递归与回溯

Wang Houfeng
CCES, PKU
wanghf@pku.edu.cn

内容

- **递归回顾**
- 递归与回溯

递归

- 递归：嵌套调用中，存在自己调用自己的语句
  - 间接递归：

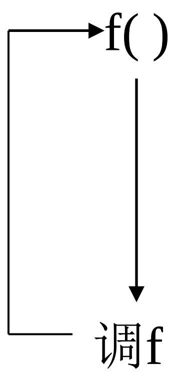
    A 调用 B，B 又调用 A 的方式
  - 直接递归：函数直接调用自身（A 调用 A）
- 递归的 2-Step 思想
  - 基始值（初始值）定义；
  - 归纳方法（向初始值靠拢）

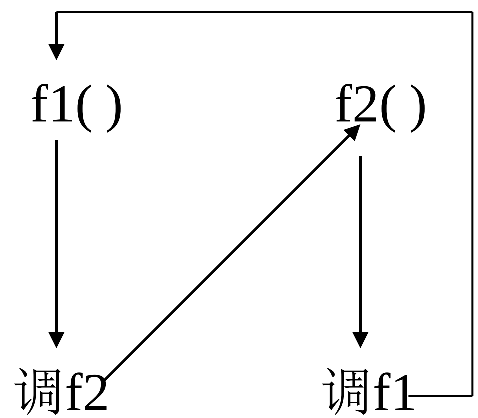
```
int  f(int x)
{    int y,z;
      ……
     z=f(y);
     …….
    return(2*z);
}
```

```
int  f1(int x)
{    int y,z;
      ……
     z=f2(y);
     …….
    return(2*z);
}
```

```
int  f2(int t)
{    int a,c;
      ……
     c=f1(a);
     …….
    return(3+c);
}
```

f 直接调用 f 自己





f 1 借助 f 2调用 f 1

- 说明
  - C编译系统对递归函数的自调用次数没有限制
  - 每调用函数一次，在内存堆栈区分配空间，用于存放函数变量、返回值等信息，递归次数过多，可能引起堆栈溢出

求解递归问题的方法

- 假定函数为 f
- 递归的步骤
  - 确定递归函数的参数，简记为，即f(n)
  - 确定如何由f(n-1)或者f(m)表示f(n)，其中 m小于 n
  - 确定初始条件，如 f(0) 、 f(1)等
- 例如：计算 1+2+…+n (n>0, n 为正整数)
- 定义：

$$f(n) = \begin{cases} 1, \text{ 当 } n=1 \text{ 时} & \text{//初始条件} \\ n+f(n-1), \text{ 当 } n > 1 \text{时} & \text{//}f(n)\text{与}f(n-1)\text{的关系} \end{cases}$$

递归的直观解释

- 计算 1+2+…+n (n>0, n 为正整数)
- 定义：

$$f(n) = \begin{cases} 1, & \text{当 } n=1 \text{ 时} \\ n+f(n-1), & \text{当 } n > 1 \text{时} \end{cases}$$

- 含义：
  - 要计算 f(n)，如果能先计算 f(n-1)，就能在基础上加上 n 而计算出 f(n)。或者，假定已经计算 f(n-1)，就可以计算 f(n)
  - 将计算 f(n) 转化为计算 f(n-1)

```
int f(int n)
{
  if (n=1) return 1;
 else return  (n + f(n-1));  // 向初始值 f(1) 逼近。
}

int main()
{  int n, y;
   cout<<"Input a integer number:"<<endl;
   cin>>n;
   cout<<"The result of f(n): "<<f(n)<<endl;
   return 0;
}
```

内容

- 递归回顾
- 递归与回溯

递归的两种常用方法

- 归纳式方法：直接求解
  - 关键点
    - 给定初始情况的解
    - 给出归纳式：如给出 f(n) 与 f(n-1)之间的关系
  - 典型问题：求n!，汉诺塔等
- 回溯方法：**试探性寻解**
  - 关键点：在第n步的情况下，试探第n+1步，**第n+1步有多种情况**（每完成一种，还需返回再解下一种…）
  - 典型问题：走迷宫、8-皇后等
  - 回溯是计算机解题中常用的算法，有很多问题无法用简单归纳式求解，需要利用试探与回溯技术

回溯与枚举

- 枚举
  - 从所有候选答案中去搜索正确的答案（逐一检测）
  - 候选答案的范围在求解之前可用一个确定的集合表示，如：用一个n元组$(x_1,\ldots,x_n)$来表示，其中的x_i取值来自于某个有穷集 S_i
  - 假设$|S_i| = m_i$，枚举方式需要从$m= m_1m_2\ldots m_n$ 个候选中确定满足解要求的向量集合
- 回溯
  - 一种**简化的枚举搜索**，避免不必要的搜索（**尽量减少不必要的枚举**）

回溯基本思想

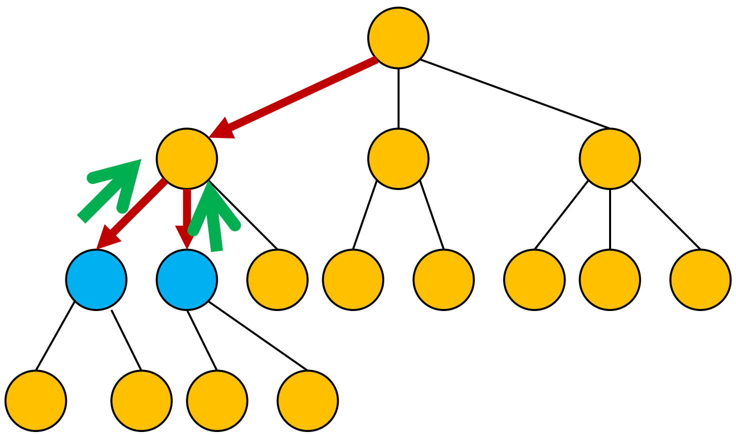
- 回溯是一种探索式的控制策略
  - 为了求得问题的解，先选择某一种可能情况向前探索，在探索过程中，一旦发现该选择错误(或存在多个解)，就退回一步作新的选择，继续向前探索，如此反复，直至得到解或证明无解（避免沿错误路径继续探索—尽量不走错误的分支）
- 回溯的两种情况
  - **找一个解**：先选择一条路径，一步步向前试探，如果发现“此路不通”则返回后再试探其它途径，直至找到一条成功路径为止
  - **找所有解**：向前试探过程中，即便找到了成功的路径，也需要回过头来试探是否有其它成功路径，直至找到所有成功路径为止

如何避免不必要搜索

- 通过一种不断修改的规范函数$P_i(x_1,\ldots,x_i)$去测试正在构造的n元组的部分向量$(x_1,\ldots,x_i)$，看是否可能导致目标解。
- 如果判定$(x_1,\ldots,x_i)$不能导致目标解，那么就将可能要测试的$m_{i+1}\ldots m_n$个分量略去（m_i表示x_i的所有可能取值数）。
- 回溯法的测试次数比硬性处理（完全枚举）的测试次数要少得多!

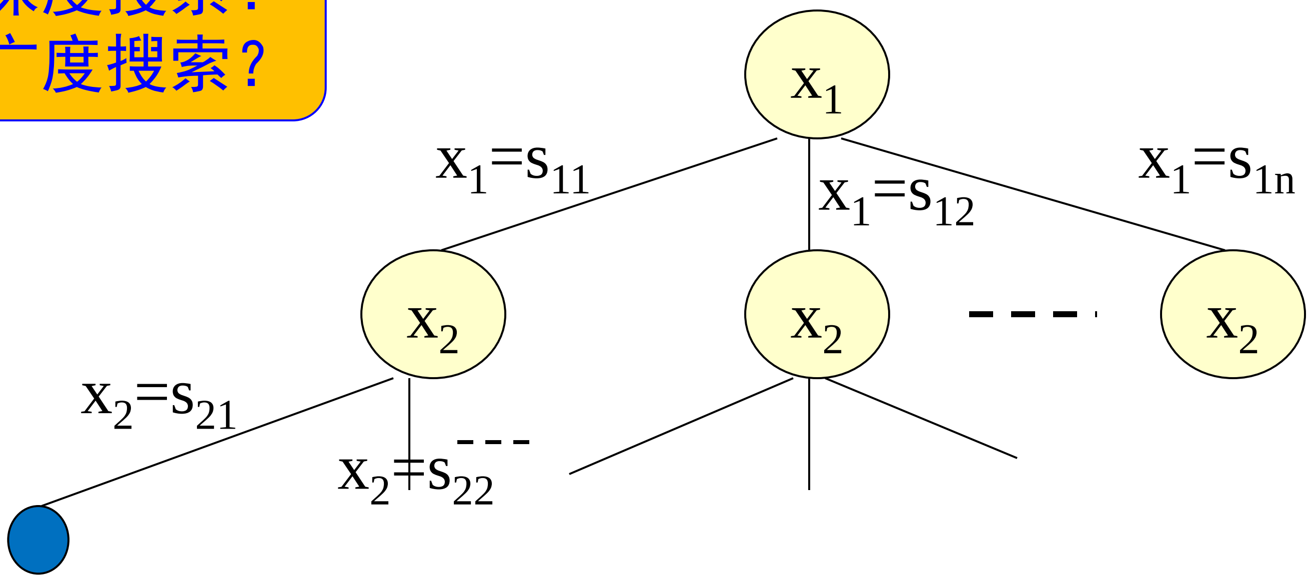
回溯的基本思想

- 回溯思想总结
  - 将候选集以**解空间树**的形式表示
  - 回溯过程：在解空间树中，按**深度优先**策略，从根出发搜索解空间树。每次搜索至解空间树的任意一点时，先判断该结点是否可能包含解；如果肯定不包含，则跳过对该结点为根的子树搜索，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
  - 回溯算法本身是在不断试探，也称为探索法



搜索空间-树结构

深度搜索？
广度搜索？



避免不必要搜索（肯定找不到答案的搜索）称为**剪枝**

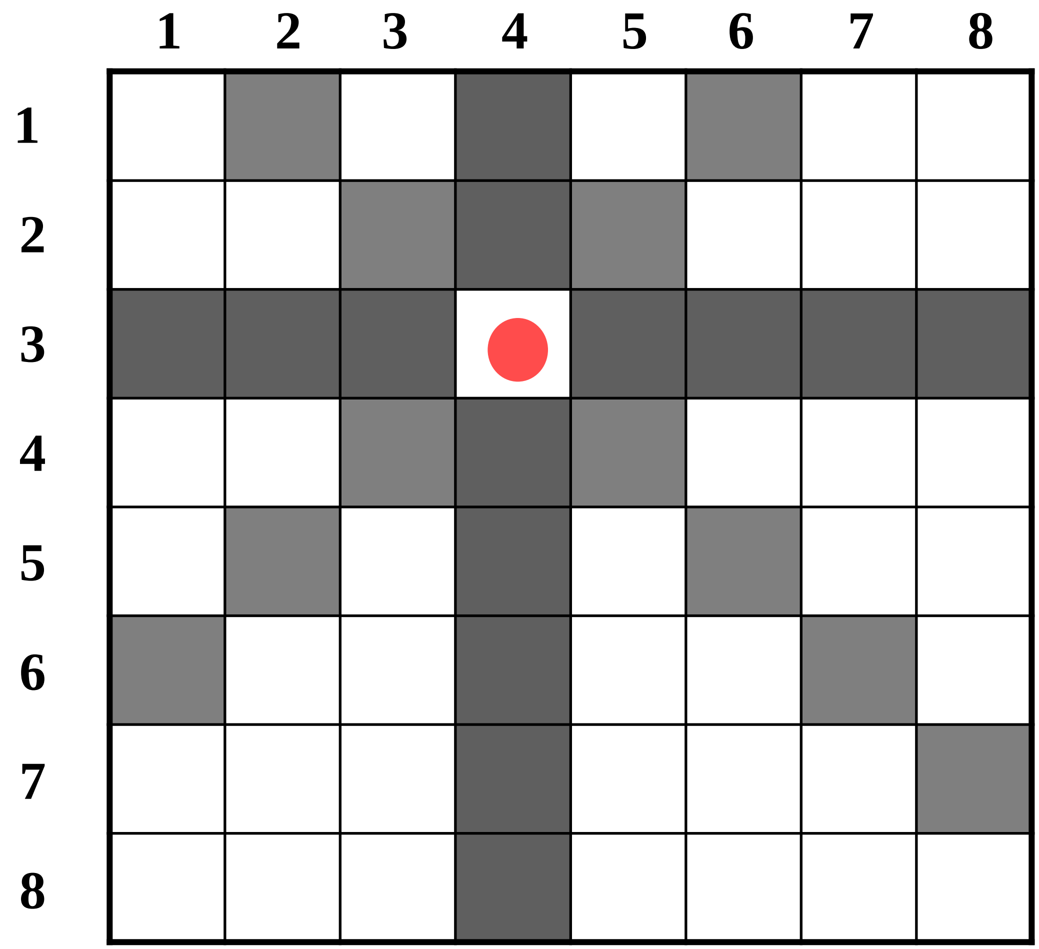
常用剪枝方法：

用**约束函数**在搜索下层结点时剪去不满足约束的子树；

例1：8-皇后问题

- 背景：
  – 国际象棋的皇后可以走水平、垂直、斜线，若在一个皇后可以走动的范围内有其他棋子，则皇后可以吃掉这个棋子（产生攻击）
- 问题：
  – 如何在棋盘上摆放8个皇后，使得每个皇后都没有被吃掉的危险。
  – 换言之，当要摆放一个新的皇后时，摆放位置的行、列、左右对角线都不能有其它棋子。

8-皇后问题图示



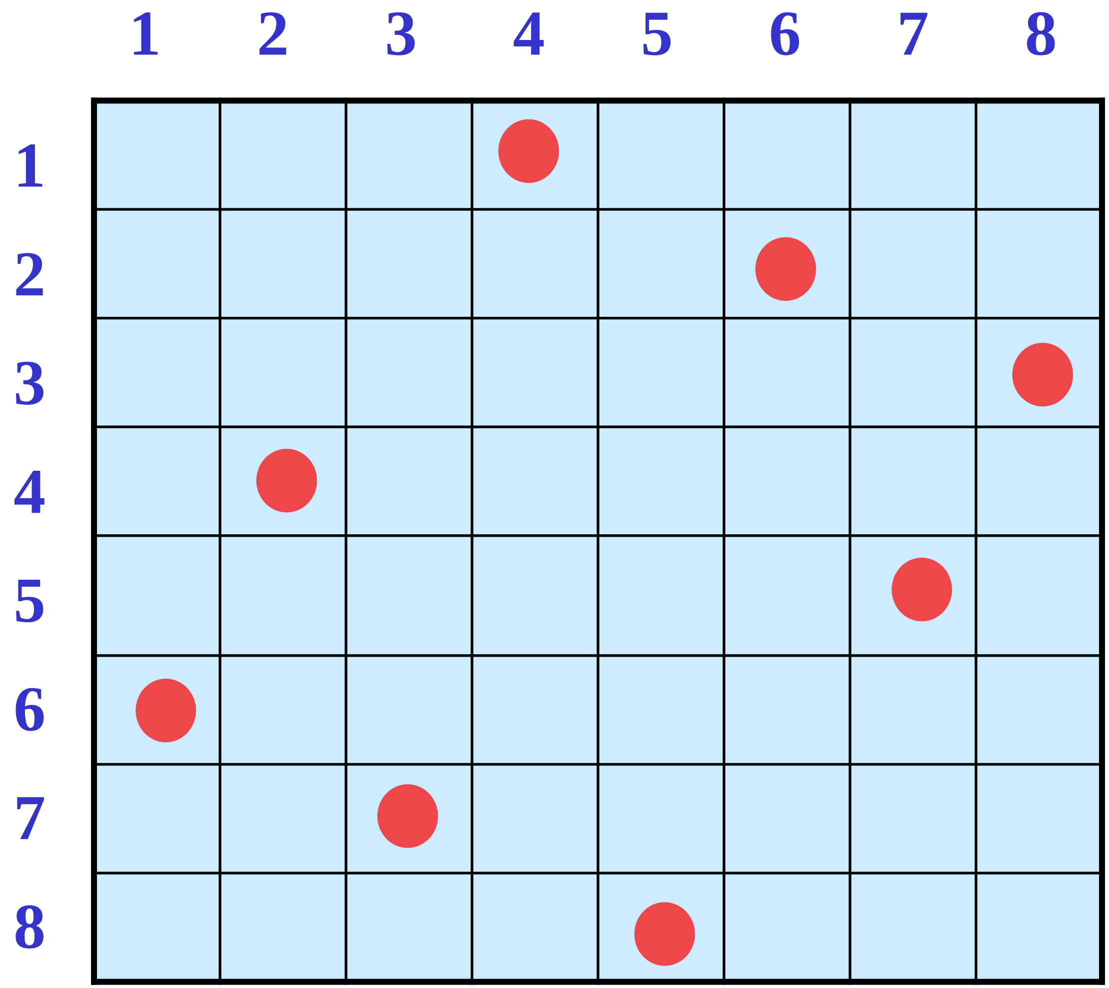
8-皇后问题的解表示

- **8皇后问题的解**
  - 8皇后问题可以表示为8-元组($x_1,\ldots,x_8$)，其中x_i是放置皇后 i 所在的列号（皇后在第 i 行的列号）
- **约束条件**
  - 显式约束条件是每个 皇后 x_i 的取值为：

    $x_i \in s_i$ ={1,2,3,4,5,6,7,8}, 1≤i≤8，共8个值，解空间的大小为：**8^8个**
  - 隐式约束条件是，没有两个x_i可以相同且没有两个皇后可以在同一条斜角线上

    没有2列相同后，解空间缩小到： 8！ 个

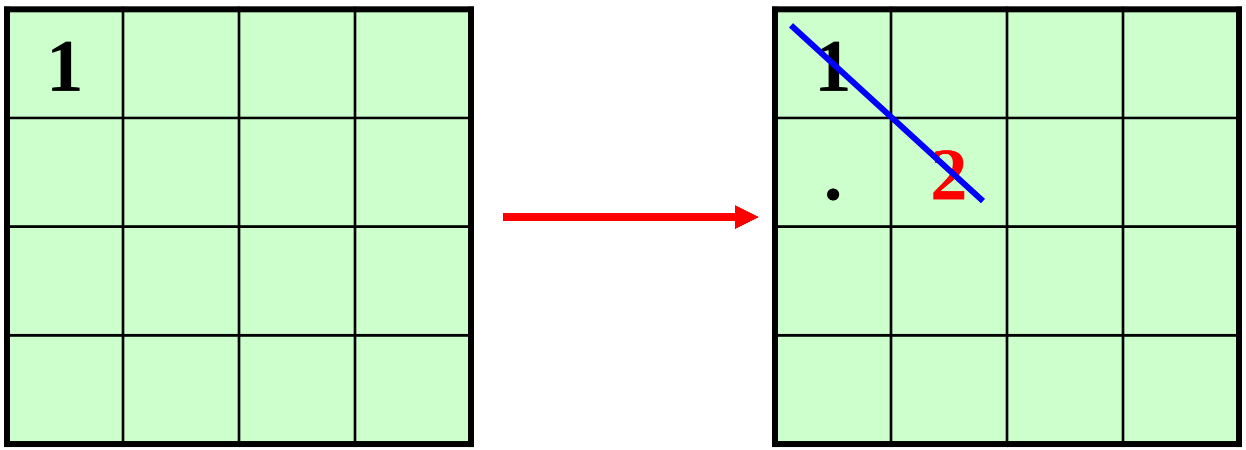
8-皇后问题的一种解



方法说明

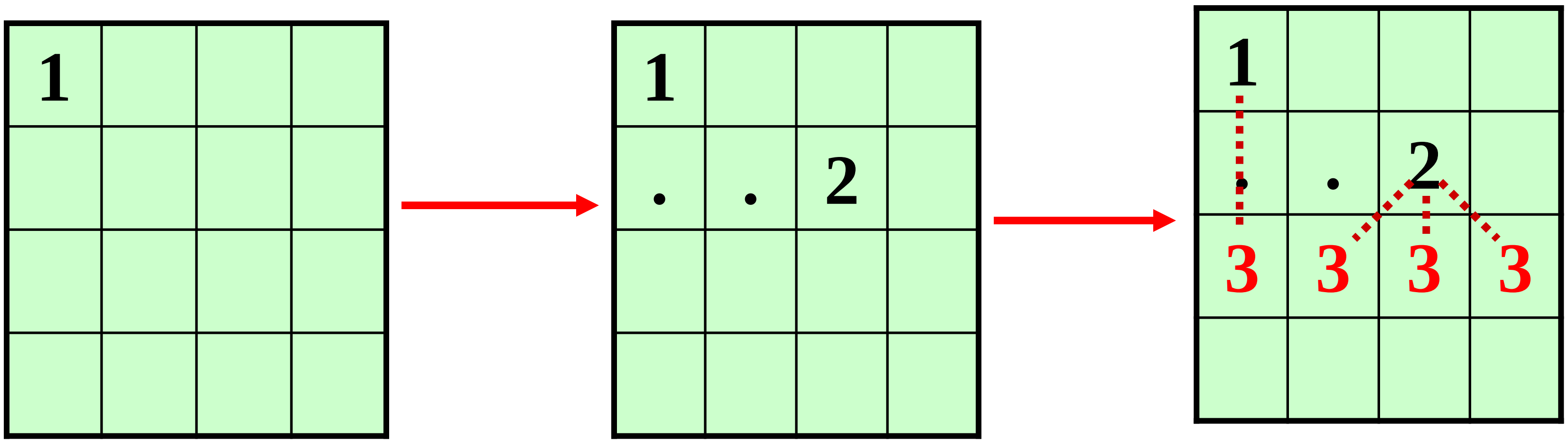
- 在所有解的解空间中，从一点(称为根)出发搜索
- 搜索至某一结点时，总是先判断该结点是否肯定不包含问题的解。如果肯定不包含，则跳过该结点;如果跳过所有节点都不行，则逐层向其祖先结点回溯。否则，继续进行搜索。

以更为简单的4-皇后为例说明

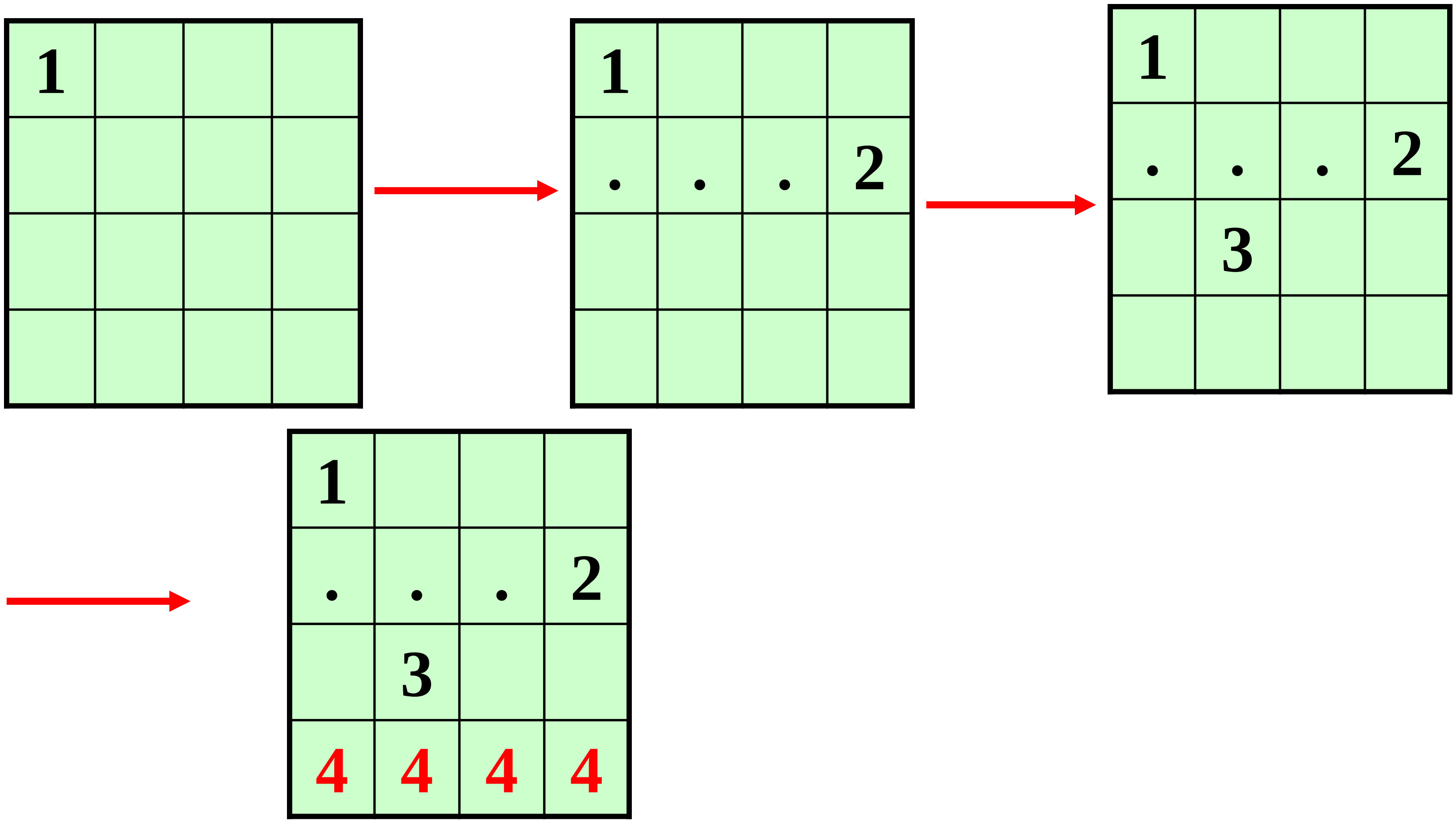


第1个皇后在第1行，有4个位置,先选择第1列;紧接着,选择第2个皇后的位置。第2个皇后不能放在第1列，将其放在第2列上，但此时，由于第1个皇后在同一条对角线上，还是会被杀死。于是，将第2个皇后放在第3列上;

4-皇后

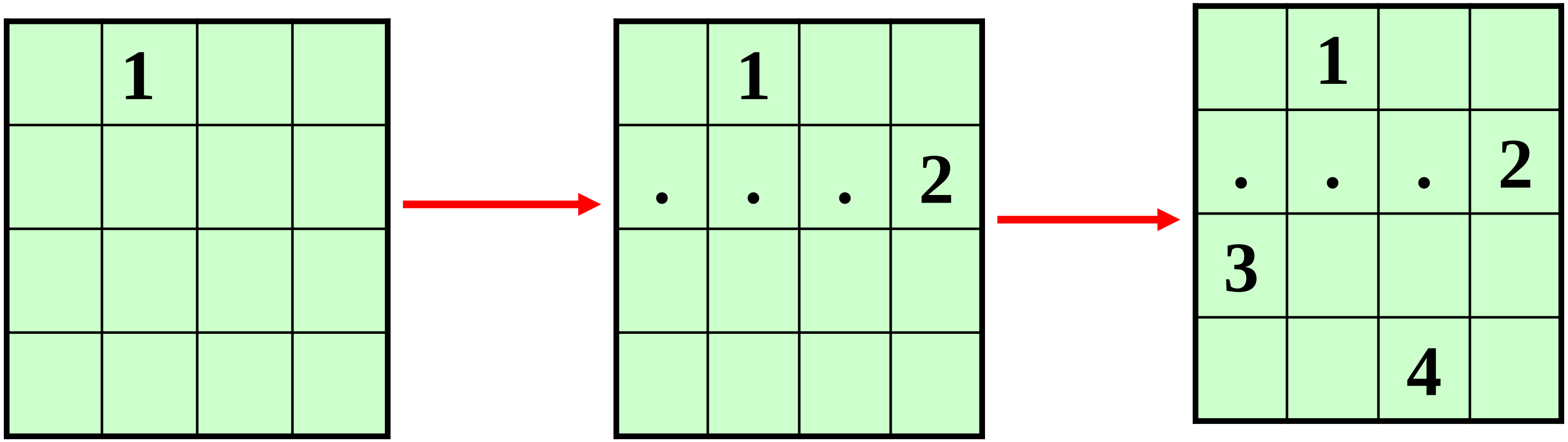


- 当第2个皇后放在第3列后，从第1列到第4列依次考虑第3个皇后的位置，显然，第1、2、3、4列均无法摆放（**此时，第4行不必再考虑，可以剪掉**），再往上返回到第2行，向后调整第2个皇后的位置



- 在第1、2个皇后的位置确定后，第3个皇后只有1个位置可以放（第3行第2列）。但此时，第4个皇后又没有位置可以摆放！往上层回，由于第3个皇后没有其他合适位置，只好回溯到第2个皇后，此时，第2个皇后也没有合适位置（已到最后），只能回溯到第1个皇后调整位置到第2列。

4-皇后



- 第1个皇后向后调整到第2列后，第2个皇后只有1种位置（第4列），第3个皇后也只有1个位置（第1列），第4个皇后也只有1个位置。
- 得到解！

回溯算法

- **算法的三个步骤：**
  - **针对所给问题，定义问题的解空间；**
    - **应用回溯法求解问题时，首先应明确定义问题的解空间。** 8皇后问题可以表示为8-元组($x_1,\ldots,x_8$)，其中x_i是放置第i个皇后所在的列号。列号值为：1-8；
  - **确定易于搜索的解空间结构；**
  - **搜索解空间，并且在搜索过程中及时判断、避免无效搜索(剪枝)；**
- **回溯法适合递归求解。**

n-皇后

- 8-皇后问题实际上很容易一般化为n-皇后问题，即要求找出一个n*n棋盘上放置n个皇后并使其不能互相攻击的所有方案
- 令$(x_1,x_2,\ldots,x_n)$表示一个解，由于没有两个皇后可以放入同一列，因此所有的x_i将互不相同
- 那么关键是应如何去测试两个皇后是否在同一条斜角线上呢？

对角线判断

- 正向：(行 - 列)值相同
- 反向：(行 +列)值相同
- 假设有两个皇后被放置在(i，j)和(k，l)位置上，那么根据以上所述，对角线条件：

$i - j = k - l$ 或 $i + j = k + l$

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}
a_{61}	A_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}

- 将这两个等式分别变换成：$j - l = i - k$ 或 $j - l = k - i$
- 因此，当且仅当 $|j- l| = |i-k|$ 时，两个皇后在同一条斜角线上。

8-皇后：冲突判断

```
int check(int line,  int list, int x[])   //第 line 个皇后放在list列是否与前面冲突
{    int i=0;
     while (i<line)
     {
            if ((x[i]==list) || (abs(x[i]-list)==abs(i-line) ))
                       return  0 ; // 发生冲突
            i←i+1     // 继续检查直至查完
     }
     return  1;     // 所有查完都没发现冲突
}
```

列相同

对角线相同

试探与回溯

```
Queen (int line, int x[])
{
    for (int list=0; list<8; list++)
    {
        if (check(line, list, x))    //不冲突
         {  x[line]=list;     //记录当前行的列号
            if (line==7)    //如果8个皇后均不冲突，则输出
                print();   //大家自己实现
            else   Queen(line+1, x); //继续判断下一皇后(递归)
            x[line]=0; //该位置都要重新归0，以便重复使用
         }
    }//for
}
```

8皇后

主程序：

```
int main ()
{
    int x[8]={0};    //8-皇后列号
    Queen(0, x);
    return 0;
}
```

问题1：如何只得到1组解？

问题1：如何不产生同构解？（旋转90°/180° 相同 ）

共92组解，部分答案如下：

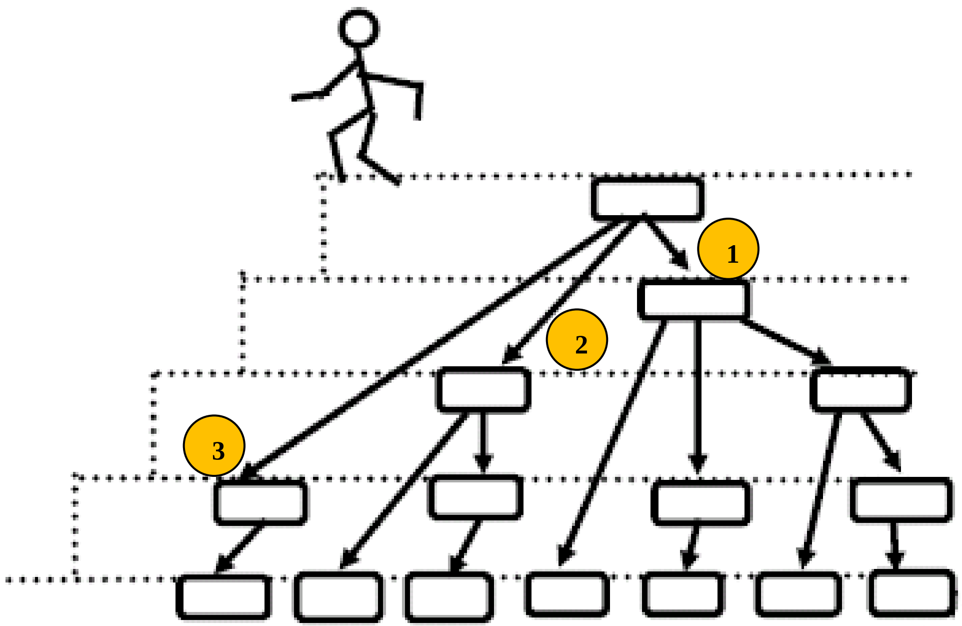
方案1：1 5 8 6 3 7 2 4
方案2：1 6 8 3 7 4 2 5
方案3：1 7 4 6 8 2 5 3
方案4：1 7 5 8 2 4 6 3
方案5：2 4 6 8 3 1 7 5
方案6：2 5 7 1 3 8 6 4
方案7：2 5 7 4 1 8 6 3
方案8：2 6 1 7 4 8 3 5
方案9：2 6 8 3 1 4 7 5
方案10：2 7 3 6 8 5 1 4
…

思考题

- 跳马：
  - 在 n*n 的棋盘上的某个位置(x,y)的马可否游历整个棋盘的每个位置且仅经过一次。
  - 注意：马行“日”字（加减2行1列或加减2列1行，但必须在棋盘范围内）

例2：下楼问题

- 问题：
  - 从楼上走到楼下共有h个台阶，每一步有3种走法：走1个台阶；走2个台阶；走3个台阶。问可以走出多少种方案？将所有的方案输出。



下楼问题分析

- 问题分析
  - 需枚举出所有的可能方案，是一个典型的递归回溯问题
  - 用向量表示解，表示为 take[n]，分量表示第n步的台阶数
  - take[s]记录第s步走几个台阶，即1，2，3中的哪一个
  - 当走到底时，take[1]~take[s]，就是一路走来的过程，即一种成功的解向量（s可能是变化的）。
  - 注意：成功的走法可能有不同长度的解向量（可用固定长度的解向量表示：后面部分状态值为0）

下楼问题的核心

- 首先确定试探的含义

  $Try^n()$ 表示什么意思？

走完第n步的状态，即，已经填写完第n个take[ ]

- 随后需要确定

  $Try^n()$与$Try^{n+1}()$ 之间的关系是什么

$Try^n()$与$Try^{n+1}()$ 间的关系

- 在走完第n步后再走第n+1步时：
  有三种选择（走1、2、3个台阶）
  - 每个选择下有三种可能性：
    - 如果剩下的台阶小于想要走的步数
      - 返回
    - 如果剩下的台阶恰好等于要走的步数；
      - 打印输出；
    - 如果剩下的台阶大于想要走的步数
      - 继续走下去；

$Try^n()$ 的参数确定

- $Try^n()$与$Try^{n+1}()$ 之间哪些数据是不一样的？而且，是需要由$Try^n()$传递给$Try^{n+1}()$ 的？
  - $Try^n()$表示走完第n步的情况，$Try^{n+1}()$表示走第n+1步的情况
  - 走完n步后剩余台阶数，需要由$Try^n()$传递给$Try^{n+1}()$
- 因此，将$Try^n()$可以定义为：

Try(i, s)　i表示剩余的台阶数，s表示当前待走的步数

Try( ) 的功能

Try(i, s)

for（j=1; j<=3; j++) // 3种可能性

- 剩余台阶数 i < j
  - 第s步走的台阶比剩下的阶梯数还多，j不可取。
- 剩余台阶数 i ＝＝ j
  - 第s步正好走完剩下的阶梯，**得到一个解决方案**。
- 剩余台阶数 i > j
  - 第s步走完后，还**剩下i-j级阶梯没有走**，可以走第s+1步。**递归调用 Try(i-j, s+1)**

```
int take[99];
int num = 0;                //num表示解决方案的总数
void Try(int i, int s)
{   //i表示所剩台阶数
    for (int j = 3; j > 0; j--) //枚举第s步走的台阶数j
    {
      if (i<j)                  //如果所剩台阶数i小于允许走的台阶数j
            continue;
      take[s] = j;                  //记录第s步走j个台阶；
      if (i == j)                   //如果已经走完全部台阶；
      {
            num++;  //方案数加1
            cout << "solution" << num << ": ";
            for (int k = 1; k <= s; k++)
                  cout << take[k];
            cout << endl;
      }
      else
            Try(i - j, s + 1); //尚未走到楼下
    }
}
```

主函数

```
int main()
{
    int h = 0;
    cout << "how many stairs : ";
    cin >> h;
    Try(h,1);   //有h级台阶要走，从第一步开始走
    cout << "there are " << num << " solutions."
    << endl;
    return 0;
}
```

例子3：分书

- 问题
  - 有编号分别为1，2，3，4，5的五本书，准备分给A,B,C,D,E五个人，每个人阅读兴趣用一个二维数组加以描述：

$$Like[i][j] = \begin{cases} 1 & i\text{喜欢书}j \\ 0 & i\text{不喜欢书}j \end{cases}$$

人＼书	0	1	2	3	4
A	0	0	1	1	0
B	1	1	0	0	1
C	0	1	1	0	1
D	0	0	0	1	0
E	0	1	0	0	1

  - 请写一个程序，输出所有分书方案，让人人皆大欢喜。

分析

- 基本思路：
  ① 试着给第i个人分书，先试分0号书，再分1号书，分2号书…，分4号书
  ② 当“第i个人喜欢j书，且j书尚未被分走”时。第i个人能够得到第j本书
  ③ 如不满足上述条件，什么也不做（循环返回条件）
  ④ 如满足条件，则做3件事：
    - 分书：将j书分给i，同时记录j书已被选用；
    - 判断：查看是否将所有5个人所要的书分完，
      – 若分完，则输出每个人所得之书；
      – 若未分完，去寻找其他解决方案；
    - 回溯：让第i人退回j书，恢复j书尚未被选的标志。

重要数据表示

1、使用二维数组定义阅读喜好用：

– int like[5][5]

={{0,0,1,1,0},{1,1,0,0,1,},{0,1,1,0,1},{0,0,0,1,0}{
0,1,0,0,1}};

2、使用数组book[5]记录书是否已被选用

int book[5]={0,0,0,0,0}; //初始化

3、使用数组take[5]存放第几个人领到了第几本书

```
void trybook(int i) {     //第i个人
    for (int j=0; j<=4; j=j+1)         //对于每本书，j为书号；
    {
      if ((like[i][j]>0)&&(book[j]==0))
                              //若第i个人喜欢第j本书，且这本书没有被分出；
      {
              take[i]=j; //把第j号书分给第i个人
              book[j]=1;          //标记第j号书已被分出
              if (i==4)
              //若第5个人也已经拿到了书，则书已分完，输出分书方案
              {
                      n = n + 1;//让方案数加1
                      cout <<"第"<<n<<"个方案"<<endl;
                      for (int k=0; k<=4; k=k+1)
                              cout<<take[k]<<"号书给"<<char(k+65);
                      cout <<endl;
              }
              else                //若书还没分完,继续给下一个人找书；
              {
                      trybook(i+1);
              }
              book[j]=0;          //回溯，把书标记为未分，找其他解决方案；
      }
    }
}
```

'A'

分书问题

```
#include<iostream.h>
int
   like[5][5]={{0,0,1,1,0},{1,1,0,0,1,},{0,1,1,0,1},{0,0,0,1,
   0},{0,1,0,0,1}};
int book[5], take[5], n;        //n表示分书方案的总数
int main()
{
   int n=0;                //分书方案数预置0
   trybook(0);             //从“为第0个人分书”开始执行
   return 0;
}
```

例子4：子集和数

- **子集和数问题：** 已知n+1个正数，w_i 1≤i≤n和M。要求找出w_i的和数是M的所有子集。
- 例如，若n=4 , (w_1, w_2, w_3, w_4)=(11,13,24,7) , M=31 ,则满足要求的子集是（11，13，7）和（24，7）。
- 如果通过给出其和数为M的那些w_i的下标来表示解向量，则比直接用这些w_i表示解向量更为方便。因此这两个解就由向量（1，2，4）和（3，4）所描述。**解向量表示！**

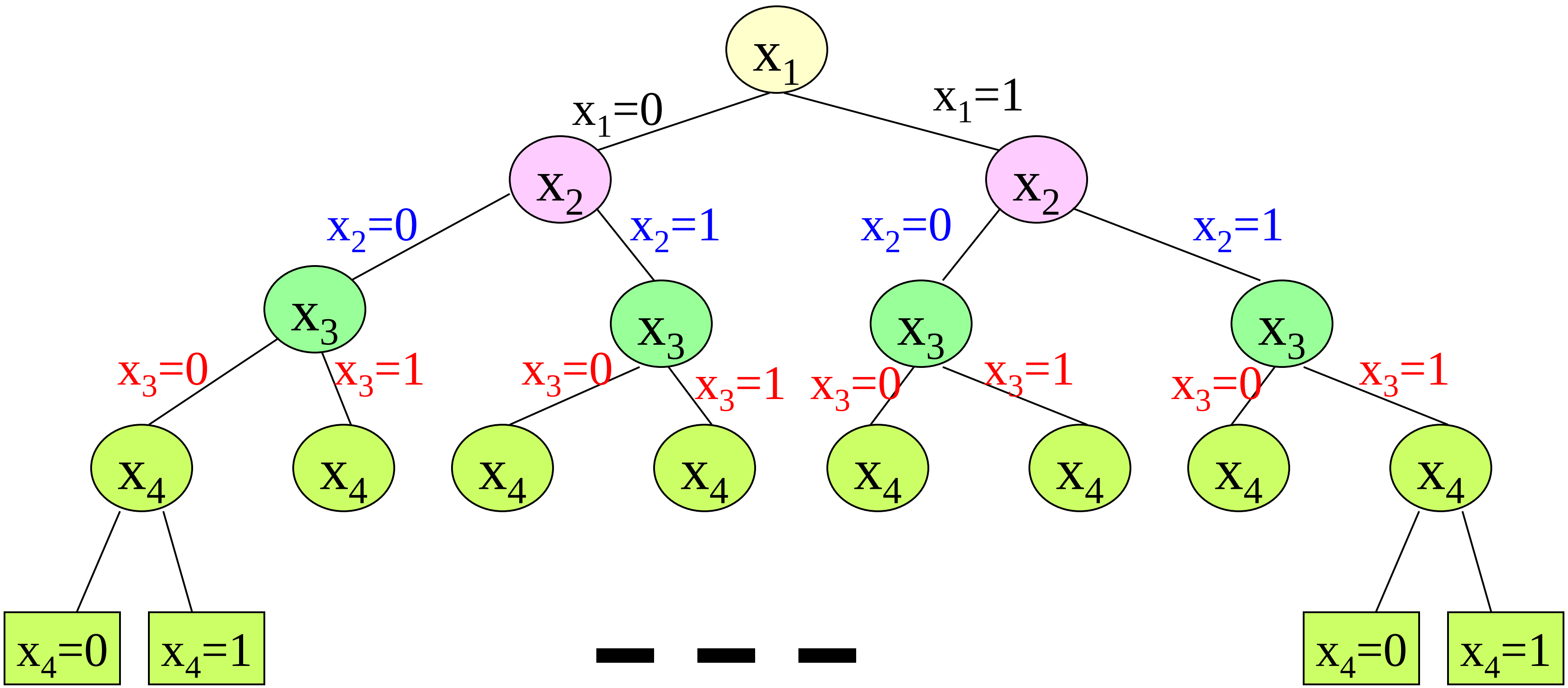
子集和数问题

- **解的表示：** $(x_1,\ldots,x_{j},\ldots)$
- **显式约束条件**要求$x_i\in\{j \mid j$是w_j的下标值，$1\leqslant j\leqslant n\}$。
- **隐式约束条件**则要求没有两个x_i是相同的且相应的w_i和数等于M。
- 为了避免产生同一个子集的重复情况（例如：(1, 2, 4)和(1, 4, 2)表示同一个子集），**附加另一个隐式约束条件**：$x_j\leqslant x_{j+1}$，$1\leqslant j\leqslant n$。
- 注意：在上述解的表示中，向量长度不固定。

子集和数问题

- 子集和数问题的另一种列式表示是，每一个解的子集由这样一个**n-元组($x_1,\ldots,x_n$)**所表示，它使得$x_i \in \{0, 1\}, 1 \leqslant i \leqslant n$。如果没有选择$w_i$，则$x_i=0$；否则$x_i=1$。
- 于是上例的解可以表示为(1, 1, 0, 1)和(0, 0, 1, 1)。
- 用固定长度的元组表示所有的解。
- 一个问题的解可以有多种表示形式！

解空间的表示



- 解空间总可以用树形结构表示。对应的变量有几种取值，就可以分成几个树枝。

限制不必要的搜索

解有效，当且仅当： $\sum_{i=1}^{i=k} x_i w_i + \sum_{i=k+1}^{i=n} w_i \geq M$

当**W[i]**以递增次序排列时，约束函数可强化为：

$$\sum_{i=1}^{i=k} x_i w_i + w_{k+1} > M \longrightarrow$$ 不可行解

于是：有效搜索的条件为：

$$B(x_1, x_2, ..., x_k)$$
$$= (\sum_{i=1}^{i=k} x_i w_i + \sum_{i=k+1}^{i=n} w_i \geq M) \&\& (\sum_{i=1}^{i=k} x_i w_i + w_{k+1} \leq M)$$

算法

$$Let \quad s=\sum_{i=0}^{i=k-1} x_i w_i, \quad r=\sum_{i=k}^{i=n-1} w_i$$

```
sub_sum(int s, int k, int r)
{   int j;
    if(k<n)  // n 是集合大小,为全局变量
    {      X[k]=1;    // X 数组和 W 数组为全局变量. X 初始值为 0 ;
           if (s+ W[k] ==M)  输出 (X[j], j=1 to k);
           else   if (s+ W[k]+W[k+1]< =M)
                  sub_sum(s+ W[k],  k+1 ,  r- W[k]);   //取第k个数
           if ((s+r- W[k]>=M) && (s+ W[k+1]<=M))   // B_k=true//
           {           X[k]=0;
                       sub_sum(s ,  k+1, r- W[k]);   //不取第k个数
           }
    }
}
```

题目扩展：可放回取样

- 题目:
  - 给定具有n个不同元素的整数集合A和另一个整数值M，问，可否从A中取最多K个元素（所取元素可以相同），使得所取的元素和为M。
- 解向量:
  - **n-元组($x_1,\ldots,x_n$)所表示，其中 $x_i \in \{0, 1, \ldots k\}$, 1≤i≤n。**
  - **增加约束条件:**

$$\sum_{i=1}^{i=n-1} x_i \le K$$

例子5：全排列

- 问题
  - 从键盘读入一个英文单词（全部字母小写，且该单词中各个字母均不相同），输出该单词英文字母的所有排列（全排列）；
  - 例如，输入abc，则打印出：

    abc
    acb
    bac
    bca
    cab
    cba

全排列问题分析

- 反复做的事情：选择第n个位置的字母
  - **依次检查 输入单词中的字母**

    如果某个字母未被选择过；

    1. 将该字母选入字符串；
    2. 标记该字母已经被选择；
    3. 如果全部位置都已选完，打印输出；

       否则，为下一个位置选择字母；
    4. 把刚刚标记过的字母重新标记为“未选择”；

全排列问题求解思路

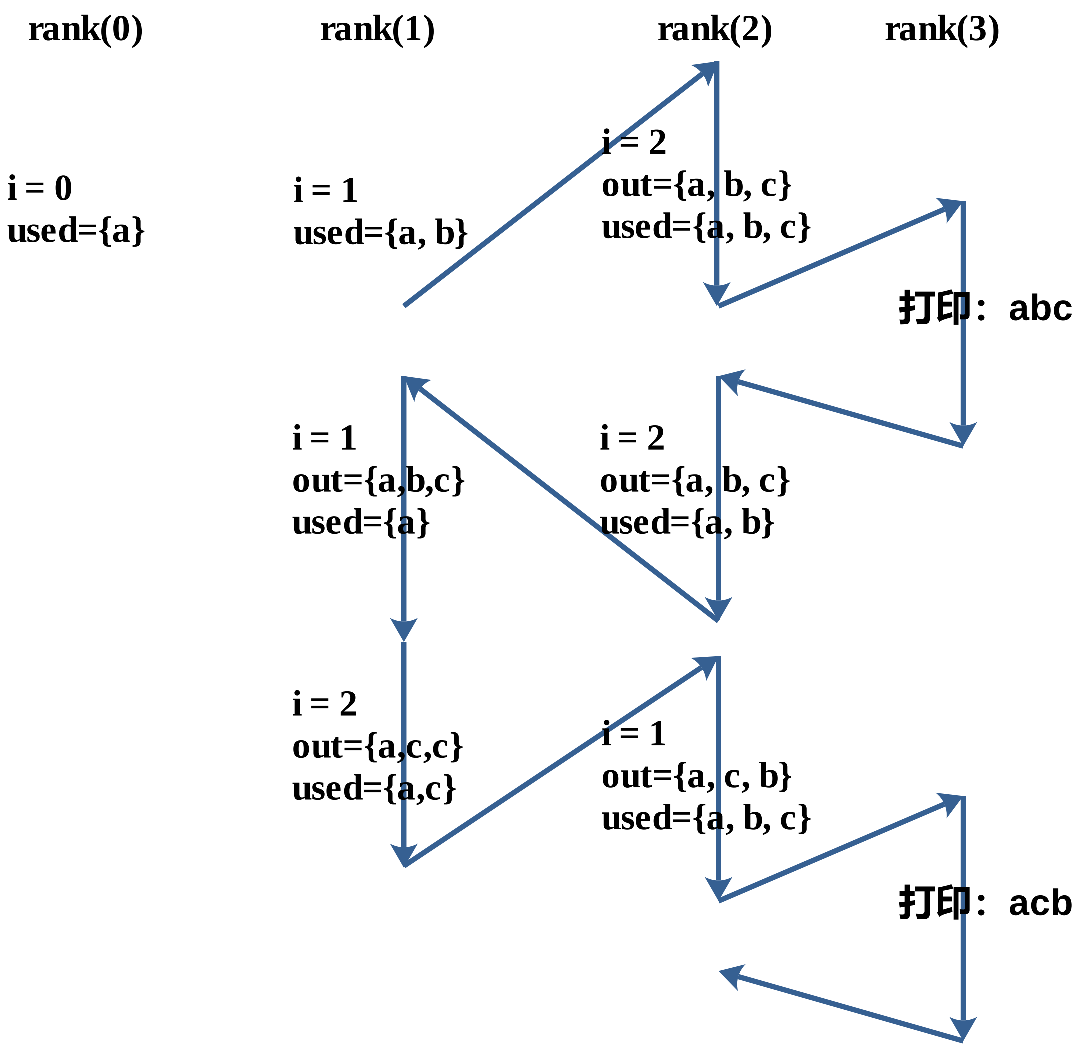
- 假设用函数ranker( )能够完成上述事情;
- ranker(n):为第n个位置选择字母
- 每次调用之间的区别在于位置n
  - ranker(1)
  - ranker(2)
  - ranker(3)
  - ranker(4)
  - …

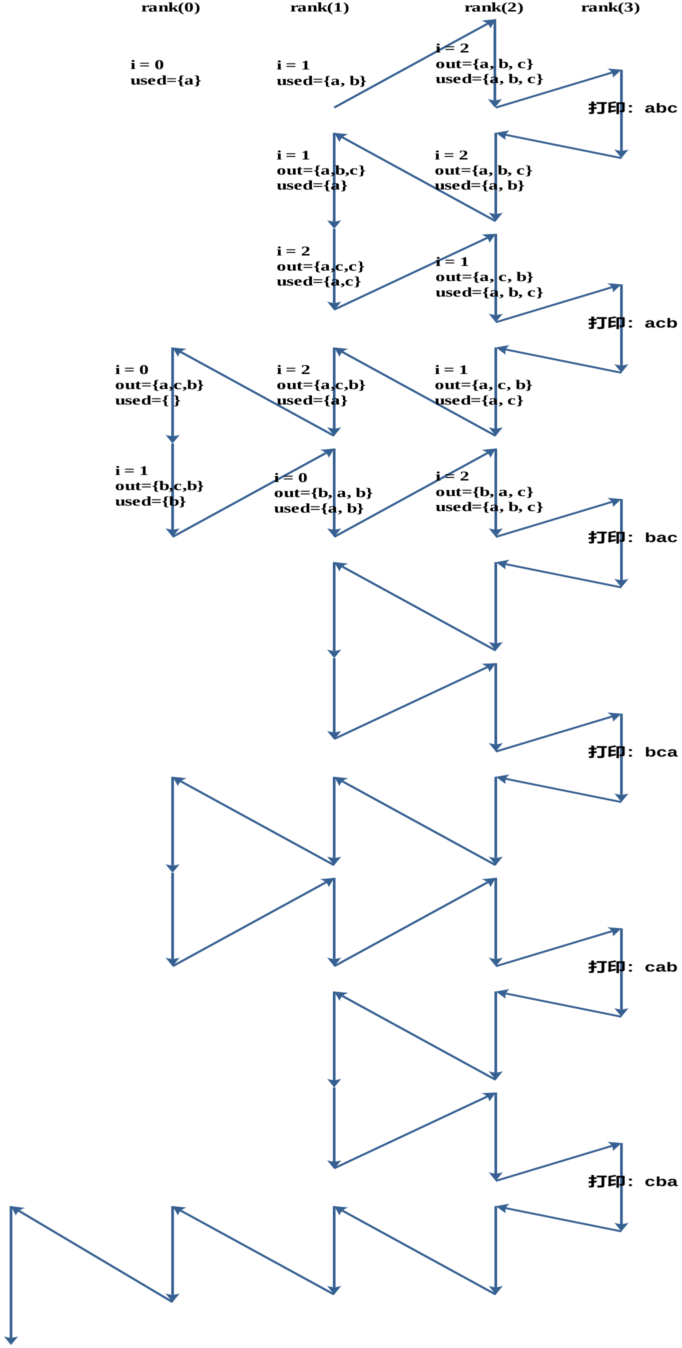
需要记录哪些字母在前面已经选过

算法

```
char in[30] = {0};            //存放输入的单词
char out[30] = {0};           //存放准备输出的字符串(一种排列)
int used[30] = {0};           //记录哪个字母已经使用过
int length = 0;
void rank(int n)              //为第n个位置寻找字母
{
    if (n == length)  {cout << out << endl;  return;}
    //如果新字符串中已经有length=3个字母，打印输出
    for (int i = 0; i < length; i++)  //依次查看每个字母
    {
            if (!used[i])           //如果某个字母尚未被选
            {
                    out[n] = in[i];   //选入该字母
                    used[i] = 1;      //标记该字母已经被选择
                    rank(n+1);
                    //否则，为下一个位置寻找字母
                    used[i] = 0;
                    //标记该字母还未被选择，使其重新可被选
            }
    }
}
```

全排列问题求解思路


rank(0)
rank(1)
rank(2)
rank(3)
i = 0
used={a}
i = 1
used={a, b}
i = 2
out={a, b, c}
used={a, b, c}
打印：abc
i = 1
out={a,b,c}
used={a}
i = 2
out={a, b, c}
used={a, b}
i = 2
out={a,c,c}
used={a,c}
i = 1
out={a, c, b}
used={a, b, c}
打印：acb




```
#include<iostream>
using namespace std;
char in[30] = {0};
char out[30] = {0};
int used[30] = {0};
int length = 0;
void ranker(int n)
{
    if (n==length) { cout<<out<<endl; return;}
    for (int i = 0; i < length; i++)
    {
        if (!used[i])
        {
            out[n]=in[i];
            used[i] = 1;
            ranker(n+1);
            used[i] = 0;
        }
    }
}
int main()
{
    cin >> in;
    length = strlen(in);
    ranker(0);//从第一个字母开始
    return 0;
}
```

回溯方法总结

- 站在第n步的状态下进行分析
  - 递归
    - 在第n步的情况下，枚举出第n+1步的所有可能，向所有可能的方法形成递归（对每一种可能性进行探索）
  - 回溯
    - 恢复影响以后的选择“现场”（使重新的选择成为可能）

探索过程

- 第n步需要做什么？
  - 对于面前的每种选择

① 把该做的事情做了！

② 判定是否得到解！

③ 递归（调用第n+1步）！

④ 看是否需要回溯！