第五章 回 溯

一、 回溯法

- 有“通用的解题法”之称。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。

问题的解空间和状态空间树

问题的输入：n个输入；

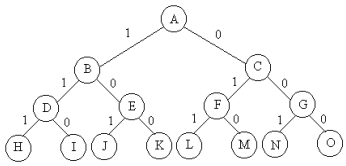
问题的解向量：n元组$X=(x_1,x_2,...x_n)$

问题的解空间：x_i的所有可能取值范围的组合

0/1背包问题中，x_i的取值为{0，1}。所以，当n=3时，问题的解空间（x_i的所有可能取值范围的组合）为：

(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)

有2^n种可能的解



n=3时，0/1背包问题的状态空间树

- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

回溯法指导思想——走不通，就掉头。

回溯法解题时，一般包含三个步骤：

1）对所给定的问题，定义问题的解空间；

2）确定状态空间树的结构；

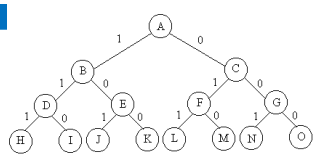
3）用深度优先搜索法搜索解空间，用约束方程和目标函数的界对状态空间树进行修剪，生成搜索树，取得问题的解。

- 状态空间树的动态搜索

问题的解只能是整个空间中的一个子集，子集中的解必须满足事先给定的某些约束条件（问题的可行解）。***可行解不止一个，因此，对于需要寻找最优解的问题，事先给出一个目标函数，使得目标函数取得极值的可行解，称为最优解。***

- 求问题所有解：要回溯到根，且根结点的所有子树都已被搜索遍才结束。
- 求任一解：只要搜索到问题的一个解就可结束。

7



n=3时，0/1背包问题的状态空间树

搜索方法：从根结点出发，沿着其儿子结点向下搜索，如果其儿子结点的边所标记的分量x_i满足约束条件和目标函数的界，就把分量x_i加入到它的部分解中，并继续向下搜索以儿子为结点作为根结点的子树；如果其儿子结点的边所标记的分量x_i不满足约束条件和目标函数的界，就结束对以儿子为结点作为根结点的子树的搜索，选择另一个儿子结点作为根结点的子树的搜索。

8

二、回溯法搜索方法

- *l*_结点（活结点）：不是叶结点，并且满足约束条件和目标函数的界，同时这个结点的所有儿子结点还没有全部搜索完毕。
- *e*_结点（扩展结点）：当前正在搜索其儿子结点的结点，*e*_结点必定是活结点。
- *d*_结点（死结点）：***不满足约束条件或目标函数***的结点，或儿子结点已经全部搜索完毕的结点，或者是叶结点。

9

- 当搜索到一个*l*_结点时，就把这个*l*_结点变为*e*_结点，继续向下搜索这个结点的儿子结点。当搜索到一个*d*_结点，而且还未得到问题的最终解时，就向上回溯到它的父亲结点。如果这个父结点当前还是*e*_结点，就继续搜索这个父亲结点的另外一个儿子结点。如果这个父亲结点随着所有的儿子结点都已搜索完毕而成为*d*_结点，就沿着这个父亲结点向上，回溯到它的祖父结点。这个过程继续进行，直到找到满足问题的最终解，或者状态空间树的根结点变为*d*_结点为止。

10

加约束的枚举算法

如果能够排除那些没有前途的状态，会节约时间；

- **如何提前发现？**
  - **在每一次扩展*e*结点后，都进行检查；**
- **对检查结果如何处理？**
  - **检查合格的才继续向下扩展；**
  - **遇到不合格的“掉头就走”。**

约束条件或目标函数

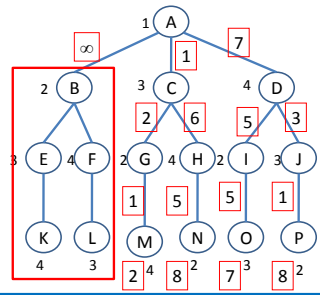
11

- 例题5.1 四个顶点的货郎担问题。求从顶点1出发，最后回到顶点1的最短路线。

	v1	v2	v3	v4
v1	∞	∞	1	7
v2	8	∞	5	1
v3	7	2	∞	6
v4	2	5	3	∞

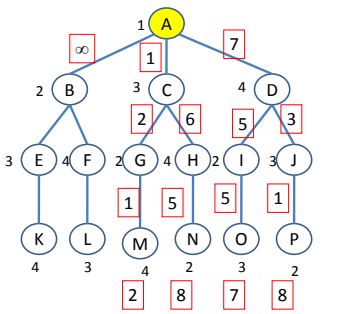
已经去掉不符合约束的可能解

$x_i \neq x_j$



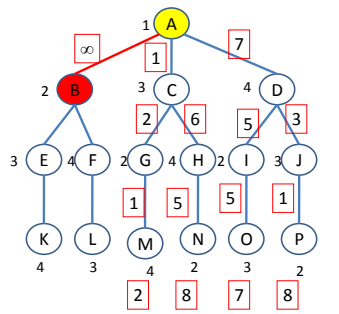
12

设目标函数$b=\infty$



13

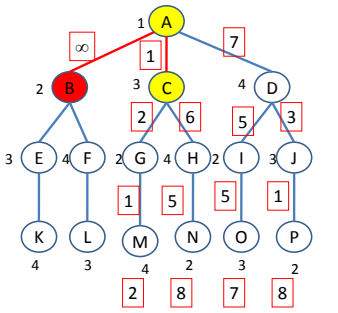
设目标函数$b=\infty$



14

设目标函数$b=\infty$

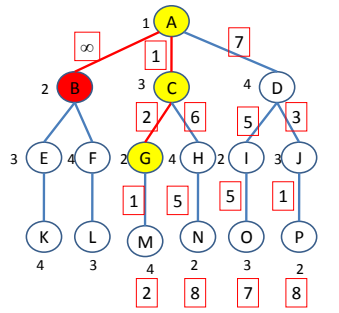
$L=1$



15

设目标函数$b=\infty$

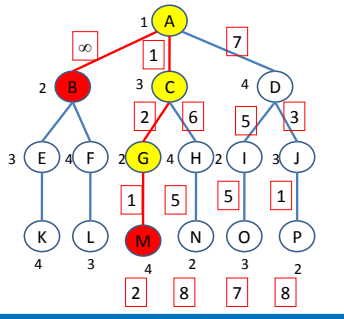
$L=3$



16

设目标函数$b=\infty$

$L=4$

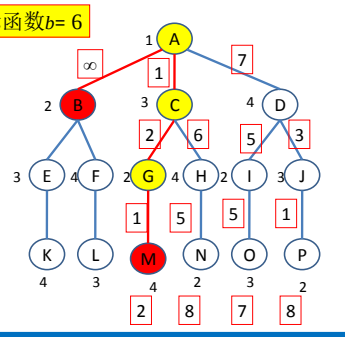


17

得到一个解（1,3,2,4,1）

设目标函数$b=6$

$L=6$

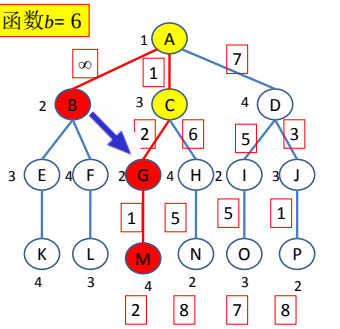


18

回溯到上一层，找是否存在其他的解

设目标函数b= 6

L=3

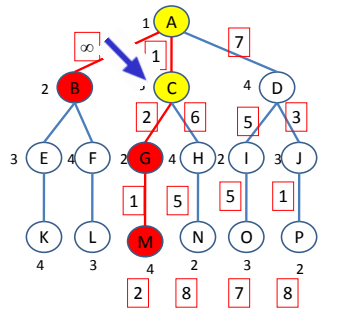


19

回溯到上一层，找是否存在其他的解

设目标函数b= 6

L=1

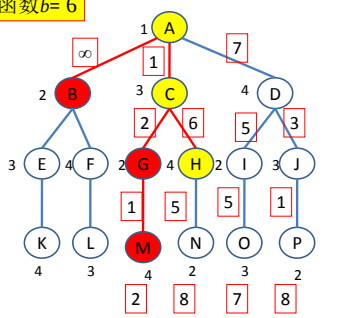


20

此时，得到的解已经不满足目标函数了，终止。

设目标函数b= 6

L=7

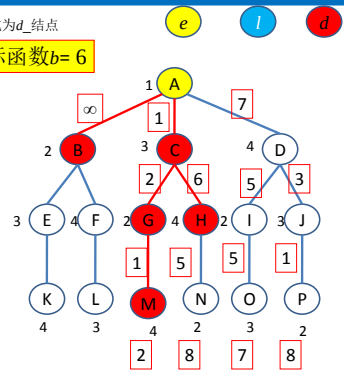


21

C 和H都成为d_结点

设目标函数b= 6

L=7

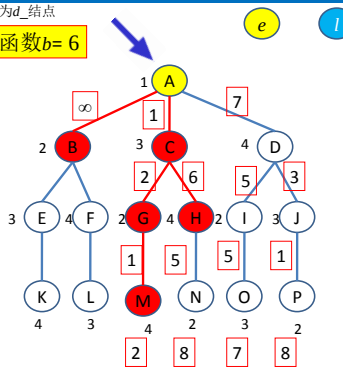


22

C 和H都成为d_结点

设目标函数b= 6

L=7



23

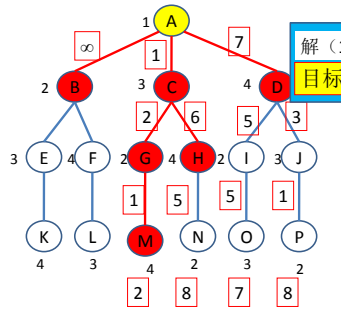
此时，得到的解已经不满足目标函数了，终止。

设目标函数b= 6

L=7

解（1,3,2,4,1）

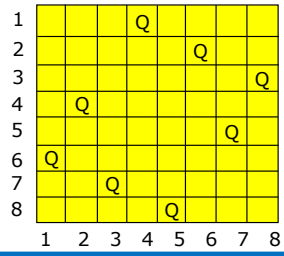
目标函数b= 6



24

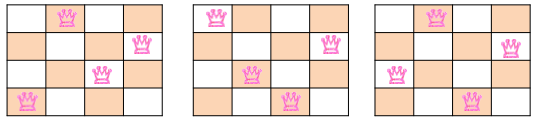
例题5.2 n 后问题

- 在$n\times n$格的棋盘上放置n个皇后。皇后不可以处在同一行或同一列或同一斜线上的位子。

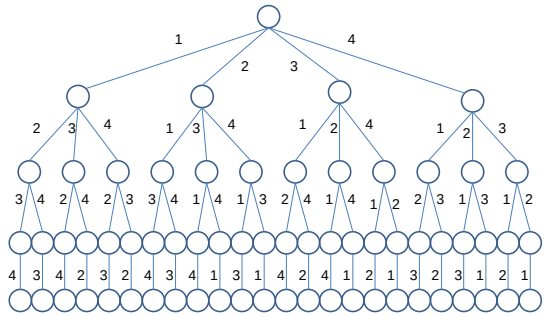


四后问题的求解过程

- 在四后问题中，每一行只能放一个皇后，每一个皇后在每一行上有四个位置可以选择。



- 在4*4格的棋盘上放置四个皇后，有4^4种可能的布局。令向量$x=(x_1,x_2,x_3,x_4)$表示皇后的布局。其中分量x_i表示第i行皇后的位置（列的位置）。
- **四皇后问题的解空间可以用一棵完全四叉树表示，每个结点有四个可能的分支。因为每一个皇后不能在同一列，因此，可以把4^4种可能压缩，成为4！种可能的解。（解空间）**



四皇后问题的状态空间树及搜索树

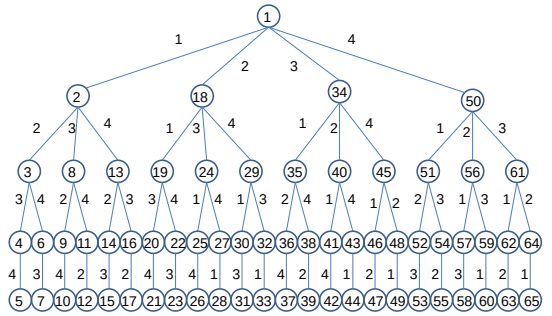
- 按照问题的需求，可以列出***约束方程：***

$$x_i \neq x_j \qquad 1\le i\le 4,\ 1\le j\le 4, i\neq j \qquad (5.2.1)$$

$$|x_i - x_j| \neq |i-j| \qquad 1\le i\le 4,\ 1\le j\le 4, i\neq j \qquad (5.2.2)$$

（5.2.1）保证第i行的皇后和第j行的皇后不会在同一列上。
（5.2.2）保证两个皇后的行号之差的绝对值不会等于列号之差的绝对值，因此它们的不会在斜率为±1的同一斜线上。

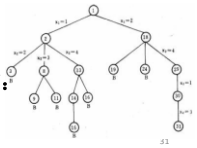
四皇后问题的状态空间树及搜索树



X=(0,0,0,0)

搜 索 代 价

- 对于4 Queen，若每个分支都是“一分为四”，则：
  - **0个皇后的状态：1个，即4^0；**
  - **1个皇后的状态：4个，即4^1；**
  - **2个皇后的状态：16个，即4^2；**
  - **3个皇后的状态：64个，即4^3；**
  - **4个皇后的状态：256个，即4^4；**
  - **共341个状态。**



- **使用回溯策略，实际扫描过的状态如图：**
  - **共16个，仅占5%。**

算法的实现

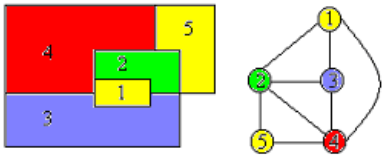
```
Bool place(int x[ ],int k)
{
   int I;
   for (i=1;i<k;i++)
    if ((x[i]==x[k])||(abs(x[i]-x[k])==abs(i-k))
      return false;
    return true;
}
```



```
Void n_queens(int n,int x[ ])
{
  int k=1;
  x[1]=0;
  while (k>0) {
    x[k]=x[k]+1;                    //在当前列加1的位置开始搜索
   while ((x[k]<=n)&&(!place(x,k))) // 当前列位置是否满足条件
    x[k]=x[k]+1;                    //不满足条件，搜索下一列位置
   if (x[k]<=n) {                   // 存在满足条件的列？
    if (k==n) break;                // 若是最后一个皇后，完成搜索
     else {
       k=k+1;      x[k]=0;          //否则，处理下一个皇后
     }
    }
   else {                           //已判断完n列，均不满足条件
     x[k]=0; k=k-1;                 //第k行复位为0，回溯到前一行
   }
  }
}
```


5.3 图的着色问题

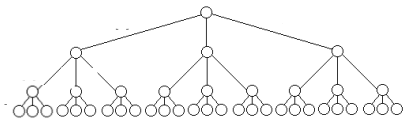
- 给定无向图$G(V,E)$，用m种颜色为图中每个顶点着色，要求每个顶点着一种颜色，并且使得相邻两个顶点之间的颜色不同。



- 用一个n元组来描述图的一种着色：

$$(x_1, x_2, \ldots, x_n), x_i \in [1,2,3,\ldots,m], 1 \le i \le n$$

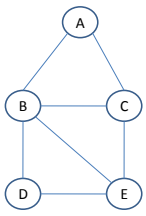
为了用m种颜色对n个顶点着色，就有m_n种可能的着色组合。其中有些是有效的着色，有些是无效的着色。因此它的状态空间树是高度为n的完全m叉树。

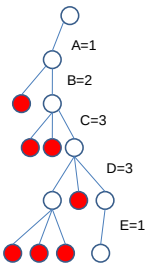


具有三个顶点的三着色问题的状态空间树

- 根据题意，可以写出约束方程如下：

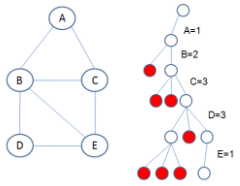
$$x[i] \neq x[j], \quad \text{若顶点}i\text{与顶点}j\text{相邻接}$$





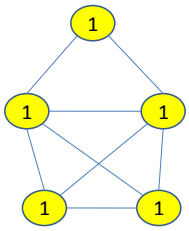
算法的实现：

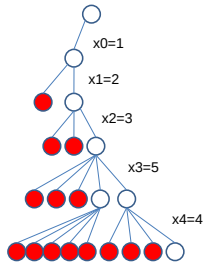
```
Bool ok(int x[ ],int k,bool c[ ][ ],int n)
{
  int I;
   for (i=0;i<k;i++)
     if (c[k][i]&&x[k]==x[i]) return false;
   return true;
}
```




```
Void m-coloring(int n,int m,int x[ ],bool c[ ][ ])
{
  int I,k;
  for (i=0;i<n;i++)  x[k]=0;
  k=0;
  while (k>=0) {
    x[k]=x[k]+1;
    while ((x[k]<=m)&&(!ok(x,k,c,n))
      x[k]=x[k]+1;
    if (x[k]<=m) {
      if (k==n-1) break;
      else k=k+1;
    }
    Else {
    x[k]=0;
    k=k-1;
   }
}
}
```


哈密尔顿回路





回溯法的效率分析

- 影响回溯法效率的主要因素：
- 生成结点所花费的时间
- 计算约束方程所花费的时间
- 计算目标函数所花费的时间
- 所生成的结点个数

例5.4 素数环问题

- 素数环问题： 把从1到20这20个数摆成一个环，要求相邻两个数的和是一个素数。

问题分析：

✓ **搜索从1开始，每个空位有2~20共19种可能；**

✓ **填进去的数合法：与前面的数不相同；与左边相邻的数的和是一个素数；**

✓ **第20个数还要判断和第1个数的和是否素数。**

20以内的素数环：

1 2 3 4

1 4 3 2 5 6

1 2 3 8 5 6 7 4

1 2 3 4 7 6 5 8 9 10

1 2 3 4 7 6 5 12 11 8 9 10

1 2 3 4 7 6 13 10 9 14 5 8 11 12

1 2 3 4 7 6 5 12 11 8 9 14 15 16 13 10

1 2 3 4 7 6 5 8 9 10 13 16 15 14 17 12 11 18

1 2 3 4 7 6 5 8 9 10 13 16 15 14 17 20 11 12 19 18

算法流程：

1、数据初始化；

2、递归地填数：

判断第i种可能是否合法？

A、如果合法：填数；判断是否到达目标（20个已填完）：是，打印结果；不是，递归填下一个；

B、如果不合法：选择下一种可能；

```
main( )
{ int a[20],k;
  for
(k=1;k<=20;k++)
      a[k]=0;
 a[1]=1;
 try(2);
}
```

```
try(int i)
{ int k;
  for (k=2;k<=20;k++)
    if (check1(k,i)=1 and
          check3(k,i)=1)
      { a[i]=k;
        if (i=20) output( );
        else
          {try(i+1);
           a[i]=0;}
      }
}
```


素数环问题-算法

```
check1(int j,int i)
{ int k;
  for (k=1;k<=i-1;k++)
     if (a[k]=j ) return(0);
  return(1);
}
check2(int x)
{ int k,n;
  n= sqrt(x);
  for (k=2;k<=n;k++)
     if (x mod k=0) return(0);
 return(1);
}
```

```
check3(int j,int i)
{ if (i<20)  return(check2(j+a[i-1]));
  else  return(check2(j+a[i-1]) and
check2(j+a[1]));
}

output( )
{ int k;
  for (k=1;k<=20;k++)
      print(a[k]);
  print("换行符" );
}
```


例5.5 马的遍历问题

在$n*m$的棋盘中，马只能走“日” 字。马从位置(x,y)处出发，把棋盘的每一格都走一次，且只走一次。找出所有路径。

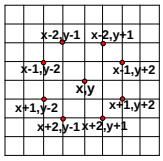
马的遍历问题-问题分析

- **问题1：问题解的搜索空间？**
  - **棋盘的规模是$n*m$，是指行有n条边，列有m条边。**
  - **马在棋盘的点上走，所以搜索空间是整个棋盘上的$n*m$个点。**
  - **用$n*m$的二维数组记录马行走的过程，初值为0表示未经过。**

马的遍历问题-问题分析

- **问题2：在寻找路径过程中，活结点的扩展规则？**
  - **对于棋盘上任意一点A(x,y)，有八个扩展方向：**

    A(x+1,y+2),A(x+2,y+1)
    A(x+2,y-1),A(x+1,y-2)
    A(x-1,y-2),A(x-2,y-1)
    A(x-2,y+1),A(x-1,y+2)

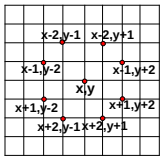


  - **为构造循环体，用数组fx[8]={1,2,2,1,-1,-2,-2,-1}, fy[8]= {2,1,-1,-2,-2,-1,1,2}来模拟马走“日”时下标的变化过程。**

马的遍历问题-问题分析

- **问题3：扩展的约束条件？**
  - **不出边界；**
  - **每个点只经过一次。**

    棋盘点对应的数组元素初值为0，对走过的棋盘点的值置为所走步数，起点存储“1”，终点存储“n*m”。



  - **函数check，检查当前状态是否合理。**

马的遍历问题-问题分析

- **问题4：搜索解空间？**

搜索过程是从任一点（*x*,*y*）出发，按深度优先的原则，从8个方向中尝试一个可以走的棋盘点，直到走过棋盘上所有$n*m$个点。用递归算法易实现此过程。

注意问题要求找出全部可能的解，就要注意回溯过程的清理现场工作，也就是置当前位置为未经过。

马的遍历问题-数据结构设计

1) 用一个变量*dep*记录递归深度，也就是走过的点数，当$dep=n*m$时，找到一组解。

2) 用$n*m$的二维数组记录马行走的过程，初始值为0表示未经过。搜索完毕后，起点存储的是“1”，终点存储的是“n*m”。

马的遍历问题-算法

```
int n=5 , m=4, dep , i, x , y , count;
int fx[8]={1,2,2,1,-1,-2,-2,-1},fy[8]={2,1,-1,-2,-2,-1,1,2},a[n][m];
main( )
{ count=0; dep=1;
  print('input x,y');     input(x,y);
  if (y>n or x>m or x<1 or y<1)
       { print('x,y error!');   return;}
  for(i=1;i<=n;i++)
     for(j=1;j<=m;j++)
          a[i][j]=0;
  a[x][y]=1;
  find(x,y,2);
  if (count=0 )   print("No answer!");
  else  print("count=",count);
}
```



```
find(int x,int y,int dep)
{ int  i,xx,yy;
 for (i=1;i<=8;i++)     //加上方向增量,形成新的坐标
 { xx=x+fx[i];  yy=y+fy[i];
  if (check(xx,yy)=1) //判断新坐标是否出界,是否已走过
   { a[xx,yy]=dep;     //走向新的坐标
    if (dep=n*m)  output( );
    else
       find(xx,yy,dep+1);   //从新坐标出发,递归下一层
   }
 }
 a[xx,yy]=0;          //回溯,恢复未走标志
}
```



```
output( )
{    count=count+1;
    print("换行符" );
     print("count=",count);
     for (x=1;x<=n;i++)
      { print("换行符" );
        for (y=1;y<=m;y++)
            print(a[x,y]);
      }
}
```


例5.6 找*n*个数中*r*个数的组合。

1、问题分析：

例如：当$n=5, r=3$时，所有组合为：

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5　　**total=10 {组合的总数}**

问题分析1（枚举）

当$n=5,r=3$时，所有组合为：

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

分析：每组3个数特点？

1)互不相同；
2)前面的数小于后面的数；
将上述两条作为约束条件。
3) 当r=3时，可用三重循环对每组中的3个数进行*枚举*。

算法1

```
main1( )
{   int n=5,i,j,k,t;
    t=0;
    for(i=1;i>=n;i++)
      for(j=1;j>=n;j++)
        for(k=1;k>=n;k++)
          if(i<>j)and(i<>k)and(i<j)and(j<k)
           {t=t+1;
            print(i,j,k);}
    print('total=',t);
}
```

算法复杂性： $O(n^3)$

问题分析2

用递归法设计此题：

在循环算法设计中，对n=5的实例，每个组合中的数据从小到大排列或从大到小排列同样可以设计出相应算法。但用递归思想设计时，每个组合中的数据必须从大到小排列；因为递归算法设计是要找出大规模问题与小规模问题之间的关系。

问题分析2

当$n=5,r=3$时，组合为：

5 4 3
5 4 2
5 4 1
5 3 2
5 3 1
5 2 1
4 3 2
4 3 1
4 2 1
3 2 1

total=10

分析$n=5,r=3$时10组组合数：

1)首先固定第一个数5，其后就是$n=4,r=2$的组合数，共6个组合。

2)其次固定第一个数4，其后就是$n=3,r=2$的组合数，共3个组合。

3)最后固定第一个数3，其后就是$n=2,r=2$的组合数，共1个组合。

至此，找到了“5个数中3个数的组合”与“4个数中2个数的组合、3个数中2个数的组合、2个数中2个数的组合”的递归关系。

递归算法的三个步骤为：

1）一般情况下：n个数中r个数组合递推到“n-1个数中r-1个数有组合，n-2个数中r-1个数有组合，……,r-1个数中r-1个数有组合”，共n-r+1 次递归。

2）递归的边界条件是r=1。

3）函数的主要操作是输出，每当递归到r=1时就有一组新组合产生，输出它们和一个换行符。

注意：n=5,r=3例子的递归规律，先固定5，然后要进行多次递归。即数字5要多次输出，所以要用数组存储以备每次递归到r=1时输出。同样每次向下递归都要用到数组，所以将数组设置为全局变量 。


```
int a[100];
main3( )
{ int n,r;
  print("n,r=");
  input(n,r);
  if （r>n）
    print("Input n,r error!");
  else
    { a[0]=r;
      comb(n,r); //调用递归过程
    }
}
```

```
comb(int m,int k)
{ int i,j;
  for (i=m;i>=k;i--)
  { a[k]=i;
    if (k>1)
        comb(i-1,k-1);
    else
      { for (j=a[0];j>0;j--)
            print(a[j]);
        print("换行符"); }
  }
}
```

分析：算法2递归的深度是r，每个算法要递归m-k+1次，所以时间复杂性是$O(r*n)$。

问题分析3(回溯法构造解)

1、问题分析：确定问题的解空间？

r元一维向量$(a_1,a_2,a_3,\ldots,a_r)$,$1\le a_i\le n$,$1\le i\le r$

可用一维数组a[ ]存储正在搜索的向量

。

2.如何进行搜索？如何表示约束条件？

通过实例来归纳，以n=5，r=3为例

在数组a中： a[1] a[2] a[3]

5	4	3
5	4	2
5	4	1
5	3	2
5	3	1
5	2	1
4	3	2
4	3	1
4	2	1
3	2	1

分析特点，搜索时依次对数组元素a[1],a[2],a[3]进行尝试：

a[1]尝试范围5-3

a[2]尝试范围4-2

a[3]尝试范围3-1

$a[r_i]$ i_1-i_2

规律：“后一个元素至少比前一个小 1 ”,r_1+i_2均为4=r+1,

r_1+i_1均为6=n+1。

归纳为一般情况：

a[1]尝试范围n-r,a[2]:n-1-r-1,……,a[r]:n-r+1-1;

结论：搜索过程中约束条件为r_1+a[r_1]>=r+1，若r_1+a[r_1]<r就要回溯到元素a[r_1-1]搜索，特别地a[r]=1时，回溯到元素a[r-1]搜索。


```
main( );
{ int n,r,a[20] ;
  print("n,r=");
  input(n,r);
  if （r>n）
     print("Input n,r error!");
  else
    { a[0]=r;
      comb(n,r);}     //调用递归过程
}
```



```
comb(int n,int r,int a[])
{ int i,ri;
  ri=1;  a[1]=n;
  while(a[1]>r-1)
    if (ri<r)            //没有搜索到底
      if (ri+a[ri]>=r+1) {a[ri+1]=a[ri]-1;   ri=ri+1;}
      else   {ri=ri-1; a[ri]=a[ri]-1;}    //回溯
    else
      { for (j=1;j<=r;j++)    print(a[j]);
        print("换行符" ); //输出组合数
        if (a[r]=1) {ri=ri-1;a[ri]=a[ri]-1;}    //回溯
        else  a[ri]=a[ri]-1;      //搜索到下一个数
      }
}
```


例5.7 排列树的回溯搜索

输出自然数1到n所有不重复的排列，即n 的全排列。

1、问题分析：确定问题的解空间？

一组n元一维向量$(x_1,x_2,x_3,\ldots,x_n)$,$1\le x_i\le n$,$1\le i\le n$

可用一维数组a[ ]存储正在搜索的向量。

2、约束条件？

x_i互不相同。

设置n个元素的数组d[ ]，数组元素用来记录数据1-n的使用情况，已使用置1，未使用置0。

```
int
p=0,n,a[100],d[100];
main( )
  {  int  j,n,
     print("Input n=");
     input(n);
     for(j=1;j<=r;j++)
          d[j]=0;
     try(1);
  }
```

```
try(int k)
{ int  j;
  for(j=1;j<=n;j++)
    { if (d[j]=0)  {a[k]=j;d[j]=1;}
      else   continue;
      if (k<n)  try(k+1);
      else
        {p=p+1; output( );}
    }
  d[a[k]]=0;
}
```



```
output( )
   {   int j;
       print(p,":")
       for(j=1;j<=n;j++)
           print(a[j]);
       print("换行符" );
   }
```

算法分析：全排列问题的复杂度为$O(n^n)$，不是一个好的算法。

例5.7 全排列算法另一解法——搜索排列树的算法框架

1、算法思路：

根据全排列的概念，定义数组初始值为(1,2,3,4,……,*n*)，这是全排列的一种，然后通过数据间的交换，则可产生所有的不同排列。

回溯法应用-例5.7-算法2

```
int a[100],n,s=0;
main( )
{  int i,;
   input(n);
   for(i=1;i<=n;i++)
        a[i]=i;
   try(1);
   print("换行符" ,"s=",s);
}
```

```
try(int t)
{  int j;
   if (t>n) {output( );}
   else
     for(j=t;j<=n;j++)
       { swap(t,j);
         try(t+1);
         swap(t,j); }
}
```



```
output( )
{ int j;
  printf("\n");
  for( j=1;j<=n;j++)  print(a[j]);
  s=s+1;
}
swap(int t1,int t2)
{  int t;
   t=a[t1];
   a[t1]= a[t2];
   a[t2]=t;
}
```

算法分析：全排列算法的复杂度为$O(n!)$。

例5.8 构造高精度数据

一个有趣的高精度数据

构造一个尽可能大的数，使其从高到低前一位能被1整除，前2位能被2整除，……，前*n*位能被*n*整除。

例5.8-问题分析

问题1：问题的解空间？

数学模型:

记高精度数据为$a_1\ a_2……a_n$，题目明确有两个要求：

1) a_1整除1且

(a_1*10+a_2)整除2且……

(a_1*10^{n-1}+a_2*10^{n-2}+…+a_n)整除n；

2) 求这样的最大的数。

问题2：如何搜索？

用从高位到低位逐位尝试，失败回溯的算法策略求解。算法的首位从1开始枚举，以后各位从0开始枚举。生成的高精度数据用数组的从高位到低位存储，1号元素开始存储最高位。

此数大小无法估计不妨为数组开辟100个空间。

问题3：如何确定和保留最优解？

算法中数组*A*为当前求解的高精度数据的暂存处，数组*B*为当前最大的满足条件的数。求解出的满足条件的数据之间只需要比较位数就能确定大小。*n* 为当前满足条件的最大数据的位数，当*i*>=*n*就认为找到了更大的解，*i*>*n*时，位数多数据一定大；*i*=*n*时，由于搜索是由小到大进行的，位数相等时后来满足条件的数据一定比前面的大。


```
main( )
{ int  A[101],B[101];  int  i,j,k,n,r;
  A[1]=1;
  for(i=2;i<=100;i++) A[i]=0;  //置初值:首位为1,其余为0
  n=1;   i=1;
  while(A[1]<=9)
  { if (i>=n)    //发现有更大的满足条件的高精度数据
       { n=i;              //转存到数组B中
         for (k=1;k<=n;k++)   B[k]=A[k]; }
    i=i+1;r=0;
    for(j=1;j<=i;j++)     //检查第i位是否满足条件
       r=r*10+A[j];
    r=r mod i;
    if(r<>0)              //若不满足条件
     {A[i]=A[i]+i-r ;     //第i位可能的解
       while (A[i]>9 and i>1)  //搜索完第i位解,回溯到前一位
         {A[i]=0;  i=i-1;  A[i]=A[i]+i;}
     }
  }
}
```


算法说明

1）从A[1]=1开始，每增加一位A[i](初值为0)先计算*r*=(A[1]*10^{i-1}+A[2]*10^{i-2}+……+A[i])，再测试*r*=r mod i是否为0。

2）*r*=0表示增加第i位后满足条件，与原有满足条件的数（存在数组*B*中）比较，若前者大，则更新数组*B*，继续增加下一位。

3）*r*≠0表示增加i位后不满足条件，接下来算法中并不是继续尝试A[i]=A[i]+1，而是继续尝试A[i]=A[i]+i-r，因为若A[i]=A[i]+i-r<=9时,(A[1]*10^{i-1}+A[2]*10^{i-2}+……+A[i]-r+i)mod *i*必为0，这样可减少尝试次数。如：17除5余2，15-2+5肯定能被5整除。

4）当A[i]-r+i>9时,要进位也不能算满足条件。这时，只能将此位恢复初值0且回退到前一位(*i*=*i*-1)尝试A[i]=A[i] +i……。这正是最后一个while循环所做工作。

5）当回溯到*i*=1时，A[1]加1开始尝试首位为2的情况，最后直到将A[1]=9的情况尝试完毕，算法结束。

例5.9 流水作业车间调度

*n*个作业要在由2台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业*i*所需的时间分别为ai和bi。流水作业调度问题要求确定这*n*个作业的最优加工顺序，使得从第一个作业在机器M1上开始加工，到最后一个作业在机器M2上加工完成所需的时间最少。作业在机器M1、M2的加工顺序相同。

问题分析

实例：考虑如下*n*=3的实例

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

3个作业有6种可能的调度方案是1,2,3;1,3,2;2,1,3;2,3,1;3,1,2; 3,2,1;所相应的完成时间和分别是19,18,20,21,19,19。

结论：最佳调度方案是1,3,2，其完成时间和为18。

问题分析

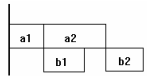
问题1：问题的解空间？

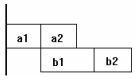
解空间是一棵排列树，用数组x[](初值为1,2,3,…,*n*)模拟不同排列，在不同排列下计算各种方案的加工耗时情况。

问题分析

问题2：搜索时如何计算作业完成时间？

机器M1进行顺序加工，其加工时间f1是固定的，f1[i]= f1[i-1]+M1[x[i]]。机器M2则有空闲（左下图）或积压（右下图）的情况，总加工时间f2。





当机器M2空闲时，f2[i]=f1[i]+ M2[x[i]]；
当机器M2有积压时，f2[i]= f2[i-1]+ M2[x[i]]；
总加工时间就是f2[n]。

问题分析

问题3：如何获得最优解？

一个最优调度应使机器M1没有空闲时间，且机器M2的空闲时间最少。简单的解决方法就是在搜索排列树的同时，不断更新最优解，最后找到问题的最优解。

在搜索过程中，当某一排列的前几步的加工时间已经大于当前的最小值，就无需进一步地搜索计算，从而提高算法效率。

数据结构设计

1)二维数组job[100][2]存储作业在M1,M2的加工时间。

2)由于f1在计算中，只需当前值，所以用变量存储即可；而f2在计算中，还依赖前一个作业的数据，所以有必要用数组存储。

3)考虑到回溯过程的需要，用变量f存储当前加工所需要的全部时间。

```
int
job[100][2],x[100],n,bestx[100],f1=0,f=0,bestf, f2[100]=0;
main( )
{  int j;
   input(n);
   for(i=1;i<=2;i++)
      for(j=1;j<=n;j++)
         input(job[j][i]);
   try(1);
}
```



```
try(int i)
{  int j;
   if (i=n+1)   //已搜索到一个叶结点，得到一新调度方案
     { for(j=1;j<=n;j++)  bestx[j]=x[j];
       bestf=f;}
   else
     for(j=i;j<=n;j++)
       { f1= f1+ job[x[j]][1];
         if (f2[i-1]>f1)  f2[i]= f2[i-1]+job[x[j]][2];
         else   f2[i]= f1+job[x[j]][2];
         f=f+f2[i];
         if (f<bestf)
            { swap(x[i],x[j]);  try(i+1);  swap(x[i],x[j]);}
         f1= f1-job[x[j]][1];
         f=f-f2[i];
       }
}
```


例5.10 0-1背包问题

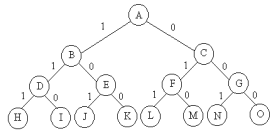
- 给定n种物品和一背包。物品i的重量是w_i，其价值为v_i，背包的容量为C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?
- 0-1背包问题也是一个子集选取问题。

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$$

问题分析

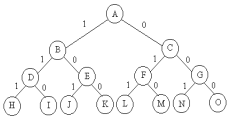
问题1：问题的解空间？



·**解空间：子集树**

问题分析

问题2：约束条件？



可行性约束函数： $\sum_{i=1}^{n} w_i x_i \le c_1$

问题3：如何扩展？

示例：n=3, C=30, w={16, 15, 15}, v={45, 25, 25}

- **开始，C_w(当前重量)=C=30, C_v=0,A唯一活结点，当前扩展结点**
- **扩展A，先到达B结点**
  - **C_w=C_w-w_1=14，C_v=C_v+v_1=45**
  - **此时A、B为活结点，B成为当前扩展结点**
  - **扩展B，先到达D**
    - **C_w<w_2，D导致一个不可行解，回溯到B**
  - **再扩展B到达E**
    - **E可行，此时A、B、E是活结点，E成为新的扩展结点**
    - **扩展E，先到达J**
      - **C_w<w_3，J导致一个不可行解，回溯到E**
    - **再次扩展E到达K**
      - **由于K是叶结点，即得到一个可行解x=(1,0,0), Cv=45**
      - **K不可扩展，成为死结点，返回到E**
    - **E没有可扩展结点，成为死结点，返回到B**
  - **B没有可扩展结点，成为死结点，返回到A**

回溯搜索过程

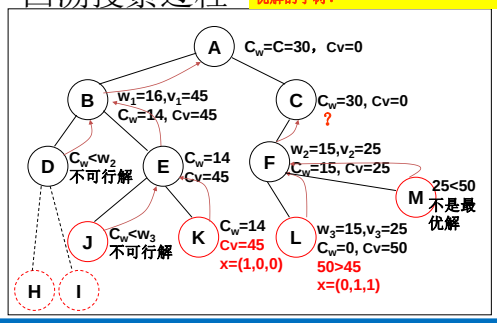
- A再次成为扩展结点，扩展A到达C
  - C_w=30，C_v=0，活结点为A、C，C为当前扩展结点
  - 扩展C，先到达F
    - C_w=C_w+w_2=15，C_v=C_v+v_2=25，此时活结点为A、C、F，F成为当前扩展结点
    - 扩展F，先到达L
      - C_w=C_w+w_3=0，C_v=C_v+v_3=50
      - L是叶结点，且50>45，得到一个可行解x=(0,1,1), C_v=50
      - L不可扩展，成为死结点，返回到F
    - 再扩展F到达M
      - M是叶结点，且25<50，不是最优解
      - M不可扩展，成为死结点，返回到F
    - F没有可扩展结点，成为死结点，返回到C

回溯搜索过程

  - 再扩展C到达G
    - C_w=30，C_v=0，活结点为A、C、G，G为当前扩展结点
    - 扩展G，先到达N，N是叶结点，且25<50，不是最优解
    - N不可扩展，返回到G
    - 再扩展G到达O，O是叶结点，且0<50，不是最优解
    - O不可扩展，返回到G
    - G没有可扩展结点，成为死结点，返回到C
  - C没有可扩展结点，成为死结点，返回到A
- A没有可扩展的结点，成为死结点，算法结束，最优解为bestX=(0,1,1),最优值bestV=50

回溯搜索过程

仅当Cw+w[i]≤C时进入左子树，递归搜索左子树；可行结点的右子树总是可行的。是否可以剪枝？剪去不含最优解的子树？



问题分析

问题4：如何剪枝？右子树的上界函数？

设r是当前尚未考虑的剩余物品价值总和；C_v是当前价值；bestv是当前最优价值。

结论：

- **当r+C_v≤bestv时，可剪去右子树。**
- **剩余物品是否都能装入？右子树中解的上界如何计算？**
- **方法：将剩余物品依其单位重量价值递减排序，然后依次装入，直至装不下时，再装入该物品的一部分，从而装满背包，由此得到的价值是右子树中解的一个上界。**

上界函数：

举例：n=4,c=7,v=(9,10,7,4),w=(3,5,2,1)。4个物品的单位重量价值分别是(3,2,3.5,4),以物品单位重量价值的递减序装入物品。装入物品4,3,1，剩余背包容量为1，只能装入0.2的物品2。

最终解为x=[1,0.2,1,1]，价值为22。

例5.11-连续邮资问题

假设国家发行了*n*种不同面值的邮票，并且规定每张信封上最多只允许贴*m*张邮票。连续邮资问题要求对于给定的*n*和*m*的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

连续邮资问题举例

例如，当n=2、m=3时，如果面值分别为1、4，则在1-6之间的每一个邮资值都能得到(当然还有8、9和12)；如果面值分别为1、3，则在1-7之间的每一个邮资值都能得到，且7就是可以得到连续的邮资最大值。

例如，当n=5和m=4时，面值为(1,3,11,15,32)的5种邮票可以贴出邮资的最大连续邮资区间是1到70。1,(1+1=2),…,(11+11+15+32=69),(3+3+32+32=70)

问题分析

- 基本思路：搜索所有可行解，找出最大连续邮资区间的方案。
- 解向量：用n元组x[1:n]表示n种不同的邮票面值，并约定从小到大排列。x[1]=1是唯一的选择。
- 可行性约束函数：已选定x[1:i-1]，最大连续邮资区间是1—r，则x[i]可取值范围是x[i-1]+1—r+1。

问题分析

如何确定r的值？
计算x[1:i]的最大连续邮资区间在算法中频繁用到，因此势必要找一个高效方法。考虑到直接递归求解的复杂度太高，尝试计算用不超过m张面值为x[1:i]的邮票贴出邮资k所需的最少邮票数y[k]。通过y[k]可很快推出r的值，y[k]可通过递推在O(n)时间内解决：

```
for (int j=0; j<= x[i-2]*(m-1);j++)
      if (y[j]<m)
        for (int k=1;k<=m-y[j];k++)
          if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;
while (y[r]<maxint) r++;
```


算法设计

```
int n,m,x[],bestx[],y[],maxint,maxl,maxvalue;
main ( )
  { int i;
    input(n,m);
    maxint=32767; maxl=1500;
    maxvalue=0;
    for(i=1;i<=n;i++) x[i]=0;
    for(i=1;i<=maxl;i++) y[i]=maxint;
    x[1]=1;y[0]=0;
    backtrack(2,1);
    return(bestx[],maxvalue);
  }
```



```
backtrack (int i,int r)
    { int j,k,z[];
      for (j=0; j<= x[i-2]*(m-1);j++)
        if (y[j]<m)
          for (k=1;k<=m-y[j];k++)
            if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;
      while (y[r]<maxint) r++;
      if(i>n){ if (r-1>maxvalue)
                    {maxvalue=r-1;
                      for(j=1;j<=n;j++) best[x]=x[j];}
               return; }
      for(k=1;k<=maxl;k++) z[k]=y[k];
      for(j=x[i-1]+1;j<=r;j++)
            { x[i]=j;
              backtrack(i+1,r);
              for(k=1;k<=maxl;k++) y[k]=z[k];}
    }
```