3时，表示Xi和Yi的最长公共子序列与Xi和Yi-1的最长公共子序列相同。* Rec[i][j]=3;  
 }  
 }  
 }  
 System.*out*.println("\n动态规划表格为：");  
 for(int i=0;i<=x.length;i++){  
 for(int j=0;j<=y.length;j++){  
 System.*out*.print(c[i][j]+" ");  
 }  
 System.*out*.print("\n");  
 }  
  
 System.*out*.println("记录子问题解的来源的数组为：");  
 for(int i=0;i<=x.length;i++){  
 for(int j=0;j<=y.length;j++){  
 System.*out*.print(Rec[i][j]+" ");  
 }  
 System.*out*.print("\n");  
 }  
   
 System.*out*.println("最长公共子序列为：");  
 lcs(x.length,y.length,x,Rec);  
 }  
  
 public void lcs(int i,int j,char[] x,int[][] Rec){  
 if(i==0||j==0) return ;  
 if(Rec[i][j]==1){  
 lcs(i-1,j-1,x,Rec);  
 System.*out*.print(x[i-1]);  
 }else if(Rec[i][j]==2) lcs(i-1,j,x,Rec);  
 else lcs(i,j-1,x,Rec);  
 }  
}

实验结果如图2所示，最长公共子序列的递推公式为：当i>0；j=0时，c[i][j]=0；当i，j>0；xi=yj时，c[i][j]=c[i-1][j-1]+1，在辅助数组用1表示这种情况；当i，j>0；xi!=yj时，c[i][j]=max{c[i][j-1]，c[i-1][j]}，其中，在辅助数组中用2表示从c[i-1][j]中得到的情况，用3表示从c[i][j-1]得到的情况。最后用递归得到公共最长子序列。

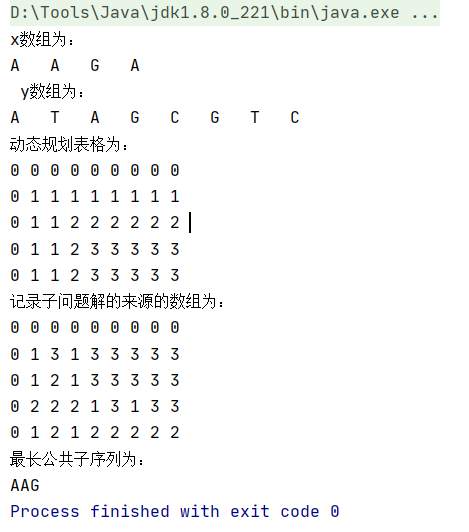


图2

### 提高题：

1、用动态规划法求解0/1背包问题：给定n种物品和一个背包，物品i的重量是Wi，其价值为Vi，问如何选择装入背包的物品，使得装入背包的物品的总价值最大？

代码如下：

package second;  
*//背包问题*public class ThirdExercise {  
 public static void main(String[] args){  
 int[] w = {1, 4, 3};*//物品的重量* int[] val = {1500, 3000, 2000}; *//物品的价值 这里val[i] 就是前面讲的v[i]* int m = 4; *//背包的容量* int n = val.length; *//物品的个数* System.*out*.println("物品的重量为：");  
 for (int i = 0; i < w.length; i++){  
 System.*out*.print(w[i] + "\t");  
 }  
  
 System.*out*.println("\n物品的价值为：");  
 for (int i = 0; i < val.length; i++){  
 System.*out*.print(val[i] + "\t");  
 }  
 System.*out*.println();  
  
 *//创建二维数组，  
 //v[i][j] 表示在前i个物品中能够装入容量为j的背包中的最大价值* int[][] v = new int[n+1][m+1];  
 *//为了记录放入商品的情况，我们定一个二维数组* int[][] path = new int[n+1][m+1];  
  
 *//初始化第一行和第一列, 这里在本程序中，可以不去处理，因为默认就是0* for(int i = 0; i < v.length; i++) {  
 v[i][0] = 0; *//将第一列设置为0* }  
 for(int i=0; i < v[0].length; i++) {  
 v[0][i] = 0; *//将第一行设置0* }  
  
 *//根据前面得到公式来动态规划处理* for(int i = 1; i < v.length; i++) { *//不处理第一行 i是从1开始的* for(int j=1; j < v[0].length; j++) {*//不处理第一列, j是从1开始的  
 //公式* if(w[i-1]> j) { *// 因为我们程序i 是从1开始的，因此原来公式中的 w[i] 修改成 w[i-1]* v[i][j]=v[i-1][j];  
 } else {  
 *//说明:  
 //因为我们的i 从1开始的， 因此公式需要调整成  
 //v[i][j]=Math.max(v[i-1][j], val[i-1]+v[i-1][j-w[i-1]]);  
 //v[i][j] = Math.max(v[i - 1][j], val[i - 1] + v[i - 1][j - w[i - 1]]);  
 //为了记录商品存放到背包的情况，我们不能直接的使用上面的公式，需要使用if-else来体现公式* if(v[i - 1][j] < val[i - 1] + v[i - 1][j - w[i - 1]]) {  
 v[i][j] = val[i - 1] + v[i - 1][j - w[i - 1]];  
 *//把当前的情况记录到path 插眼* path[i][j] = 1;  
 } else {  
 v[i][j] = v[i - 1][j];  
 }  
  
 }  
 }  
 }  
  
 *//输出一下v 看看目前的情况* System.*out*.println("背包的价值矩阵为：");  
 for(int i =0; i < v.length;i++) {  
 for(int j = 0; j < v[i].length;j++) {  
 System.*out*.printf("%-7d",v[i][j]);  
 }  
 System.*out*.println();  
 }  
  
 System.*out*.println("============================");  
 *//输出最后我们是放入的哪些商品  
 //遍历path, 这样输出会把所有的放入情况都得到, 其实我们只需要最后的放入* for(int k=0; k < n+1; k++){  
 for(int l = 0; l < path[0].length; l++){  
 System.*out*.print(path[k][l] + "\t");  
 }  
 System.*out*.println();  
 }  
  
  
 *// 回溯选择的商品* int i = path.length - 1; *//行的最大下标* int j = path[0].length - 1; *//列的最大下标* while(i > 0 && j > 0 ) { *//从path的最后开始找* if(path[i][j] == 1) {  
 System.*out*.printf("第%d个商品放入到背包\n", i);  
 j -= w[i-1]; *//w[i-1]* }  
 i--;  
 }  
 }  
}

实验结果如图3所示，在0-1背包问题中，对于物品的选择只有选和不选两种情况。当第i个物品装入背包时，首先要判断该物品装入背包后，是否超过背包的最大承重量，如果超过，则该物品不能加入背包，如果没有超过，则要判断是将该物品装入后的价值大，还是不装入得到的价值大，其递归公式为：当0<=j<wi时，m(i,j)=m(i-1,j)；当j>wi时，m(i,j)=max{m(i-1,j)，m(i-1,j-wi)+vi}。其中wi表示第i个物品的重量，vi表示第i个物品的价值。最后，用递归公式得到装入背包的物品。

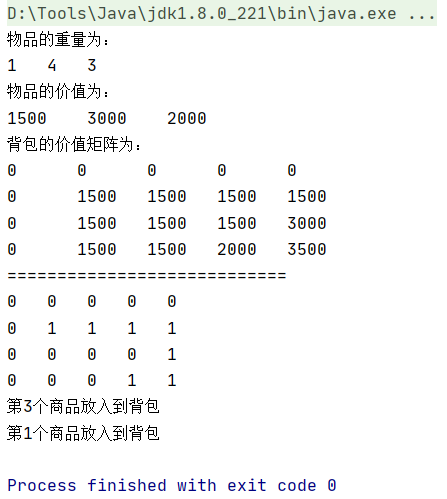


图3

2、图像压缩 ：要求确定象素序列{p1,p2,…,pn}的最优分段，使得依此分段所需的存储空间最少。每个分段的长度不超过256位。

代码如下：

package second;  
  
import java.util.Arrays;  
  
*//图像压缩*public class FourthExercise {  
 public static void main(String[] args){  
 int[] text=new int[]{10,12,15,255,1,2};  
 System.*out*.println("图像的灰度序列为：");  
 for (int i = 0; i < text.length; i++) {  
 System.*out*.print(text[i] + "\t");  
 }  
 System.*out*.println();  
  
 System.*out*.println("图像最后的压缩空间为：" + *optimalBits*(text));  
 }  
  
 public static int optimalBits(int[] dots){  
 int n= dots.length;  
 int[] s=new int[n+1];  
 s[0]=0;  
 for(int i=1;i<=n;i++){  
 s[i] = s[i-1]+*minBit*(dots[i-1])+11;  
 for(int j=1;j<=i;j++){  
 *//本次划分使用的内存值* int Lmin=s[i-j]+j\*(*minBit*(*maxNumber*(dots,i-j+1,i)))+11;  
 if(Lmin<s[i]){  
 s[i]=Lmin;  
 }  
 }  
 }  
  
 System.*out*.println("最优序列为：");  
 for(int j = 0; j <= n; j++) {  
 System.*out*.print(s[j] + "\t");  
 }  
 System.*out*.println();  
  
 *//返回s数组最后一个值* return s[s.length-1];  
 }  
 private static int maxNumber(int[] dots, int start, int end){  
 int[] copy = Arrays.*copyOfRange*(dots,start -1,end);  
 Arrays.*sort*(copy);  
 return copy[copy.length -1];  
 }  
 */\*\*  
 \* 求解数字number所需要的最小二进制位数  
 \* @param number  
 \* @return  
 \*/* private static int minBit(int number){  
 int min = 0;  
 *//最多8位* for(int i = 1; i <= 8; i++){  
 if(Math.*pow*(2,i-1) - 1 <= number && Math.*pow*(2,i) - 1 >= number){  
 min = i;  
 break;  
 }  
 }  
 return min;  
 }  
  
}

实验结果如图4所示，在图像压缩中，我们规定，在分段的时候，每段的像素个数不能超过256，每个像素的位数不能超过8位，即在分段的时候，每个段的段头需要有8+3=11位二进制位进行记录每段的信息。利用一维数组s记录每段的最优值，在计算过程中，需找到每段的最大像素点的位数，保证每段的像素都是相同的位数。所以，递推公式为：s[i]=min{s[i-k]+k\*bmax(i-k+1,i)}+11，1<=k<=min{i，256}，最后得到结果。

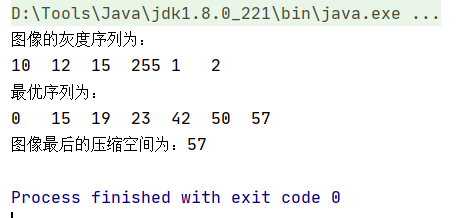


图4

## 思考问题：

1、深刻理解动态规划与递归求解问题的区别？

答：动态规划是自底向上进行求解，其基本要素有最优子结构性质和重叠子问题性质，在动态规划中，将子问题的结果用数组存储起来，当后面要使用该子问题的解的时候不需要重读计算，直接读取存储值即可。而递归是自上而下进行求解，将大问题转换为小问题，最后合并得到求解，但对于重复问题需要重复求解。

2、动态规划思想解题的步骤？

答：（1）找出最优解的性质，并刻画其结构特征；（2）递归地定义最优解；（3）以自底向上的方式计算最优值；（4）根据计算最优值时得到的信息构造最优解。