（1）用计算机求解问题的步骤：

1、问题分析 2、数学模型建立 3、算法设计与选择 4、算法指标 5、算法分析 6、算法实现 7、程序调试 8、结果整理文档编制

（2）算法定义：算法是指在解决问题时，按照某种机械步骤一定可以得到问题结果的处理过程

（3）算法的三要素

1、操作 2、控制结构 3、数据结构

算法具有以下 5 个属性：

有穷性：一个算法必须总是在执行有穷步之后结束，且每一步都在有穷时间内完成。

确定性：算法中每一条指令必须有确切的含义。不存在二义性。只有一个入口和一个出口

可行性：一个算法是可行的就是算法描述的操作是可以通过已经实现的基本运算执行有限次来实现的。

输入：一个算法有零个或多个输入，这些输入取自于某个特定对象的集合。

输出：一个算法有一个或多个输出，这些输出同输入有着某些特定关系的量。

算法设计的质量指标：

正确性：算法应满足具体问题的需求；

可读性：算法应该好读，以有利于读者对程序的理解；

健壮性：算法应具有容错处理，当输入为非法数据时，算法应对其作出反应，而不是产生莫名其妙的输出结果。

效率与存储量需求：效率指的是算法执行的时间；存储量需求指算法执行过程中所需要的最大存储空间。一般这两者与问题的规模有关。

经常采用的算法主要有迭代法、分而治之法、贪婪法、动态规划法、回溯法、分支限界法

迭代法 也称“辗转法”，是一种不断用变量的旧值递推出新值的解决问题的方法。

利用迭代算法解决问题，需要做好以下三个方面的工作：

一、确定迭代模型。在可以用迭代算法解决的问题中，至少存在一个直接或间接地不断由旧值递推出新值的变量，这个变量就是迭代变量。

二、建立迭代关系式。所谓迭代关系式，指如何从变量的前一个值推出其下一个值的公式（或关系）。迭代关系式的建立是解决迭代问题的关键，通常可以使用递推或倒推的方法来完成。

三、对迭代过程进行控制。在什么时候结束迭代过程？这是编写迭代程序必须考虑的问题。不能让迭代过程无休止地重复执行下去。迭代过程的控制通常可分为两种情况：一种是所需的迭代次数是个确定的值，可以计算出来；另一种是所需的迭代次数无法确定。对于前一种情况，可以构建一个固定次数的循环来实现对迭代过程的控制；对于后一种情况，需要进一步分析出用来结束迭代过程的条件。

编写计算斐波那契（Fibonacci）数列的第 n 项函数 fib（n）。

斐波那契数列为：0、1、1、2、3、……，即：

```
fib(0)=0;
fib(1)=1;
```

```
fib(n)=fib(n-1)+fib(n-2) （当 n>1 时）。
写成递归函数有：
int fib(int n)
{ if (n==0) return 0;
if (n==1) return 1;
if (n>1) return fib(n-1)+fib(n-2);
    }
```

一个饲养场引进一只刚出生的新品种兔子，这种兔子从出生的下一个月开始，每月新生一只兔子，新生的兔子也如此繁殖。如果所有的兔子都不死去，问到第 12 个月时，该饲养场共有兔子多少只？

分析：这是一个典型的递推问题。我们不妨假设第 1 个月时兔子的只数为 u1，第 2 个月时兔子的只数为 u2，第 3 个月时兔子的只数为 u3，……根据题意，“这种兔子从出生的下一个月开始，每月新生一只兔子”，则有

u1 ＝ 1， u2 ＝ u1 ＋ u1 × 1 ＝ 2， u3 ＝ u2 ＋ u2 × 1 ＝ 4，……

根据这个规律，可以归纳出下面的递推公式：

u n ＝ u n － 1 × 2 (n ≥ 2)

对应 u n 和 u n － 1，定义两个迭代变量 y 和 x，可将上面的递推公式转换成如下迭代关系：

```
y=x*2
x=y
```

让计算机对这个迭代关系重复执行 11 次，就可以算出第 12 个月时的兔子数。参考程序如下：

```
cls
x=1
for i=2 to 12
y=x*2
x=y
next i
print y
    end
```

分而治之法

1、分治法的基本思想

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 N 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。

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。

3、分治法的基本步骤

分治法在每一层递归上都有三个步骤：

（1）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；

（2）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；

（3）合并：将各个子问题的解合并为原问题的解。

快速排序

在这种方法中， n 个元素被分成三段（组）：左段 l e f t，右段 r i g h t 和中段 m i d d l e。中段仅包含一个元素。左段中各元素都小于等于中段元素，右段中各元素都大于等于中段元素。因此 l e f t 和 r i g h t 中的元素可以独立排序，并且不必对 l e f t 和 r i g h t 的排序结果进行合并。m i d d l e 中的元素被称为支点(p i v o t)。图 1 4 - 9 中给出了快速排序的伪代码。

/ /使用快速排序方法对 a[0 :n- 1]排序

从 a[0 :n- 1]中选择一个元素作为 m i d d l e，该元素为支点

把余下的元素分割为两段 left 和 r i g h t，使得 l e f t 中的元素都小于等于支点，而 right 中的元素都大于等于支点

递归地使用快速排序方法对 left 进行排序

递归地使用快速排序方法对 right 进行排序

所得结果为 l e f t + m i d d l e + r i g h t

考察元素序列[4 , 8 , 3 , 7 , 1 , 5 , 6 , 2]。假设选择元素 6 作为支点，则 6 位于 m i d d l e；4，3，1，5，2 位于 l e f t；8，7 位于 r i g h t。当 left 排好序后，所得结果为 1，2，3，4，5；当 r i g h t 排好序后，所得结果为 7，8。把 right 中的元素放在支点元素之后， l e f t 中的元素放在支点元素之前，即可得到最终的结果[1 , 2 , 3 , 4 , 5 , 6 , 7 , 8]。

把元素序列划分为 l e f t、m i d d l e 和 r i g h t 可以就地进行（见程序 1 4 - 6）。在程序 1 4 - 6 中，支点总是取位置 1 中的元素。也可以采用其他选择方式来提高排序性能，本章稍后部分将给出这样一种选择。

程序 14-6 快速排序

```
template<class T>
    void QuickSort(T*a, int n)
    {// 对 a[0:n-1] 进行快速排序
    {// 要求 a[n] 必需有最大关键值
    quickSort(a, 0, n-1);
    template<class T>
    void quickSort(T a[], int l, int r)
    {// 排序 a [ l : r ]， a[r+1] 有大值
    if (l >= r) return;
    int i = l, // 从左至右的游标
    j = r + 1; // 从右到左的游标
    T pivot = a[l];
     // 把左侧>= pivot 的元素与右侧<=
pivot 的元素进行交换
while (true) {
do {// 在左侧寻找>= pivot 的元素
i = i + 1;
} while (a < pivot);
do {// 在右侧寻找<= pivot 的元素
j = j - 1;
} while (a[j] > pivot);
if (i >= j) break; // 未发现交换对象
Swap(a, a[j]);
}
// 设置 p i v o t
a[l] = a[j];
a[j] = pivot;
quickSort(a, l, j-1); // 对左段排序
```

quickSort(a, j+1, r); // 对右段排序　　}

贪婪法

它采用逐步构造最优解的思想，在问题求解的每一个阶段，都作出一个在一 定标准下看上去最优的决策；决策一旦作出，就不可再更改。制定决策的依据称为贪婪准则。

贪婪法是一种不追求最优解，只希望得到较为满意解的方法。贪婪法一般可以快速得到满意的解，因为它省去了为找最优解要穷尽所有可能而必须耗费的大量时间。贪婪法常以当前情况为基础作最优选择，而不考虑各种可能的整体情况，所以贪婪法不要回溯。

【问题】　背包问题

问题描述：有不同价值、不同重量的物品 n 件，求从这 n 件物品中选取一部分物品的选择方案，使选中物品的总重量不超过指定的限制重量，但选中物品的价值之和最大。

```
 #include<stdio.h>
void main()
{
    int
m,n,i,j,w[50],p[50],pl[50],b[50],s=0,max;
    printf("输入背包容量 m,物品种类 n :");
    scanf("%d %d",&m,&n);
    for(i=1;i<=n;i=i+1)
    {
        printf("输入物品的重量 W 和价值
P :");
        scanf("%d %d",&w[i],&p[i]);
        pl[i]=p[i];
        s=s+w[i];
    }
    if(s<=m)
    {
        printf("whole choose\n");
        //return;
}
for(i=1;i<=n;i=i+1)
{
    max=1;
    for(j=2;j<=n;j=j+1)
    if(pl[j]/w[j]>pl[max]/w[max])
    max=j;
    pl[max]=0;
    b[i]=max;
}
for(i=1,s=0;s<m && i<=n;i=i+1)
s=s+w[b[i]];
if(s!=m)
w[b[i-1]]=m-w[b[i-1]];
for(j=1;j<=i-1;j=j+1)
printf("choose weight %d\n",w[b[j]]);
```

}动态规划的基本思想

前文主要介绍了动态规划的一些理论依据，我们将前文所说的具有明显的阶段划分和状态转移方程的动态规划称为**标准动态规划**，这种标准动态规划是在研究多阶段决策问题时推导出来的，具有严格的数学形式，适合用于理论上的分析。在实际应用中，许多问题的阶段划分并不明显，这时如果刻意地划分阶段法反而麻烦。一般来说，只要该问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优子化原理），则可以考虑用动态规划解决。

动态规划的实质是**分治思想**和**解决冗余**，因此，**动态规划**是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。由此可知，动态规划法与分治法和贪心法类似，它们都是将问题实例归纳为更小的、相似的

子问题，并通过求解子问题产生一个全局最优解。
贪心法的当前选择可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心法自顶向下，一步一步地作出贪心选择；
而分治法中的各个子问题是独立的（即不包含公共的子问题），因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。
不足之处：如果当前选择可能要依赖子问题的解时，则难以通过局部的贪心策略达到全局最优解；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。
解决上述问题的办法是利用动态规划。该方法主要应用于最优化问题，这类问题会有多种可能的解，每个解都有一个值，而动态规划找出其中最优（最大或最小）值的解。若存在若干个取最优值的解的话，它只取其中的一个。在求解过程中，该方法也是通过求解局部子问题的解达到全局最优解，但与分治法和贪心法不同的是，动态规划允许这些子问题不独立，（亦即各子问题可包含公共的子问题）也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存起来，避免每次碰到时都要重复计算。
因此，动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。
3、动态规划算法的基本步骤
设计一个标准的动态规划算法，通常可按以下几个步骤进行：
（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。注意这若干个阶段一定要是有序的或者是可排序的（即无后向性），否则问题就无法用动态规划求解。
（2）选择状态：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：之所以把这两步放在一起，是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，如果我们确定了决策，状态转移方程也就写出来了。但事实上，我们常常是反过来做，根据相邻两段的各状态之间的关系来确定决策。
（4）写出规划方程（包括边界条件）：动态规划的基本方程是规划方程的通用形式化表达式。一般说来，只要阶段、状态、决策和状态转移确定了，这一步还是比较简单的。动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。根据动态规划的基本方程可以直接递归计算最优值，但是一般将其改为递推计算，实现的大体上的框架如下：
标准动态规划的基本框架
1. 对 $f_{n+1}(x_{n+1})$初始化；　{边界条件}
for k:=n downto 1 do
for 每一个 $x_k \in X_k$ do
for 每一个 $u_k \in U_k(x_k)$ do
begin
$f_k(x_k)$:=一个极值;　　{∞或－∞}
$x_{k+1}:=T_k(x_k,u_k)$;　　{状态转移方程}
$t:=\varphi(f_{k+1}(x_{k+1}),v_k(x_k,u_k))$;　　{基本方程(9)式}
if　t 比 $f_k(x_k)$更优　then $f_k(x_k):=t$; {计算 $f_k(x_k)$的最优值}
end;
t:=一个极值;　　{∞或－∞}
for 每一个 $x_1 \in X_1$ do
if $f_1(x_1)$比 t 更优　then $t:=f_1(x_1)$;　　{按照 10 式求出最优指标}

输出 t;

但是，实际应用当中经常不显式地按照上面步骤设计动态规划，而是按以下几个步骤进行：

（1）分析最优解的性质，并刻划其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

步骤（1）～（3）是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤（4）可以省略，若需要求出问题的一个最优解，则必须执行步骤（4）。此时，在步骤（3）中计算最优值时，通常需记录更多的信息，以便在步骤（4）中，根据所记录的信息，快速地构造出一个最优解。

总结：动态规划实际上就是最优化的问题，是指将原问题的大实例等价于同一最优化问题的较小实例，自底向上的求解最小实例，并将所求解存放起来，存放的结果就是为了准备数据。与递归相比，递归是不断的调用子程序求解，是自顶向下的调用和求解。

回溯法

回溯法也称为试探法，该方法首先暂时放弃关于问题规模大小的限制，并将问题的候选解按某种顺序逐一枚举和检验。当发现当前候选解不可能是解时，就选择下一个候选解；倘若当前候选解除了还不满足问题规模要求外，满足所有其他要求时，继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。在回溯法中，放弃当前候选解，寻找下一个候选解的过程称为回溯。扩大当前候选解的规模，以继续试探的过程称为向前试探。

1、回溯法的一般描述

可用回溯法求解的问题 P，通常要能表达为：对于已知的由 n 元组（x_1，x_2，…，x_n）组成的一个状态空间 E={（x_1，x_2，…，x_n）｜$x_i \in S_i$，i=1，2，…，n}，给定关于 n 元组中的一个分量的一个约束集 D，要求 E 中满足 D 的全部约束条件的所有 n 元组。其中 S_i 是分量 x_i 的定义域，且 $|S_i|$ 有限，i=1，2，…，n。我们称 E 中满足 D 的全部约束条件的任一 n 元组为问题 P 的一个解。

解问题 P 的最朴素的方法就是枚举法，即对 E 中的所有 n 元组逐一地检测其是否满足 D 的全部约束，若满足，则为问题 P 的一个解。但显然，其计算量是相当大的。

我们发现，对于许多问题，所给定的约束集 D 具有完备性，即 i 元组（x_1，x_2，…，x_i）满足 D 中仅涉及到 x_1，x_2，…，x_i 的所有约束意味着 j（j<i）元组（x_1，x_2，…，x_j）一定也满足 D 中仅涉及到 x_1，x_2，…，x_j 的所有约束，i=1，2，…，n。换句话说，只要存在 $0 \leqslant j \leqslant n-1$，使得（$x_1$，$x_2$，…，$x_j$）违反 D 中仅涉及到 x_1，x_2，…，x_j 的约束之一，则以（x_1，x_2，…，x_j）为前缀的任何 n 元组（x_1，x_2，…，x_j，x_{j+1}，…，x_n）一定也违反 D 中仅涉及到 x_1，x_2，…，x_i 的一个约束，$n \geqslant i > j$。因此，对于约束集 D 具有完备性的问题 P，一旦检测断定某个 j 元组（x_1，x_2，…，x_j）违反 D 中仅涉及 x_1，x_2，…，x_j 的一个约束，就可以肯定，以（x_1，x_2，…，x_j）为前缀的任何 n 元组（x_1，x_2，…，x_j，x_{j+1}，…，x_n）都不会是问题 P 的解，因而就不必去搜索它们、检测它们。回溯法正是针对这类问题，利用这类问题的上述性质而提出来的比枚举法效率更高的算法。

回溯法首先将问题 P 的 n 元组的状态空间 E 表示成一棵高为 n 的带权有序树 T，把在 E 中求问题 P 的所有解转化为在 T 中搜索问题 P 的所有解。树 T 类似于检索树，它可以这样构造：

设 S_i 中的元素可排成 $x_i^{(1)}$，$x_i^{(2)}$，…，$x_i^{(mi-1)}$，$|S_i| = m_i$，i=1，2，…，n。从根开始，让 T 的第 I 层的每一个结点都有 m_i 个儿子。这 m_i 个儿子到它们的双亲的边，按从左到右的

次序，分别带权 $x_{i+1}^{(1)}$，$x_{i+1}^{(2)}$，…，$x_{i+1}^{(mi)}$，i=0，1，2，…，n-1。照这种构造方式，E 中的一个 n 元组（x_1，x_2，…，x_n）对应于 T 中的一个叶子结点，T 的根到这个叶子结点的路径上依次的 n 条边的权分别为 x_1，x_2，…，x_n，反之亦然。另外，对于任意的 0≤i≤n-1，E 中 n 元组（x_1，x_2，…，x_n）的一个前缀 I 元组（x_1，x_2，…，x_i）对应于 T 中的一个非叶子结点，T 的根到这个非叶子结点的路径上依次的 I 条边的权分别为 x_1，x_2，…，x_i，反之亦然。特别，E 中的任意一个 n 元组的空前缀（），对应于 T 的根。

因而，在 E 中寻找问题 P 的一个解等价于在 T 中搜索一个叶子结点，要求从 T 的根到该叶子结点的路径上依次的 n 条边相应带的 n 个权 x_1，x_2，…，x_n 满足约束集 D 的全部约束。在 T 中搜索所要求的叶子结点，很自然的一种方式是从根出发，按深度优先的策略逐步深入，即依次搜索满足约束条件的前缀 1 元组（x_{1i}）、前缀 2 元组（x_1，x_2）、…，前缀 I 元组（x_1，x_2，…，x_i），…，直到 i=n 为止。

在回溯法中，上述引入的树被称为问题 P 的状态空间树；树 T 上任意一个结点被称为问题 P 的状态结点；树 T 上的任意一个叶子结点被称为问题 P 的一个解状态结点；树 T 上满足约束集 D 的全部约束的任意一个叶子结点被称为问题 P 的一个回答状态结点，它对应于问题 P 的一个解。

【问题】　n 皇后问题

问题描述：求出在一个 n×n 的棋盘上，放置 n 个不能互相捕捉的国际象棋“皇后”的所有布局。

这是来源于国际象棋的一个问题。皇后可以沿着纵横和两条斜线 4 个方向相互捕捉。如图所示，一个皇后放在棋盘的第 4 行第 3 列位置上，则棋盘上凡打“×”的位置上的皇后就能与这个皇后相互捕捉。

```
1   2   3   4   5   6   7   8
        ×           ×
×         ×         ×
    ×     ×     ×
×   ×     Q   ×   ×   ×   ×   ×
    ×     ×     ×
×         ×         ×
          ×           ×
          ×               ×
```

从图中可以得到以下启示：一个合适的解应是在每列、每行上只有一个皇后，且一条斜线上也只有一个皇后。

求解过程从空配置开始。在第 1 列至第 m 列为合理配置的基础上，再配置第 m+1 列，直至第 n 列配置也是合理时，就找到了一个解。接着改变第 n 列配置，希望获得下一个解。另外，在任一列上，可能有 n 种配置。开始时配置在第 1 行，以后改变时，顺次选择第 2 行、第 3 行、…、直到第 n 行。当第 n 行配置也找不到一个合理的配置时，就要回溯，去改变前一列的配置。得到求解皇后问题的算法如下：

```
{    输入棋盘大小值 n；
     m=0;
```

```
    good=1;
    do {
        if (good)
            if (m==n)
            {    输出解；
                改变之，形成下一个候选解;
            }
            else    扩展当前候选接至下一列；
        else    改变之，形成下一个候选解；
        good=检查当前候选解的合理性；
    } while (m!=0);
}
```

在编写程序之前，先确定边式棋盘的数据结构。比较直观的方法是采用一个二维数组，但仔细观察就会发现，这种表示方法给调整候选解及检查其合理性带来困难。更好的方法乃是尽可能直接表示那些常用的信息。对于本题来说，“常用信息”并不是皇后的具体位置，而是“一个皇后是否已经在某行和某条斜线合理地安置好了”。因在某一列上恰好放一个皇后，引入一个一维数组（col[

]），值 col[i]表示在棋盘第 i 列、col[i]行有一个皇后。例如：col[3]=4，就表示在棋盘的第 3 列、第 4 行上有一个皇后。另外，为了使程序在找完了全部解后回溯到最初位置，设定 col[0]的初值为 0 当回溯到第 0 列时，说明程序已求得全部解，结束程序运行。

为使程序在检查皇后配置的合理性方面简易方便，引入以下三个工作数组：

（1）　数组 a[ ]，a[k]表示第 k 行上还没有皇后；

（2）　数组 b[ ]，b[k]表示第 k 列右高左低斜线上没有皇后；

（3）　数组 c[ ]，c[k]表示第 k 列左高右低斜线上没有皇后；

棋盘中同一右高左低斜线上的方格，他们的行号与列号之和相同；同一左高右低斜线上的方格，他们的行号与列号之差均相同。

初始时，所有行和斜线上均没有皇后，从第 1 列的第 1 行配置第一个皇后开始，在第 m 列 col[m]行放置了一个合理的皇后后，准备考察第 m+1 列时，在数组 a[

]、b[ ]和 c[ ]中为第 m 列，col[m]行的位置设定有皇后标志；当从第 m 列回溯到第 m-1 列，并准备调整第 m-1 列的皇后配置时，清除在数组 a[

]、b[ ]和 c[ ]中设置的关于第 m-1 列，col[m-1]行有皇后的标志。一个皇后在 m 列，col[m]行方格内配置是合理的，由数组 a[ ]、b[ ]和 c[ ]对应位置的值都为 1 来确定。细节见以下程序：

【程序】

```
# include
# include
# define    MAXN    20
int n,m,good;
int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];

void main()
```

```
{    int j;
    char awn;
    printf("Enter n:      ");     scanf("%d",&n);
    for (j=0;j<=n;j++)     a[j]=1;
    for (j=0;j<=2*n;j++)     cb[j]=c[j]=1;
    m=1;     col[1]=1;          good=1;     col[0]=0;
    do {
        if (good)
            if (m==n)
            {    printf("列\t 行");
                for (j=1;j<=n;j++)
                    printf("%3d\t%d\n",j,col[j]);
                printf("Enter a character (Q/q for exit)!\n");
                scanf("%c",&awn);
                if (awn=='Q'||awn=='q')     exit(0);
                while (col[m]==n)
                {    m--;
                    a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1;
                }
                col[m]++;
            }
            else
            {    a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=0;
                col[++m]=1;
            }
        else
        {    while (col[m]==n)
            {    m--;
                a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1;
            }
            col[m]++;
        }
        good=a[col[m]]&&b[m+col[m]]&&c[n+m-col[m]];
    } while (m!=0);
}
```

试探法找解算法也常常被编写成递归函数，下面两程序中的函数 queen_all()和函数 queen_one()能分别用来解皇后问题的全部解和一个解。

【程序】

```
# include
# include
# define    MAXN    20
int n;
int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];
```

```
void main()
{    int j;
    printf("Enter n:      ");    scanf("%d",&n);
    for (j=0;j<=n;j++)    a[j]=1;
    for (j=0;j<=2*n;j++)    cb[j]=c[j]=1;
    queen_all(1,n);
}

void queen_all(int k,int n)
{    int i,j;
    char awn;
    for (i=1;i<=n;i++)
        if (a[i]&&b[k+i]&&c[n+k-i])
        {    col[k]=i;
            a[i]=b[k+i]=c[n+k-i]=0;
            if (k==n)
            {    printf("列\t 行");
                for (j=1;j<=n;j++)
                    printf("%3d\t%d\n",j,col[j]);
                printf("Enter a character (Q/q for exit)!\n");
                scanf("%c",&awn);
                if (awn=='Q'||awn=='q')    exit(0);
            }
            queen_all(k+1,n);
            a[i]=b[k+i]=c[n+k-i];
        }
}
```

采用递归方法找一个解与找全部解稍有不同，在找一个解的算法中，递归算法要对当前候选解最终是否能成为解要有回答。当它成为最终解时，递归函数就不再递归试探，立即返回；若不能成为解，就得继续试探。设函数 queen_one()返回 1 表示找到解，返回 0 表示当前候选解不能成为解。细节见以下函数。

【程序】

```
# define    MAXN    20
    int n;
    int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];
    int queen_one(int k,int n)
    {    int i,found;
        i=found=0;
        While (!found&&i        {    i++;
            if (a[i]&&b[k+i]&&c[n+k-i])
            {    col[k]=i;
                a[i]=b[k+i]=c[n+k-i]=0;
```

```
                if (k==n)     return 1;
                else
                    found=queen_one(k+1,n);
                a[i]=b[k+i]=c[n+k-i]=1;
            }
        }
        return found;
    }
```

分支定界法：

分支限界法：

这是一种用于求解组合优化问题的排除非解的搜索算法。类似于回溯法，分枝定界法在搜索解空间时，也经常使用树形结构来组织解空间。然而与回溯法不同的是，回溯算法使用深度优先方法搜索树结构，而分枝定界一般用宽度优先或最小耗费方法来搜索这些树。因此，可以很容易比较回溯法与分枝定界法的异同。相对而言，分枝定界算法的解空间比回溯法大得多，因此当内存容量有限时，回溯法成功的可能性更大。

算法思想：分枝定界（branch and bound）是另一种系统地搜索解空间的方法，它与回溯法的主要区别在于对 E-节点的扩充方式。每个活节点有且仅有一次机会变成 E-节点。当一个节点变为 E-节点时，则生成从该节点移动一步即可到达的所有新节点。在生成的节点中，抛弃那些不可能导出（最优）可行解的节点，其余节点加入活节点表，然后从表中选择一个节点作为下一个 E-节点。从活节点表中取出所选择的节点并进行扩充，直到找到解或活动表为空，扩充过程才结束。

有两种常用的方法可用来选择下一个 E-节点（虽然也可能存在其他的方法）：

1) 先进先出（F I F O） 即从活节点表中取出节点的顺序与加入节点的顺序相同，因此活节点表的性质与队列相同。

2) 最小耗费或最大收益法在这种模式中，每个节点都有一个对应的耗费或收益。如果查找一个具有最小耗费的解，则活节点表可用最小堆来建立，下一个 E-节点就是具有最小耗费的活节点；如果希望搜索一个具有最大收益的解，则可用最大堆来构造活节点表，下一个 E-节点是具有最大收益的活节点

装载问题

用一个队列 Q 来存放活结点表，Q 中 weight 表示每个活结点所相应的当前载重量。当 weight＝－1 时，表示队列已达到解空间树同一层结点的尾部。

算法首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。2 个儿子结点都产生后，当前扩展结点被舍弃。

活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空。当取出的元素是-1 时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1 加入活结点队列，算法开始处理下一层的活结点。

```
/*该版本只算出最优解*/
#include<stdio.h>
#include<malloc.h>
struct Queue{
int weight ;
struct Queue* next ;
};
int bestw = 0 ; //  目前的最优值
Queue* Q; //  活结点队列
Queue* lq = NULL ;
Queue* fq = NULL ;
```

```
int Add(int w)
{
Queue* q ;
q = (Queue*)malloc(sizeof(Queue)) ;
if(q ==NULL)
{
   printf("没有足够的空间分配\n") ;
   return 1 ;
}
q->next = NULL ;
q->weight = w ;
if(Q->next == NULL)
{
   Q->next = q ;
   fq = lq = Q->next ; //一定要使元素放到链中
}
else
{
   lq->next = q ;
   lq = q ;
//     lq = q->next ;
}
return 0 ;
}
int IsEmpty()
{
if(Q->next==NULL)
   return 1 ;
return 0 ;
}
int Delete(int&w)
{
Queue* tmp = NULL ;
// fq = Q->next ;
tmp = fq ;
w = fq->weight ;
Q->next = fq->next ; /*一定不能丢了链表头*/
fq = fq->next ;
free(tmp) ;
return 0 ;
}
void EnQueue(int wt,
    int& bestw, int i, int n) //该函数负责加入活结点
{ // 如果不是叶结点，则将结点权值 wt 加入队列 Q
if (i == n)
    { // 叶子
   if (wt>bestw)
    bestw = wt;
}
else
   Add(wt); // 不是叶子
}
int MaxLoading(int w[], int c, int n)
{ // 返回最优装载值
// 为层次 1 初始化
int err ; //返回值
int i = 1; // 当前扩展结点的层
int Ew = 0; // 当前扩展结点的权值
bestw = 0; // 目前的最优值
Q = (Queue*)malloc(sizeof(Queue)) ;
Q->next = NULL ;
Q->weight = -1 ;
err = Add(-1) ; //标记本层的尾部
    if(err)
{
   return 0 ;
}

while (true)
{
   // 检查左孩子结点
   if (Ew + w[i] <= c) // x[i] = 1
    EnQueue(Ew + w[i], bestw , i, n);
   // 右孩子总是可行的
   EnQueue(Ew, bestw, i, n); // x[i] = 0
   Delete(Ew); // 取下一个扩展结点
   if (Ew == -1)
   { // 到达层的尾部
    if (IsEmpty())
     return bestw;
    if(i<n)
     Add(-1); // 同层结点的尾部
    Delete(Ew); // 取下一扩展结点
          i++; // 进入下一层
```

```
    }
}
}
int main()
{
int n =0 ;
int c = 0 ;
int i = 0 ;
int* w ;
FILE *in , *out ;
in = fopen("input.txt" , "r") ;
out = fopen("output.txt" , "w") ;
if(in==NULL||out==NULL){
    printf("没有输入输出文件\n") ;
    return 1 ;
}
fscanf(in , "%d" , &n) ;
fscanf(in , "%d" , &c) ;
w = (int*)malloc(sizeof(int)*(n+1)) ;
for(i =1 ; i<=n ; i++)
    fscanf(in , "%d" , &w[i]) ;
MaxLoading(w , c , n) ;
fprintf(out , "%d\n" , bestw) ;
return 0 ;
}
```