```
/* D. Osovlanski & B. Nissenbaum, 1990 */
int v,i,j,k,l,s,a[99];void main(){for(scanf("%d",&s);*a-s;
v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(
!l<<!j)," .Q"[l^v?(l^j)&1:2])&&++l||a[i]<s&&v&&v-i+j&&v+i-
j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);}
```

4.栈与队列

（x2）试探回溯法：八皇后

```
/* Andor@net9.org, 2002 */
#define q(o)a[j]o[j+i+7]o[j-i+31]
a[39];
main(i,j)
{ for(j=9;--j;i>8?printf("%10d",a[j]):q(|a)||(q(=a)=i,main(i+1),q(=a)=0)); }
```

邓俊辉

deng@tsinghua.edu.cn

指数爆炸

- 很多问题的解，形式上都可看作若干元素按特定次序构成的序列

  以TSP问题为例，即给定n个城市之间总成本最低的环游路线

- 每一排列组合都是一个候选解，往往构成一个极大的搜索空间

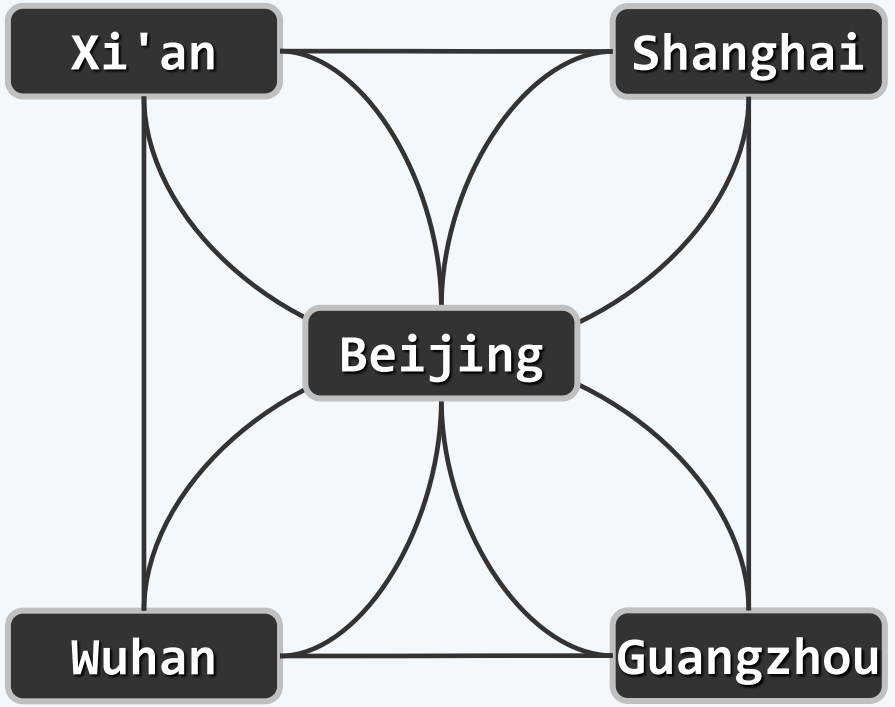
  仍以TSP为例，共有：

  $$n!/n = (n-1)! = O(n^n)$$

- 若采用蛮力策略求解

  需逐一生成可能的候选解，并检查其是否合理

  如此，必然无法将时间复杂度控制在多项式以内



试探-回溯-剪枝

- 为尽可能多、尽可能早地排除候选解，须深刻理解应用问题，并利用其特有的规律
- 事实上，根据候选解的某种局部特征，即可判断其是否合理

  此时只要策略得当，便可成批地排除候选解

  此即所谓剪枝（pruning）
- 试探回溯（probe-backtrack）模式

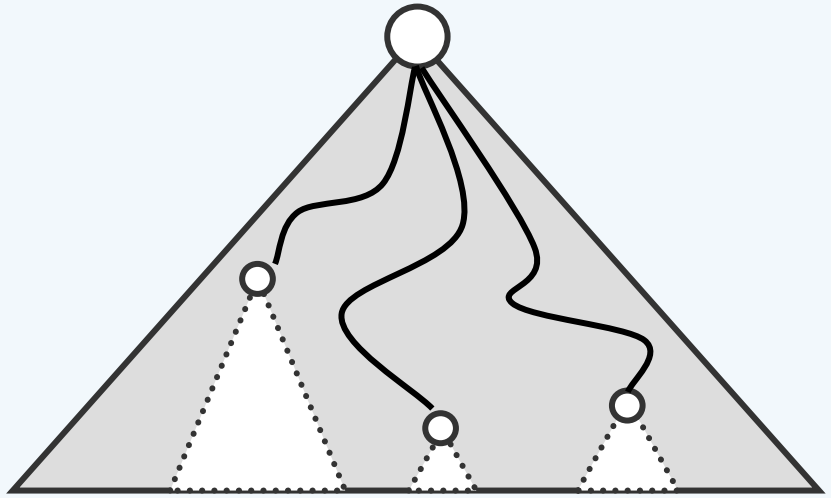
  从0开始，逐渐增加候选解长度 //试探

  一旦发现注定要失败，则

  收缩至前一长度，并 //剪枝回溯

  继续试探
- 特修斯的法宝 = 线绳 + 粉笔

  如何以数据结构的形式兑现？



八皇后

- 在n×n的棋盘上放置n个皇后，使得她们彼此互不攻击

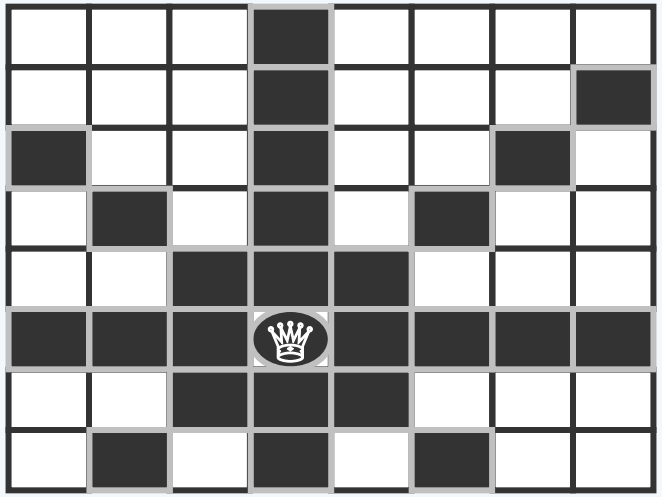
  有多少种可行的布局？如何布局？

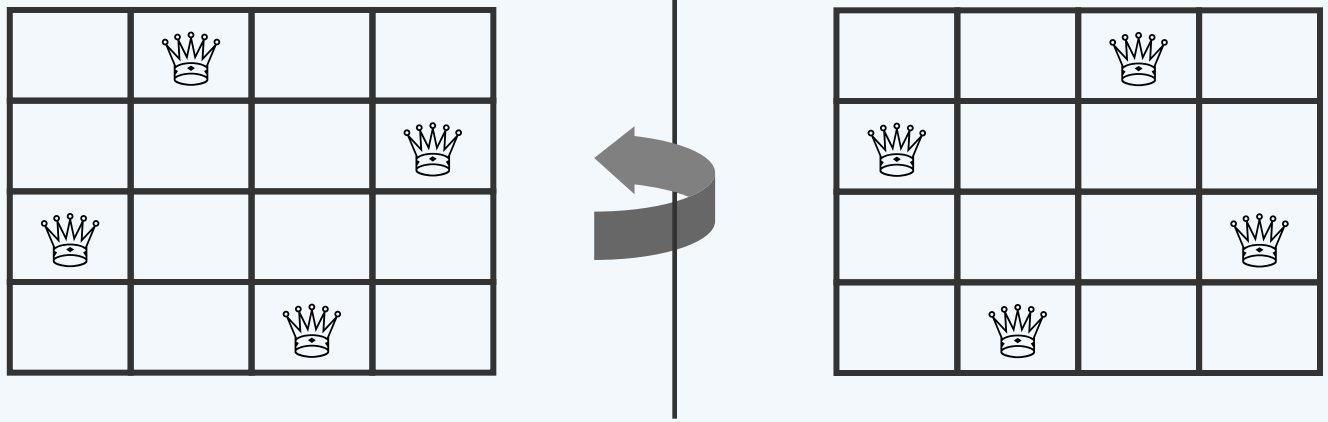
  是否考虑旋转、翻转之后的等价？

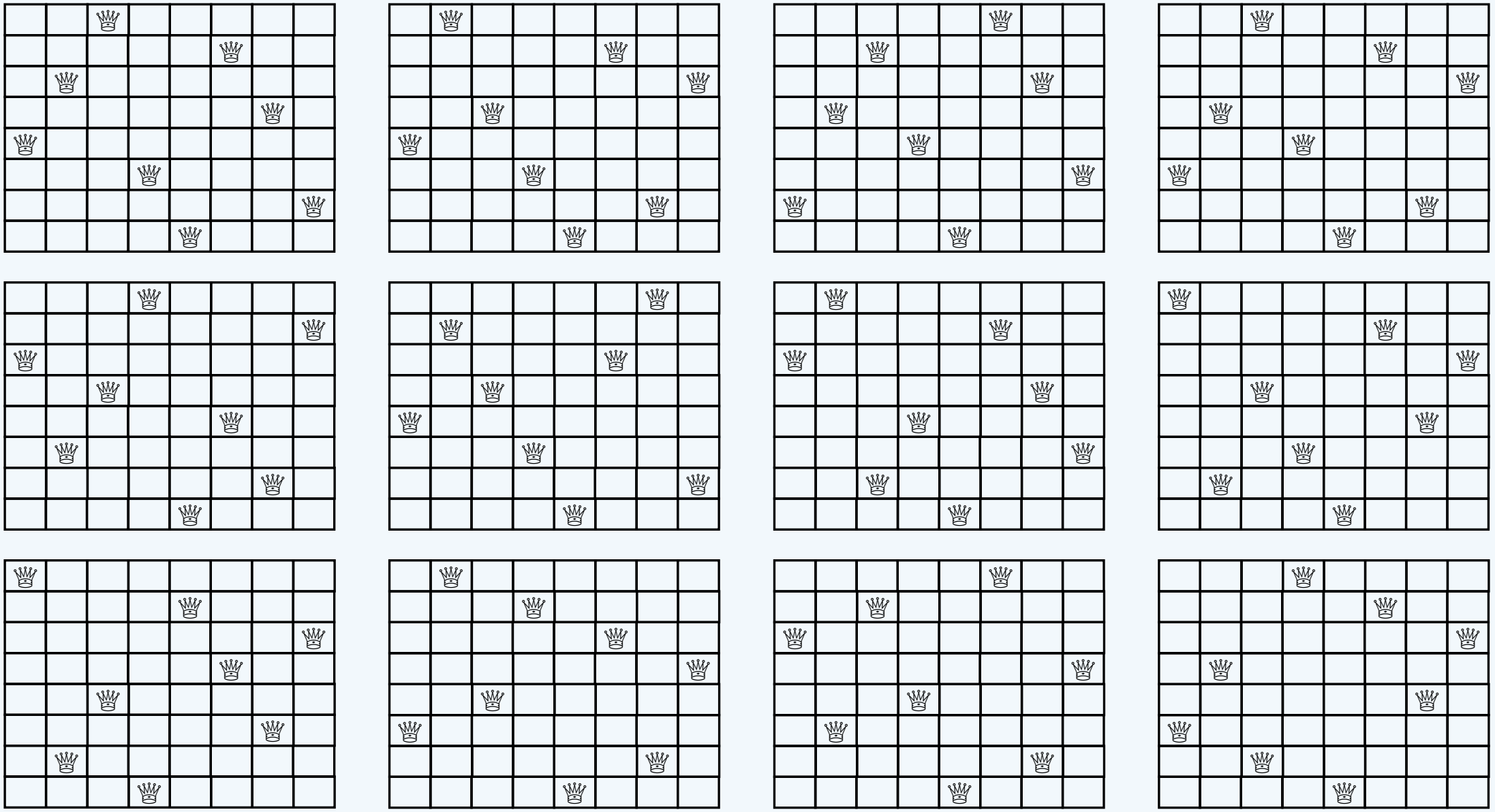
- n = 1, 2, 3, 4, ...

  允许重复： 1, 0, 0, 2, 10, 4, 40, 92, ...

  不许重复： 1, 0, 0, 1, 2, 1, 6, 12, 46, 92, ...

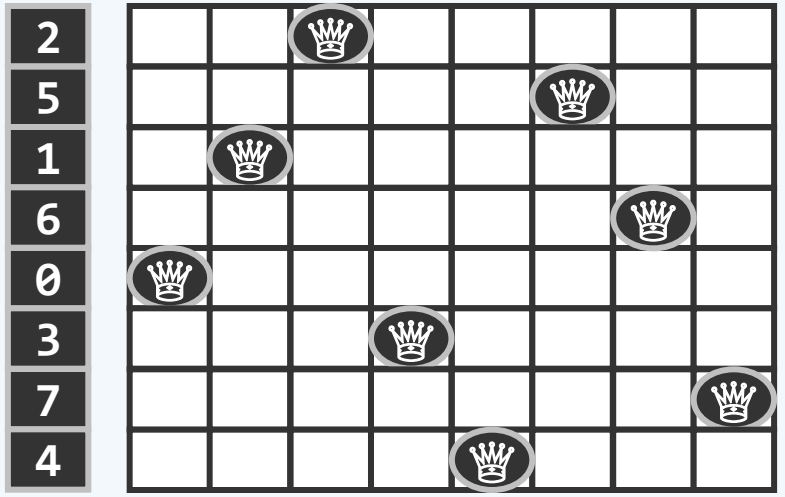


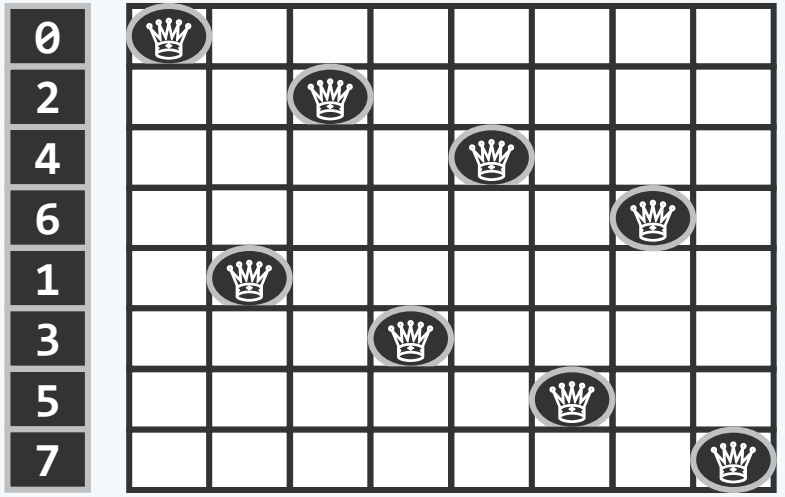




编码

- 观察：每行（列）有且仅有一个皇后
- 因此，每一布局（候选解）都可编码为整数{ 0, ..., n - 1 }的一个排列
- 反之，每一这样的排列，未必是一个可行布局（解）





蛮力搜索

```
❖ void place4Queens_BruteForce() { //4皇后蛮力算法
    int solu[4]; //候选解编码向量
    for (solu[0] = 0; solu[0] < 4; solu[0]++)
    for (solu[1] = 0; solu[1] < 4; solu[1]++)
    for (solu[2] = 0; solu[2] < 4; solu[2]++)
    for (solu[3] = 0; solu[3] < 4; solu[3]++) { //枚举所有候选解
       if (collide(solu, 0)) continue;
       if (collide(solu, 1)) continue;
       if (collide(solu, 2)) continue;
       if (collide(solu, 3)) continue;
       nSolu++; displaySolution(solu, 4);
    }
  } //复杂度高达O(4^4) = O(n^n)
```

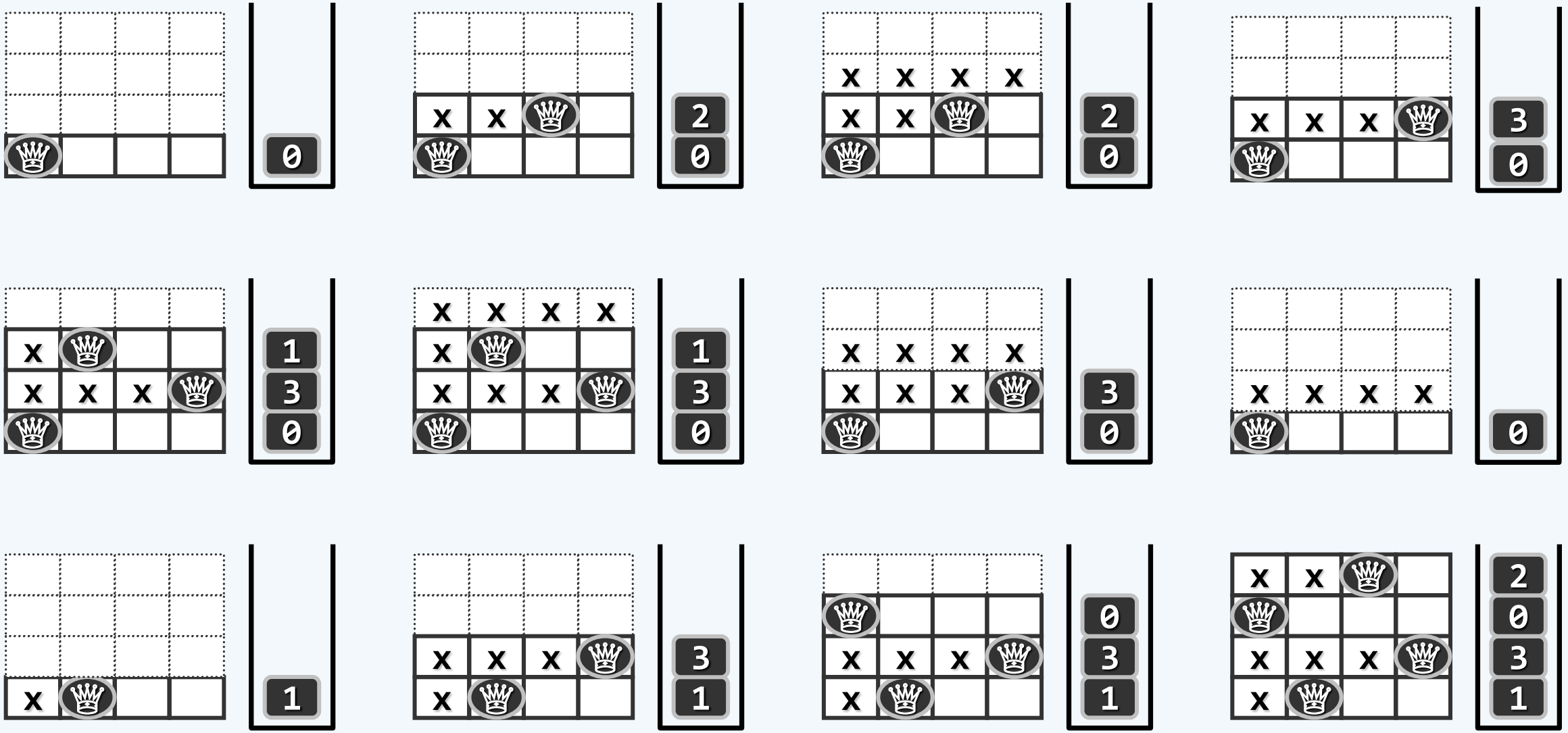
剪枝

```
❖ void place4Queens() { //4皇后剪枝算法
    int solu[4]; //候选解编码向量
    for (solu[0] = 0; solu[0] < 4; solu[0]++)
       if (!collide(solu, 0)) //剪枝
          for (solu[1] = 0; solu[1] < 4; solu[1]++)
             if (!collide(solu, 1)) //剪枝
                for (solu[2] = 0; solu[2] < 4; solu[2]++)
                   if (!collide(solu, 2)) //剪枝
                      for (solu[3] = 0; solu[3] < 4; solu[3]++)
                         if (!collide(solu, 3)) { //剪枝
                            nSolu++; displaySolution(solu, 4);
                         }
  } //复杂度大大降低，但算法的通用性欠佳
```

通用算法

```
❖ void placeQueens(int N) { //N = 棋盘大小 = 皇后总数，问题的规模可任意
    Stack<Queen> solu; Queen q(0, 0); //存放（部分）解的栈，从原点位置出发
    do { //反复试探、回溯
       if (N <= solu.size() || N <= q.y) { //若已出界，则
          q = solu.pop(); q.y++; //回溯一行，并继续试探下一列
       } else { //否则，试探下一行
          while ( (q.y < N) && ( 0 <= solu.find(q) ) ) //通过与已有皇后的比对
             q.y++; //尝试找到可摆放下一皇后的列
          if (N > q.y) { //若存在可摆放的列，则摆上当前皇后
             solu.push(q); if (N <= solu.size()) nSolu++; //若局部解已成全局解，则计数
             q.x++; q.y = 0; //转入下一行，从第0列开始，试探下一皇后
          }
       }
    } while ((0 < q.x) || (q.y < N)); //直至所有分支均已被检查或剪枝
 }
```

实例



- 以上算法中的“solu.find(q)”，如何利用栈（向量）的查找接口？
- 定义皇后类Queen，重新定义判等器，使之在语义上与冲突等价
-

```
struct Queen { //皇后类
   int x, y; Queen( int xx = 0, int yy = 0 ) : x(xx), y(yy) {}; //皇后的坐标
   bool operator==( Queen const & q ) { //重载判等操作符
      return (x == q.x) //行冲突（不会发生，可省略）
          || (y == q.y) //列冲突
          || (x + y == q.x + q.y) //沿正对角线冲突
          || (x - y == q.x - q.y); //沿反对角线冲突
   }
   bool operator!=( Queen const & q ) { return !(*this == q); }
};
```