|  |
| --- |
| **斐波那契数列**  public static int solutionFibonacci(int n){  if(n==0){  return 0;  }else if(n == 1){  return 1;  }else{  int result[] = new int[n+1];  result[0] = 0;  result[1] = 1;  for(int i=2;i<=n;i++){  result[i] = result[i-1] + result[i-2];  }  return result[n];  } |
| **跳台阶问题/填充长方体问题：将一个2\*1的长方体填充到2\*n的长方体中,有几种方法**  public int getResultByDP(int n){  if (n <1) {  return 0;  }  if (n == 1){  return 1;  }  if (n == 2){  return 2;  }  int result[] = new int[n+1];  result[0] = 0;  result[1] = 1;  result[2] = 2;  for (int i = 3; i < n+1 ; i++) {  result[i] = result[i-1] + result[i-2];  }  return result[n];  }   已知从0级到9级的走法有M种，从0级到8级的走法有N种，那么思考一下，从0到10级的走法和M、N有什么关系呢？从0到10级的走法一共是多少种呢？答案是M+N。  也就是说，用F(n)来表示从0级到n级的走法，可得出  F(9)=M；  F(8)=N；  F(10)=M+N；  F(10)=F(9)+F(8); |

1. **数组最大不连续递增子序列**

arr[] = {3,1,4,1,5,9,2,6,5}的最长递增子序列长度为4。即为：1,4,5,9

设置一个数组temp，长度为原数组长度，数组第i个位置上的数字代表0...i上最长递增子序列，当增加一个数字时，最大递增子序列可能变成前面最大的递增子序列+1，也可能就是前面最大递增子序列，这需要让新增加进来的数字arr[i]跟前面所有数字比较大小，即当 arr[i] > arr[j]，temp[i] = max{temp[j]}+1，其中，j 的取值范围为：0,1...i-1，当 arr[i] < arr[j]，temp[i] = max{temp[j]}，j 的取值范围为：0,1...i-1，所以在状态转换方程为temp[i]=max{temp[i-1], temp[i-1]+1}

|  |
| --- |
| public static int MaxChildArrayOrder(int a[]) {  int n = a.length;  int temp[] = new int[n];//temp[i]代表0...i上以i结尾的最长递增子序列  for(int i=0;i<n;i++){  temp[i] = 1;//初始值都为1  }  for(int i=1;i<n;i++){  for(int j=0;j<i;j++){  if(a[i]>a[j]&&temp[j]+1>temp[i]){  //如果有a[i]比它前面所有的数都大，则temp[i]为它前面的比它小的数的那一个temp+1取得的最大值  temp[i] = temp[j]+1;  }  }  }  int max = temp[0];  //从temp数组里取出最大的值  for(int i=1;i<n;i++){  if(temp[i]>max){  max = temp[i];  }  }  return max;  } |
| 例： 3，1，4，1，5，9，  输出：4（1，4，5，9）  i=1 j=0 a[i]=1>a[j]=3（不成立）  i=2 j=0 a[i]=4>a[j]=3 temp[j]+1=2>temp[i]=1 temp[i]=temp[j]+1=2 (112111)  i=2 j=1 a[i]=4>a[j]=1 temp[j]+1=2>temp[i]=2(不成立)  i=3 j=0 a[i]=1>a[j]=3(不成立)  i=3 j=1 a[i]=1>a[j]=1(不成立)  i=3 j=2 a[i]=1>a[j]=4(不成立)  i=4 j=0 a[i]=5>a[j]=3 temp[j]+1=2>temp[i]=1 temp[i]=temp[j]+1=2 (112121)  i=4 j=1 a[i]=5>a[j]=1 temp[j]+1=2>temp[i]=2（不成立）  i=4 j=2 a[i]=5>a[j]=4 temp[j]+1=3>temp[i]=2 temp[i]=temp[j]+1=3 (112131)  i=4 j=3 a[i]=5>a[j]=1 temp[j]+1=2>temp[i]=3(不成立)  i=5 j=0 a[i]=9>a[j]=3 temp[j]+1=2>temp[i]=1 temp[i]=temp[j]+1=2 (112132)  i=5 j=1 a[i]=9>a[j]=1 temp[j]+1=2>temp[i]=2(不成立)  i=5 j=2 a[i]=9>a[j]=4 temp[j]+1=3>temp[i]=2 temp[i]=temp[j]+1=3 (112133)  i=5 j=3 a[i]=9>a[j]=1 temp[j]+1=2>temp[i]=3(不成立)  i=5 j=4 a[i]=9>a[j]=5 temp[j]+1=4>temp[i]=3 temp[i]=temp[j]+1=4 (112143) |
| public class Solution {  public int lengthOfLIS(int[] nums) {  if (nums.length == 0) {  return 0;  }  int[] dp = new int[nums.length];  dp[0] = 1; 以num[i]结尾的最长长度  int maxans = 1;  for (int i = 1; i < dp.length; i++) { //遍历第二个数到最后  int maxval = 0; //每遍历一个数都清0  for (int j = 0; j < i; j++) { //遍历判断的数字前面的数字  if (nums[i] > nums[j]) { //如果num[i]大于前面的数字  maxval = Math.max(maxval, dp[j]); 找dp中的最大值  }  }  dp[i] = maxval + 1; //因为自己本身要算，所以不论大小都加1  //如果有比他小的则找出前面的比他小的dp中最大的  maxans = Math.max(maxans, dp[i]); 找dp中的最大值  }  return maxans;  }  } |
| 3 1 4 1 5 9 2 6 5  1 1 1+1(接1/3) 1 2+1(接4) 3+1(接5) 1+1(接1) 3+1(接5) 2+1(接4/2)  Num[i]前面的数字中比他小的数字的dp中最大的值加一（他本身）  dp[i]是以num[i]为结尾的最长子序列的长度 |

**3.** **数组最大连续子序列和**

如arr[] = {6,-1,3,-4,-6,9,2,-2,5}的最大连续子序列和为14。即为：9,2,-2,5

创建一个数组a，长度为原数组长度，不同位置数字a[i]代表0...i上最大连续子序列和，a[0]=arr[0]设置一个最大值max，初始值为数组中的第一个数字。当进来一个新的数字arr[i+1]时，判断到他前面数字子序列和a[i]+arr[i+1]跟arr[i+1]哪个大，前者大就保留前者，后者大就说明前面连续数字加起来都不如后者一个新进来的数字大，前面数字就可以舍弃，从arr[i+1]开始，每次比较完都跟max比较一下，最后的max就是最大值。

|  |
| --- |
| public static int MaxContinueArraySum(int a[]) {  int n = a.length;  int max = a[0];  int sum = a[0];  for(int i=1;i<n;i++){  sum = Math.max(sum+a[i], a[i]);  if(sum>=max){  max = sum;  }  }  return max;  } |

**4、数字塔从上到下所有路径中和最大的路径**

数字塔是第i行有i个数字组成，从上往下每个数字只能走到他正下方数字或者右下方数字，求数字塔从上到下所有路径中和最大的路径，如有下数字塔

3

1    5

8    4    3

2    6    7    9

6    2    3    5    1

最大路径是3-5-3-9-5，和为25。我们可以分别从从上往下看跟从下往上看两种动态规划的方式去解这个题

从上往下看：当从上往下看时，每进来新的一行，新的一行每个元素只能选择他正上方或者左左方的元素，也就是说，第一个元素只能连他上方的元素，最后一个元素只能连他左上方的元素，其他元素可以有两种选择，所以需要选择加起来更大的那一个数字，并把这个位置上的数字改成相应的路径值，具体过程如下图所示

|  |
| --- |
| 3 3 3 3  1 5 4 8 4 8 4 8  8 4 3 8 4 3 12 12 11 12 12 11  2 6 7 9 2 6 7 9 2 6 7 9 14 18 19 20  6 2 3 5 1 6 2 3 5 1 6 2 3 5 1 20 20 22 25 21 |

所以最大值就是最底层的最大值也就是25。

具体运算过程就是，建立一个n\*n的二维数组dp[][]，n是数字塔最后一行的数字个数，二维数组每一行数字跟数字塔每一行数字个数一样，保存的值是从上方到这一个位置最大路径的值，填入边界值dp[0][0]=3，每一行除了第一个值跟最后一个值，其他的值选择上方或者左上方更大的值与这个位置上的值相加得来的值，即dp[i][j]=Math.max(dp[i-1][j-1], dp[i-1][j]) + n[i][j]

|  |
| --- |
| public static int minNumberInRotateArray(int n[][]) {  int max = 0;  int dp[][] = new int[n.length][n.length];  dp[0][0] = n[0][0];  for(int i=1;i<n.length;i++){  for(int j=0;j<=i;j++){  if(j==0){  //如果是第一列，直接跟他上面数字相加  dp[i][j] = dp[i-1][j] + n[i][j];  }else{  //如果不是第一列，比较他上面跟上面左面数字谁大，谁大就跟谁相加，放到这个位置  dp[i][j] = Math.max(dp[i-1][j-1], dp[i-1][j]) + n[i][j];  }  max = Math.max(dp[i][j], max);  }  }  return max; |

优化：动态规划中每一个需要创建一个二维数组的解法，都可以换成只创建一个一维数组的滚动数组解法，依据的规则是一般二维数组中存放的是所有的结果，但是一般我们需要的结果实在二维数组的最后一行的某个值，前面几行的值都是为了得到最后一行的值而需要的，所以可以开始就创建跟二维数组最后一行一样大的一维数组，每次存放某一行的值，下一次根据这一行的值算出下一行的值，在存入这个数组，也就是把这个数组滚动了，最后数组存储的结果就是原二维数组中最后一行的值。

拿到本题来说，开始创建一个一维数组dp[n]，初始值只有dp[0]=3，新进来一行时，仍然遵循dp[i][j]=Math.max(dp[i-1][j-1], dp[i-1][j]) + n[i][j]，现在为求dp[j]，所以现在dp[i-1][j]其实就是数组中这个位置本来的元素即dp[j]，而dp[i-1][j-1]其实就是数组中上一个元素dp[j-1]，也就是说dp[j]=Math.max(dp[j], dp[j-1])+n[i][j]

|  |
| --- |
| public static int minNumberInRotateArray2(int n[][]) {  int[] temp = new int[n.length];  temp[0] = n[0][0];  for(int i=1;i<n.length;i++){  for(int j=i;j>=0;j--){  if(j==i){  temp[i]=temp[i-1]+n[i][j];  }else if(j==0){  temp[0]+=n[i][0];  }else{  temp[j]=Math.max(temp[j], temp[j-1])+n[i][j];  }  }  }  int max = temp[0];  //从temp数组里取出最大的值  for(int i=1;i<temp.length;i++){  if(temp[i]>max){  max = temp[i];  }  }  return max;  } |

从下往上看时：从下往上看时大体思路跟从上往下看一样，但是要简单一些，因为不用考虑边界数据，从下往上看时，每进来上面一行，上面一行每个数字有两条路径到达下面一行，所以选一条最大的就可以

|  |
| --- |
| 3                            3                            3                            25  1    5                      1    5                      1    5                     18   22  8    4    3                 8    4    3                  17  16  17               17  16  17  2    6    7    9           8    9    12  14          8    9   12  14          8    9    12  14  6    2    3    5    1     6    2    3    5    1      6    2    3    5    1    6    2    3    5    1 |

所以最大值就是最上面数字就是25.

具体方法也是建立一个二维数组，最下面一行数据添到二维数组最后一行，从下往上填数字，所以状态转化方程是dp[i][j]=Math.max(dp[i+1][j+1], dp[i+1][j]) + n[i][j]，具体解决方法跟从上往下看一样，就不写具体代码了。

|  |
| --- |
| public static int minNumberInRotateArray(int n[][]) {  int max = 0;  int dp[][] = new int[n.length][n.length];  for(int i=0;i<n.length;i++){  dp[n.length][i]=n[n.length][i];  }  for(int i=n.length-2;i>=0;i--){  for(int j=0;j<I;j++){  dp[i][j]= Math.max(dp[i+1][j+1], dp[i+1][j]) + n[i][j];  }  }  return dp[0][0];  } |

优化：滚动数组，只创建一个一维数组，数组初始值是数字塔最下面一行的值，每次新加一行值，将数组中的值改变，最后数组中第一个数字就是最大路径的值。状态转化方程就是temp[j] = Math.max(temp[j], temp[j+1])+n[i][j]。具体代码如下

|  |
| --- |
| public static int minNumberInRotateArray3(int n[][]) {  int[] temp = new int[n.length];  for(int i=0;i<n.length;i++){  temp[i] = n[n.length-1][i];  }  for(int i=n.length-2;i>=0;i--){  for(int j=0;j<=i;j++){  temp[j] = Math.max(temp[j], temp[j+1])+n[i][j];  }  }  return temp[0];  } |

从下往上看跟从上往下看相比，虽然逻辑较为简单，但是从下往上看时需要得到完整的数字塔之后才能开始计算，而从上往下看时可以随着数字塔的深入来计算，也可以返回任意一层的结果，是最好的方法。

**5、两个字符串最大公共子序列**

比如字符串1：BDCABA；字符串2：ABCBDAB，则这两个字符串的最长公共子序列长度为4，最长公共子序列是：BCBA

具体思想：设 X=(x1,x2,.....xn)和 Y={y1,y2,.....ym} 是两个序列，将 X 和 Y 的最长公共子序列记为LCS(X,Y)，如果 xn=ym，即X的最后一个元素与Y的最后一个元素相同，这说明该元素一定位于公共子序列中。因此，现在只需要找：LCS(Xn-1，Ym-1)就好，LCS(X,Y)=LCS(Xn-1，Ym-1)+1；如果xn != ym，这下要麻烦一点，因为它产生了两个子问题：LCS(Xn-1，Ym) 和 LCS(Xn，Ym-1)。

动态规划解法：先创建一个解空间即数组，因为给定的是两个字符串即两个一维数组存储的数据，所以要创建一个二维数组，设字符串X有n个值，字符串Y有m个值，需要创建一个m+1\*n+1的二维数组，二维数组每个位置（i，j）代表当长度为i的X子串与长度为j的Y的子串他们的最长公共子串，之所以要多创建一个是为了将边界值填入进去，边界值就是第一行跟第一列，指X长度为0或者Y长度为0时，自然需要填0，其他位置填数字时，当这**两个位置数字相同，dp[i][j] = dp[i-1][j-1]+1**；当这**两个位置数字不相同时，dp[i][j] = Math.max(dp[i][j-1], dp[i-1][j])。**最后二维数组最右下角的值就是最大子串。

|  |
| --- |
| public class MaxTwoArraySameOrder {  public static int MaxTwoArraySameOrderMethod(String str1,String str2) {  int m = str1.length();  int n = str2.length();  /\*  \* 定义一个二维数组保存公共子序列长度  \* dp[i][j]表示字符串1从头开始长度是i，字符串2从头开始长度是j，这两个字符串的最长公共子序列的长度  \* 设置数组行列比他们长度大一往二维数组中填写数字时，每个位置的数字跟他上方或者左方或者左上方数字有关系，这样处理边界数字时不用处理这种情况，方便接下来的循环  \*/  int dp[][] = new int[m+1][n+1];  /\*  \* 初始化第一行第一列  \* dp[0,j]表示啥？表示字符串1的长度是0，字符串2的长度是j，这两个字符串的最长公共子序列的长度是0，因为，字符串1 根本就没有嘛  \*/  for(int i=0;i<=m;i++){  dp[i][0] = 0;  }  for(int i=0;i<=n;i++){  dp[0][i] = 0;  }  for(int i=1;i<=m;i++){  for(int j=1;j<=n;j++){  /\*  \* 如果当c[i][j]时，字符串1从头开始长度是i，字符串2从头开始长度是j时他们最后一个字符相同  \* 就同时把他们向前移动一位，找c[i-1][j-1]时长度最大的再加一  \* 表现在二维数组中就是c[i][j]左上方的点  \*/  if(str1.charAt(i-1) == str2.charAt(j-1)){  dp[i][j] = dp[i-1][j-1]+1;  /\*  \* 如果当c[i][j]时，他们最后一个字符不相同  \* 要将str1往前移动一位的c[i-1][j]的lcs长度，或者将str2往前移动一位的c[i][j-1]的lcs长度  \* 哪个长，将它赋给c[i][j]  \* 表现在二维数组中就是c[i][j]上方的点或者左方的点  \*/  }else{  dp[i][j] = Math.max(dp[i][j-1], dp[i-1][j]);  }  }  }  return dp[m][n];  }  public static void main(String[] args) {  String str1 = "BDCABA";  String str2 = "ABCBDAB";  int array = MaxTwoArraySameOrderMethod(str1,str2);  System.out.println(array);  }    } |

6、背包问题

在N件物品取出若干件放在容量为W的背包里，每件物品的体积为W1，W2……Wn（Wi为整数），与之相对应的价值为P1,P2……Pn（Pi为整数），求背包能够容纳的最大价值。

动态规划的解法就是：创建一个二维数组，横坐标是从1开始到W，纵坐标是组成W的各种元素，本题中就是指W1，W2……Wn，数组中每个位置（i，j）的数字就是当组成元素只有W1，W2……Wi，背包可放容量为j时的结果，本题中就是容纳的最大价值。所以很容易分析出，当（i，j）时，如果Wi能放的下，空间减小，但是会增加Pi的价值，如果Wi不能放的下，空间不变，是（i-1，j）的价值，取其中最大值就好了，即状态转化方程为能放的下，dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-w[i]]+p[i])；放不下，dp[i][j] = dp[i-1][j]；

|  |
| --- |
| import java.util.\*;  public class DynamicProgramming {  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  while (sc.hasNext()) { /\* 1.读取数据 \*/  int number = sc.nextInt(); // 物品的数量  // 注意：我们声明数组的长度为"n+1",并另score[0]和time[0]等于0。  // 从而使得 数组的下标，对应于题目的序号。即score[1]对应于第一题的分数,time[1]对应于第一题的时间  int[] weight = new int[number + 1]; // {0,2,3,4,5} 每个物品对应的重量  int[] value = new int[number + 1]; // {0,3,4,5,6} 每个物品对应的价值  weight[0] = 0;  for (int i = 1; i < number + 1; i++) {  weight[i] = sc.nextInt();  }  value[0] = 0;  for (int i = 1; i < number + 1; i++) {  value[i] = sc.nextInt();  }  int capacity = sc.nextInt(); // 背包容量  /\* 2.求解01背包问题 \*/  int[][] v = new int[number + 1][capacity + 1];// 声明动态规划表.其中v[i][j]对应于：当前有i个物品可选，并且当前背包的容量为j时，我们能得到的最大价值  // 填动态规划表。当前有i个物品可选，并且当前背包的容量为j。  for (int i = 0; i < number + 1; i++) {  for (int j = 0; j < capacity + 1; j++) {  if (i == 0) {  v[i][j] = 0; // 边界情况：若只有0道题目可以选做，那只能得到0分。所以令V(0,j)=0  } else if (j == 0) {  v[i][j] = 0; // 边界情况：若只有0分钟的考试时间，那也只能得0分。所以令V(i,0)=0  } else {  if (j < weight[i]) {  v[i][j] = v[i - 1][j];// 包的容量比当前该物品体积小，装不下，此时的价值与前i-1个的价值是一样的，即V(i,j)=V(i-1,j)；  } else {  v[i][j] = Math.max(v[i - 1][j], v[i - 1][j - weight[i]] + value[i]);// 还有足够的容量可以装当前该物品，但装了当前物品也不一定达到当前最优价值，所以在装与不装之间选择最优的一个，即V(i,j)=max｛V(i-1,j)，V(i-1,j-w(i))+v(i)｝。  }  }  }  }  System.out.println(“输出动态规划表”);  System.out.println("动态规划表如下：");  for (int i = 0; i < number + 1; i++) {  for (int j = 0; j < capacity + 1; j++) {  System.out.print(v[i][j] + "\t");  }  System.out.println();  }  System.out.println("背包内最大的物品价值总和为：" + v[number][capacity]);  // 有number个物品可选，且背包的容量为capacity的情况下，能装入背包的最大价值  /\* 3.价值最大时，包内装入了哪些物品？ \*/  int[] item = new int[number + 1];// 下标i对应的物品若被选中，设置值为1  Arrays.fill(item, 0);// 将数组item的所有元素初始化为0    // 从最优解，倒推回去找  int j = capacity;  for (int i = number; i > 0; i--) {  if (v[i][j] > v[i - 1][j]) {// 在最优解中，v[i][j]>v[i-1][j]说明选择了第i个商品  item[i] = 1;  j = j - weight[i];  }  }    System.out.print("包内物品的编号为：");  for (int i = 0; i < number + 1; i++) {  if (item[i] == 1) {  System.out.print(i + " ");  }  }  System.out.println("----------------------------");    }    }  } |

|  |
| --- |
| public static int PackageHelper2(int n,int w[],int p[],int v) {  //设置一个二维数组，横坐标代表从第一个物品开始放到第几个物品，纵坐标代表背包还有多少容量，dp代表最大价值  int dp[] = new int[v+1];  for(int i=1;i<=n;i++){  for(int j=v;j>0;j--){  if(j>w[i]){  dp[j] = Math.max(dp[j], dp[j-w[i]]+p[i]);  }else{  dp[j] = dp[j];  }  }  }  return dp[v];  } |

0-1背包问题求方案数

7、找零钱问题：有几种方法

具体思路同背包问题，这里只分析一下动态转化方程，能用这种零钱，分为用了这种零钱的方法跟没用到这种零钱的方法，dpi = dpi-1 + dpi]；如果不能用这种零钱，即所组成的面额小于当前零钱，直接等于不用这种零钱的数值，dpi = dpi-1。这里要特别注意的是。1、开始填写二维数组边界值时，第一行是填写只用第一种面额零钱组成相应数额的方法，要注意是总数额除以第一种面额取余为0才能组成，即如果第一种面额为2，不能组成3,5的数额等；2、填写二维数组第一列时，代表到用到面额为i时，剩余数额为0，即只用i就可以组成相应数额，这也是一种方法，所以第一列的值，第一个为0，后面全为1.

|  |
| --- |
| public static int SmallMoney(int num[],int target) {  int m = num.length;  int dp[][] = new int[m][target+1];  dp[0][0] = 1;  for(int i=1;i<=target;i++){  if(i%num[0] == 0){  dp[0][i] = 1;//第一行数值填写  }else{  dp[0][i] = 0;  }  }  for(int i=0;i<m;i++){  dp[i][0] = 1;//第一列数值填写  }  for(int i=1;i<m;i++){  for(int j=1;j<=target;j++){  if(j<num[i]){  dp[i][j] = dp[i-1][j];  }else{  dp[i][j] = dp[i-1][j] + dp[i][j-num[i]];  }  }  }  return dp[m-1][target];  } |

优化：动态数组，同背包问题即以上分析

|  |
| --- |
| public static int SmallMoney2(int num[],int target) {  int m = num.length;  int dp[] = new int[target+1];  dp[0] = 1;  for(int i=1;i<=target;i++){  if(i%num[0] == 0){  dp[i] = 1;  }else{  dp[i] = 0;  }  }  for(int i=1;i<m;i++){  for(int j=1;j<=target;j++){  if(j>=num[i]){  dp[j] = dp[j] + dp[j-num[i]];  }  }  }  return dp[target];  } |