# 动态规划

已知问题规模为n的前提A，求解一个未知解B。（我们用An表示“问题规模为n的已知条件”）

此时，如果把问题规模降到0，即已知A0，可以得到A0->B.

如果从A0添加一个元素，得到A1的变化过程。即A0->A1; 进而有A1->A2; A2->A3; …… ; Ai->Ai+1. 这就是严格的归纳推理，也就是我们经常使用的数学归纳法；

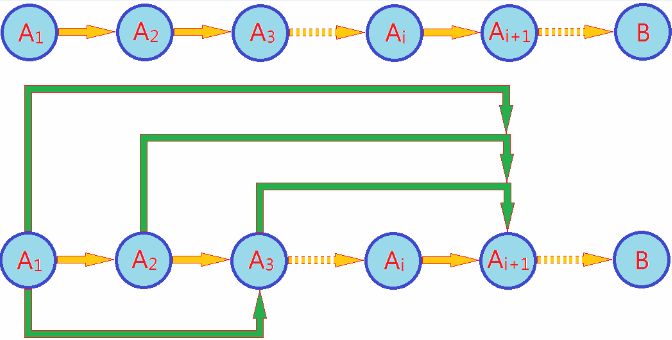
对于Ai+1，只需要它的上一个状态Ai即可完成整个推理过程（而不需要更前序的状态）。我们将这一模型称为马尔科夫模型。对应的推理过程叫做“贪心法”。

然而，Ai与Ai+1往往不是互为充要条件，随着i的增加，有价值的前提信息越来越少，我们无法仅仅通过上一个状态得到下一个状态，因此可以采用如下方案：

{A1->A2}; {A1, A2->A3}; {A1,A2,A3->A4};……; {A1,A2,...,Ai}->Ai+1. 这种方式就是第二数学归纳法。

对于Ai+1需要前面的所有前序状态才能完成推理过程。我们将这一模型称为高阶马尔科夫模型。对应的推理过程叫做“动态规划法”。

上述两种状态转移图如下图所示



能采用动态规划求解的问题的一般要具有3个性质：

1. 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。
2. 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。
3. 有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决

动规解题的一般思路

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤。

初始状态→│决策１│→│决策２│→…→│决策ｎ│→结束状态

(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。

实际应用中可以按以下几个简化的步骤进行设计：

（1）分析最优解的性质，并刻画其结构特征。

（2）递归的定义最优解。

（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（4）根据计算最优值时得到的信息，构造问题的最优解

算法实现的说明

 动态规划的主要难点在于理论上的设计，也就是上面4个步骤的确定，一旦设计完成，实现部分就会非常简单。

 使用动态规划求解问题，最重要的就是确定动态规划三要素：

1. 问题的阶段

（2）每个阶段的状态

 （3）从前一个阶段转化到后一个阶段之间的递推关系。

递推关系必须是从次小的问题开始到较大的问题之间的转化，从这个角度来说，动态规划往往可以用递归程序来实现，不过因为递推可以充分利用前面保存的子问题的解来减少重复计算，所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。

确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，从1行1列开始，以行或者列优先的顺序，依次填写表格，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。

          f(n,m)=max{f(n-1,m), f(n-1,m-w[n])+P(n,m)}

算法实现的步骤

1、创建一个一维数组或者二维数组，保存每一个子问题的结果，具体创建一维数组还是二维数组看题目而定，基本上如果题目中给出的是一个一维数组进行操作，就可以只创建一个一维数组，如果题目中给出了两个一维数组进行操作或者两种不同类型的变量值，比如背包问题中的不同物体的体积与总体积，找零钱问题中的不同面值零钱与总钱数，这样就需要创建一个二维数组。

注：需要创建二维数组的解法，都可以创建一个一维数组运用滚动数组的方式来解决，即一位数组中的值不停的变化，后面会详细徐叙述

2、设置数组边界值，一维数组就是设置第一个数字，二维数组就是设置第一行跟第一列的值，特别的滚动一维数组是要设置整个数组的值，然后根据后面不同的数据加进来变幻成不同的值。

3、找出状态转换方程，也就是说找到每个状态跟他上一个状态的关系，根据状态转化方程写出代码。

4、返回需要的值，一般是数组的最后一个或者二维数组的最右下角。

代码基本框架：

for(j=1; j<=m; j=j+1) // 第一个阶段

xn[j] = 初始值;

for(i=n-1; i>=1; i=i-1)// 其他n-1个阶段

for(j=1; j>=f(i); j=j+1)//f(i)与i有关的表达式

xi[j]=j=max（或min）{g(xi-[j1:j2]), ......, g(xi-1[jk:jk+1])};

t = g(x1[j1:j2]); // 由子问题的最优解求解整个问题的最优解的方案

print(x1[j1]);

for(i=2; i<=n-1; i=i+1）

{

t = t-xi-1[ji];

for(j=1; j>=f(i); j=j+1)

if(t=xi[ji])

break;

}

下面通过几个典型例子，从简单到难帮助我们理解动态规划。

斐波那契数列大家都很熟悉，而且知道用递归可以很容易的做出来

if(n == 0){

return 0;

}else if(n == 1){

return 1;

}else{

return solutionFibonacci(n-1)+solutionFibonacci(n-2);

}

如果用动态规划，就是把结果存到一个数组中

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];

}

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;

}

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正右方数字，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;

}

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。

像这种固定数值的组合问题，比如这个问题的W总容量，跟下个实例零钱问题的总钱数，都是适合用动态规划来解决的问题，对于这样的问题，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]

<https://www.cnblogs.com/Christal-R/p/Dynamic_programming.html>

public static int PackageHelper(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[n+1][v+1];

for(int i=1;i<n+1;i++){

for(int j=1;j<=v;j++){

if(j>=w[i]){

/\*

\* 当能放得下这个物品时，放下这个物品，价值增加，但是空间减小，最大价值是dp[i-1][j-w[i]]+p[i]

\* 当不放这个物品时，空间大，物品还是到i-1，最大价值是dp[i-1][j]

\* 比较这两个大小，取最大的，就是dp[i][j]

\*/

dp[i][j] = Math.max(dp[i-1][j], dp[i-1][j-w[i]]+p[i]);

}else{

//如果放不下，就是放上一个物品时的dp

dp[i][j] = dp[i-1][j];

}

}

}

return dp[n][v];

}

虽然上面算法已求得最优解，但无法知道是有哪些物品组成，为求哪些物品构成最优解需回溯查找

void FindWhat(int i,int j)//寻找解的组成方式

{

if(i>=0)

{

if(V[i][j]==V[i-1][j])//相等说明没装

{

item[i]=0;//全局变量，标记未被选中

FindWhat(i-1,j);

}

else if( j-w[i]>=0 && V[i][j]==V[i-1][j-w[i]]+v[i] )

{

item[i]=1;//标记已被选中

FindWhat(i-1,j-w[i]);//回到装包之前的位置

}

}

}

# 分治算法

一、基本概念

　　在计算机科学中，分治法是一种很重要的算法。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换(快速傅立叶变换)……

　　任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于n个元素的排序问题，当n=1时，不需任何计算。n=2时，只要作一次比较即可排好序。n=3时只要作3次比较即可，…。而当n较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

二、基本思想及策略

　　分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

　　分治策略是：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

　　如果原问题可分割成k个子问题，1<k≤n，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

三、分治法使用场景

分治法所能解决的问题一般具有以下几个特征：

1) 该问题的规模缩小到一定的程度就可以容易地解决

2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

3) 利用该问题分解出的子问题的解可以合并为该问题的解；

4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；

第二条特征是应用分治法的前提它也是大多数问题可以满足的，此特征反映了递归思想的应用；

第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。

第四条特征涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

四、分治法得基本步骤

分治法在每一层递归上都有三个步骤：

step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

step3 合并：将各个子问题的解合并为原问题的解。

它的一般的算法设计模式如下：

Divide-and-Conquer(P)

1. if |P|≤n0

2. then return(ADHOC(P))

3. 将P分解为较小的子问题 P1 ,P2 ,…,Pk

4. for i←1 to k

5. do yi ← Divide-and-Conquer(Pi) △ 递归解决Pi

6. T ← MERGE(y1,y2,…,yk) △ 合并子问题

7. return(T)

其中|P|表示问题P的规模；n0为一阈值，表示当问题P的规模不超过n0时，问题已容易直接解出，不必再继续分解。ADHOC(P)是该分治法中的基本子算法，用于直接解小规模的问题P。因此，当P的规模不超过n0时直接用算法ADHOC(P)求解。算法MERGE(y1,y2,…,yk)是该分治法中的合并子算法，用于将P的子问题P1 ,P2 ,…,Pk的相应的解y1,y2,…,yk合并为P的解。

五、分治法的复杂性分析

一个分治法将规模为n的问题分成k个规模为n／m的子问题去解。设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需用f(n)个单位时间。用T(n)表示该分治法解规模为|P|=n的问题所需的计算时间，则有：

T（n）= k T(n/m)+f(n)

通过迭代法求得方程的解：

递归方程及其解只给出n等于m的方幂时T(n)的值，但是如果认为T(n)足够平滑，那么由n等于m的方幂时T(n)的值可以估计T(n)的增长速度。通常假定T(n)是单调上升的，从而当mi≤n<mi+1时，T(mi)≤T(n)<T(mi+1)。

六、可使用分治法求解的一些经典问题

（1）二分搜索

（2）大整数乘法

（3）Strassen矩阵乘法

（4）棋盘覆盖

（5）合并排序

（6）快速排序

（7）线性时间选择

（8）最接近点对问题

（9）循环赛日程表

（10）汉诺塔

二分搜索

public static void main(String[] args) {

int[] a = {1,2,3,4,5,6,7,8,9};

int pos =bSearch(a, 0, a.length-1, 1);

System.out.println(pos);

}

public static int bSearch(int[] data,int left,int right,int key){

//获取中间位置

int middle = (left+right)/2;

//比较key值如相等，返回当前位置，否则确认新的查找空间

if(data[middle] == key){

return middle;

}else if(data[middle] >key){

return bSearch(data, left, middle-1, key);

}else{

return bSearch(data, middle+1, right, key);

}

}

棋盘覆盖

public class ChessboardCover {

static int board[][];

static int tile=1; //L型骨牌序号，根据填入的顺序给L型骨牌编号

/\*

\* tr：棋盘左上角方格的行号；tc：棋盘左上角方格的列号

\* dr：特殊方格的行号；dc：特殊方格的列号

\* size：棋盘的大小是size×size

\*/

static void ChessBoard(int tr,int tc,int dr,int dc,int size){

if(size==1)

return;

int t=tile++;

int s=size/2; //分割棋盘

//覆盖左上角棋盘

if(dr<tr+s && dc<tc+s)

ChessBoard(tr,tc,dr,dc,s); //特殊方格在此棋盘中

else{ //此棋盘无特殊方格

board[tr+s-1][tc+s-1]=t; //用t号L型骨牌覆盖此子棋盘的右下角

ChessBoard(tr,tc,tr+s-1,tc+s-1,s);

}

//覆盖右上角棋盘

if(dr<tr+s && dc>=tc+s)

ChessBoard(tr,tc+s,dr,dc,s);

else{

board[tr+s-1][tc+s]=t; //用t号L型骨牌覆盖此子棋盘的左下角

ChessBoard(tr,tc+s,tr+s-1,tc+s,s);

}

//覆盖左下角棋盘

if(dr>=tr+s && dc<tc+s)

ChessBoard(tr+s,tc,dr,dc,s);

else{

board[tr+s][tc+s-1]=t; //用t号L型骨牌覆盖此棋盘的右上角

ChessBoard(tr+s,tc,tr+s,tc+s-1,s);

}

//覆盖右下角棋盘

if(dr>=tr+s && dc>=tc+s)

ChessBoard(tr+s,tc+s,dr,dc,s);

else{

board[tr+s][tc+s]=t; //用t号L型骨牌覆盖此棋盘左上角

ChessBoard(tr+s,tc+s,tr+s,tc+s,s);

}

}

public static void main(String[] args) {

int k=3; //棋盘大小是2^k×2^k

int size=(int)Math.pow(2, k);

board=new int[size][size];

ChessBoard(0,0,0,1,size);

for(int i=0;i<size;i++){

for(int j=0;j<size;j++){

System.out.printf("%4d",board[i][j]);

}

System.out.println();

}

}

}

# 回溯

回溯算法也叫试探法，它是一种系统地搜索问题的解的方法。

用回溯算法解决问题的一般步骤：

1、针对所给问题，定义问题的解空间，它至少包含问题的一个（最优）解。

2、确定易于搜索的解空间结构,使得能用回溯法方便地搜索整个解空间 。

3、以深度优先的方式搜索解空间，并且在搜索过程中用剪枝函数避免无效搜索。

确定了解空间的组织结构后，回溯法就从开始结点（根结点）出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前扩展结点就成为死结点。此时，应往回移动（回溯）至最近的一个活结点处，并使这个活结点成为当前的扩展结点。回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已没有活结点时为止。

实例一：八皇后问题

八皇后问题是一个古老而著名的问题，是回溯算法的典型例题。该问题是十九世纪著名的数学家高斯1850年提出：在8X8格的国际象棋上摆放八个皇后（棋子），使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。

bool is\_ok(int row){ //判断设置的皇后是否在同一行，同一列，或者同一斜线上

for (int j=0;j<row;j++)

{

if (queen[row]==queen[j]||row-queen[row]==j-queen[j]||row+queen[row]==j+queen[j])

return false;

}

return true;

}

void back\_tracking(int row=0) //算法函数，从第0行开始遍历

{

if (row==n)

t ++; //判断若遍历完成，就进行计数

for (int col=0;col<n;col++) //遍历棋盘每一列

{

queen[row] = col; //将皇后的位置记录在数组

if (is\_ok(row)) //判断皇后的位置是否有冲突

back\_tracking(row+1); //递归，计算下一个皇后的位置

}

}

# 贪心算法

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关。

贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。贪心选择是采用从顶向下、以迭代的方法做出相继选择，每做一次贪心选择就将所求问题简化为一个规模更小的子问题。对于一个具体问题，要确定它是否具有贪心选择的性质，我们必须证明每一步所作的贪心选择最终能得到问题的最优解。通常可以首先证明问题的一个整体最优解，是从贪心选择开始的，而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步贪心选择，最终可得到问题的一个整体最优解

贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止

过程

建立数学模型来描述问题；

把求解的问题分成若干个子问题；

对每一子问题求解，得到子问题的局部最优解；

把子问题的解局部最优解合成原来解问题的一个解。

范例<https://www.jianshu.com/p/fede80bad3f1>

<https://blog.csdn.net/effective_coder/article/details/8736718>

### 案例一

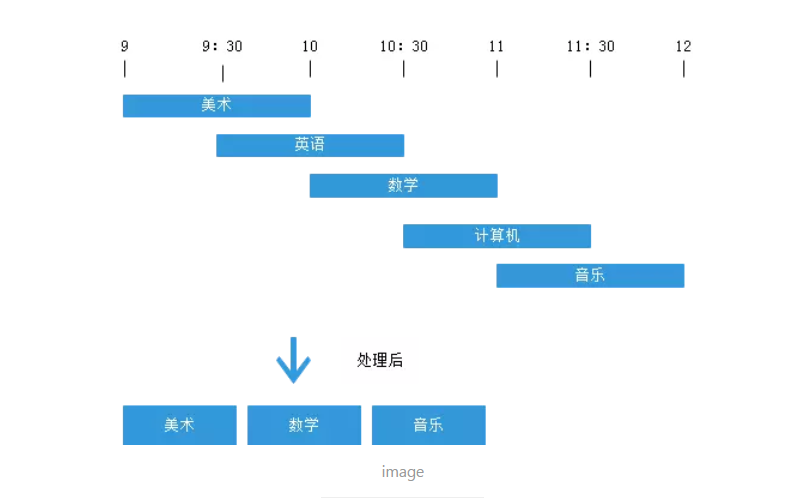
区间调度问题:

假设有如下课程，希望尽可能多的将课程安排在一间教室里：

| **课程** | **开始时间** | **结束时间** |
| --- | --- | --- |
| 美术 | 9AM | 10AM |
| 英语 | 9:30AM | 10:30AM |
| 数学 | 10AM | 11AM |
| 计算机 | 10:30AM | 11:30AM |
| 音乐 | 11AM | 12PM |

这个问题看似要思考很多，实际上算法很简单:

1. 选择**结束最早**的课，便是要在这教室上课的第一节课  
   2.接下来，选择第一堂课结束后才开始的课，并且结束最早的课，这将是第二节在教室上的课。



重复这样做就能找出答案，这边的选择策略便是结束最早且和上一节课不冲突的课进行排序，因为每次都选择结束最早的，所以留给后面的时间也就越多，自然就能排下越多的课了。

每一节课的选择都是策略内的局部最优解(留给后面的时间最多)，所以最终的结果也是近似最优解(这个案例上就是最优解)。  
(该案例的代码实现，就是一个简单的时间遍历比较过程)

### 案例二

背包问题：有一个背包，容量为35磅 ， 现有如下物品

| **物品** | **重量** | **价格** |
| --- | --- | --- |
| 吉他 | 15 | 1500 |
| 音响 | 30 | 3000 |
| 笔记本电脑 | 20 | 2000 |
| 显示器 | 29 | 2999 |
| 笔 | 1 | 200 |

要求达到的目标为装入的背包的总价值最大，并且重量不超出。

方便计算所以只有3个物品，实际情况可能是成千上万。

同上使用贪婪算法，因为要总价值最大，所以每次每次都装入最贵的,然后在装入下一个最贵的，选择结果如下：

选择: 音响 + 笔，总价值 3000 + 200 = 3200

并不是最优解: 吉他 + 笔记本电脑, 总价值 1500 + 2000 = 3500

当然选择策略有时候并不是很固定，可能是如下：

(1)每次挑选价值最大的,并且最终重量不超出：

选择: 音响 + 笔，总价值 3000 + 200 = 3200

(2)每次挑选重量最大的,并且最终重量不超出(可能如果要求装入最大的重量才会优先考虑)：

选择: 音响 + 笔，总价值 3000 + 200 = 3200

(3)每次挑选单位价值最大的(价格/重量),并且最终重量不超出：

选择: 笔+ 显示器，总价值 200 + 2999 = 3199

如上最终的结果并不是最优解，在这个案例中贪婪算法并无法得出最优解，只能得到近似最优解,也算是该算法的**局限性之一**。该类问题中需要得到最优解的话可以采取动态规划算法(后续更新，也可以关注我的公众号第一时间获取更新信息)。