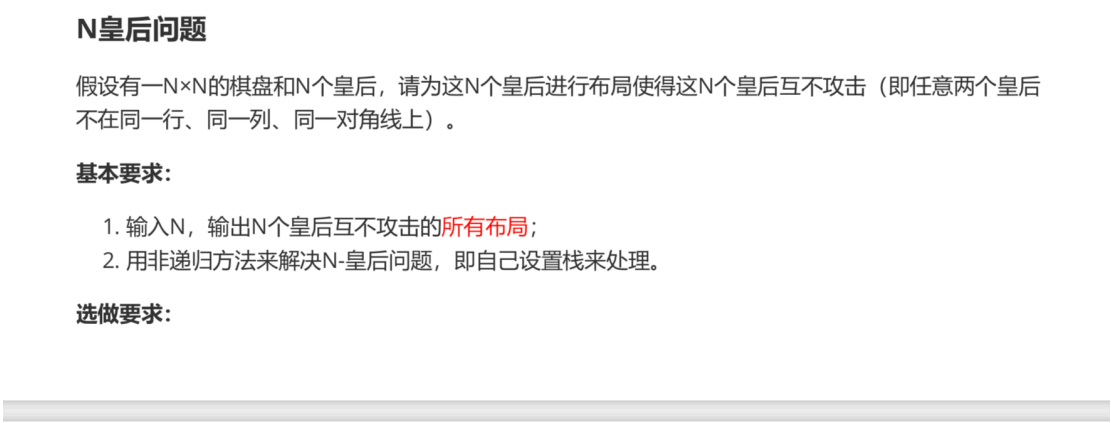
数据结构实验 02

一．实验要求



N皇后问题

假设有一N×N的棋盘和N个皇后，请为这N个皇后进行布局使得这N个皇后互不攻击（即任意两个皇后不在同一行、同一列、同一对角线上）。

基本要求:

1. 输入N，输出N个皇后互不攻击的所有布局;
2. 用非递归方法来解决N-皇后问题，即自己设置栈来处理。

选做要求:

1. 再用递归方法来解决N-皇后问题，并比较递归与非递归程序的运行效率。

基本要求:

1. 输入 N，输出 N 个皇后互不攻击的所有布局;

2. 用非递归方法来解决 N-皇后问题，即自己设置栈来处理。

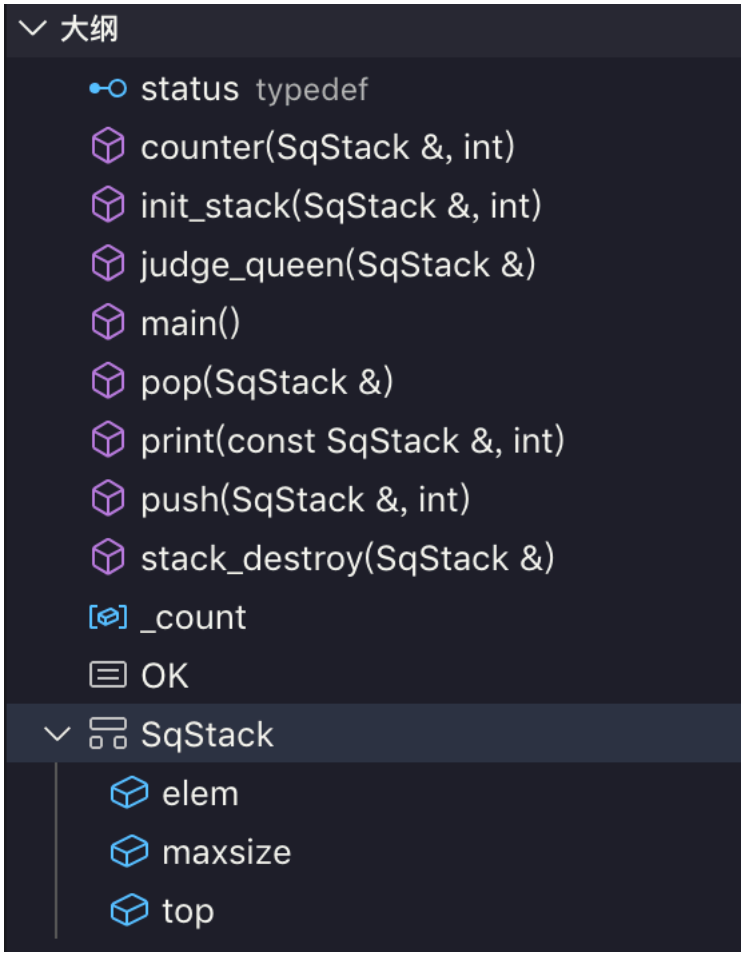
选做要求:

1. 再用递归方法来解决 N-皇后问题，并比较递归与非递归程序的运行效率。

二．设计思路

1. 非递归实现：

整体结构如下：



定义了结构体 SqStack 为顺序栈；

init_stack，pop，push，stack_destory 四个函数进行初始化栈空间，入栈，出栈，销毁栈（释放申请空间）四项基本操作；

print 函数按要求打印出符合要求的布局情况："Q"表示放置皇后的位置，"#"表示空位置；

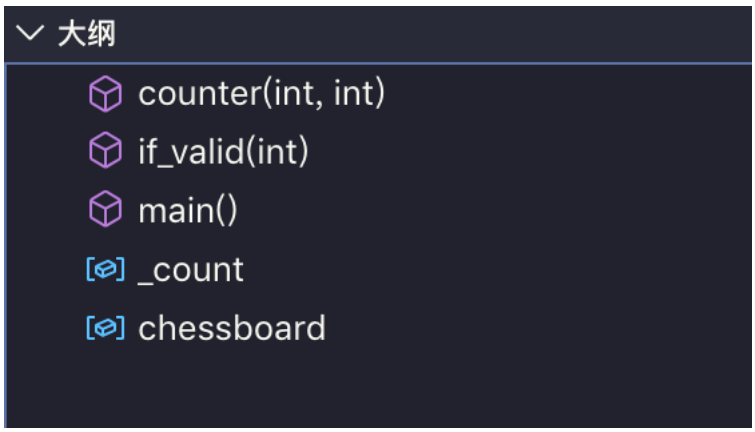
judge_queen 函数用于判断栈顶位置（即目前尝试

的放置皇后的位置）是否合法；

counter 函数是本程序的核心函数，后面会详细介绍。

2. *递归实现：*

程序大致结构如下：



其中 counter 为 本程序的核心函数，后面会细致讲解；

if_valid 函数用于判断新插入的皇后位置是否合法；

全局变量 _count 用于对成功的情况进行计数，一维数组 chessboard 用于记录第 i 行的皇后放在了第 j 列 （chessboar[i] = j）

三．关键代码讲解

1．非递归实现：

下面为 counter 函数的代码，具体的介绍对应在注释中：

```
void counter(SqStack &S,int n){
    int j = 0;
    do
    {
        if(S.top == n-1)
        {   /* n 行已被全部占满，得到一种可行的情况
                记录、打印出该结果之后，最后一行的 j 向下一列移动 */
            _count++;
            print(S,n);
            j = pop(S);
            j++;
            continue;
        }
        if(j == n)
        {//若 j 已经试遍了某行的所有列，则上一行的皇后向下一列移动
            j = pop(S);
            j++;
            continue;
        }
        push(S,j);  //将皇后放在该行第 j 列
        if(judge_queen(S))
        {//插入位置合法，进入下一行寻找
            j = 0;
            continue;
        }
```

```
        else
        {//插入位置非法，j 出栈，向该行的下一列寻找
            pop(S);
            j++;
        }
    }while( j != -1 );//结束条件为第一行所有列已尝试完毕
}//counter
```

由之前的定义，栈空时使用 pop 函数得到的异常返回值为 -2，所以最终的结束条件为 j 等于-1

judge_queen 函数的代码如下：

```
bool judge_queen(SqStack &S)
{
    //检验栈顶元素是否合法
    for(int j = 0;j<S.top;j++){     //这里的 j 控制行数，遍历 i 之前的行
        if( S.elem[S.top]==S.elem[j] ||
            S.elem[S.top]-S.elem[j] == S.top-j ||
            S.elem[S.top]-S.elem[j] == j-S.top )
            return(0);
    }
    return(1);
}
```

if 语句中的三种情况分别对应与之前行的皇后在同一列、左上——右下对角线、左下——右上对角线。

2.递归实现：

下面为 counter 函数的代码：

```
//尝试在第 i 行放皇后
void counter(int i,int n){
    if(i == n){
        //此时 n 行已经全部放满
        _count++;
        //打印该种情况下的结果
        for(int m = 0;m<n;m++)
        {
            for(int k = 0;k<n;k++)
            {
                if(k == chessboard[m])
                    cout<<"Q ";
                else
                    cout<<"# ";
            }
            cout<<endl;
        }
        cout<<endl;
        return;
    }
    for(int j = 0;j<n;j++)  //这个 for 控制每一行的所有列均进行尝试。 所有列均尝试完成
//后，退栈
    {
        chessboard[i] = j;
        if(if_valid(i))     counter(i+1,n);     //若这个插入合法，则开始进行下一行
//的插入(进入下一个函数）。
                                                //否则，仍停留在该行，向下一列寻找
    }
}
```

而 if_valid 函数与非递归实现中的 judge_queen 函数基本一致，这里不再赘述。

四．调试分析

1. 非递归实现：

时间复杂度：

第一行放置有 N 种情况；之后下一行有 N-2 种情况；再下一行最多有 N-3 种情况

……

当前 N-1 行的皇后位置确定后，最后一行的可能位置最多只有 1 种。

故最坏情况下时间复杂度为 O(N!)

空间复杂度：

只申请了一个长度为 N 的栈，故空间复杂度为

O(N)

2. *递归实现*：

时间复杂度:

与非递归时基本一致，为 O(N!)

空间复杂度：

函数递归调用最多 N 次，因此空间复杂度也为 O(N)

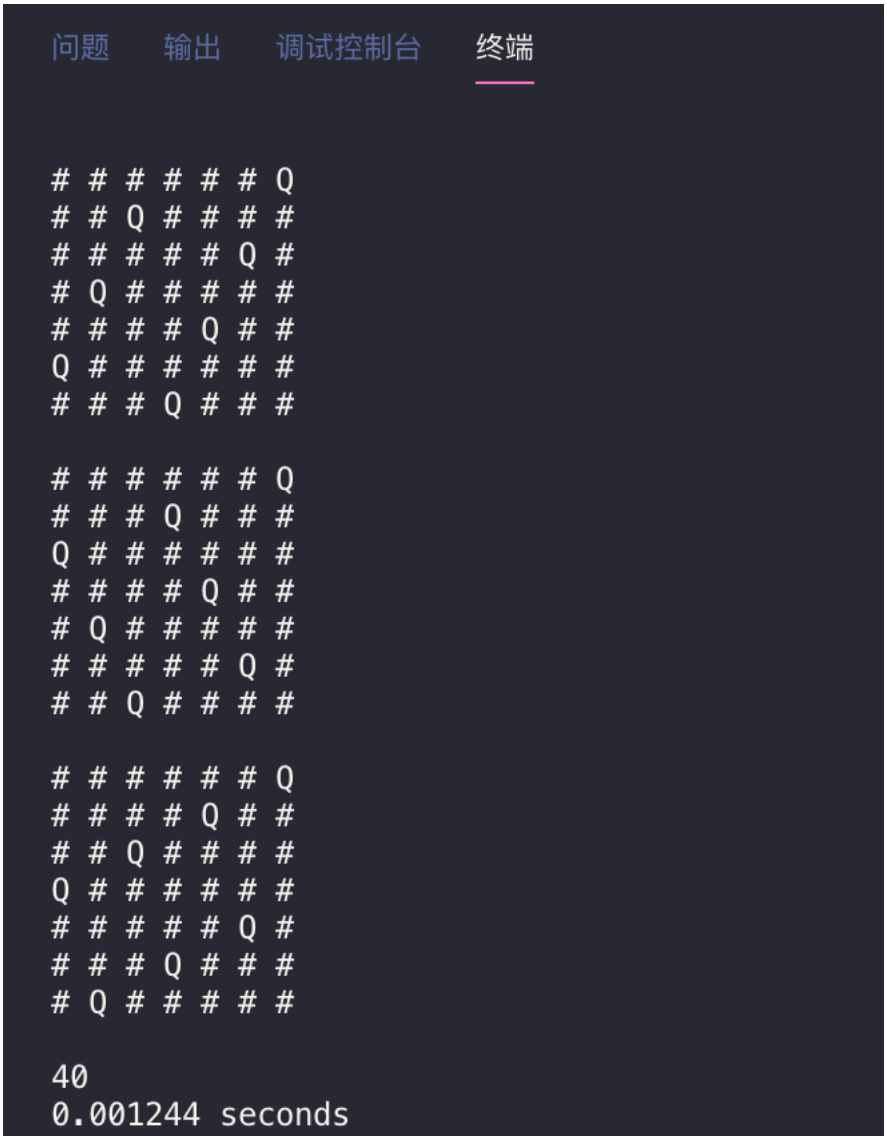
N 皇后问题属于回溯法问题中的经典案例，本次实验过程中没有遇到太大的问题。

五．代码测试

在 visual studio code 中使用 code runner 插件配合 g++编译器进行了测试，结果截图如下：

1.非递归实现：

N = 7 :



N = 8:

N = 10:


问题 输出 调试控制台 终端
Q
Q #
Q # # # #
Q # # # # # # #
Q # # # # #
Q # # # # # # # # #
Q # #
Q
Q # # #
Q # # # # # #
Q
Q #
Q # # # #
Q # # # # # # #
Q
Q # #
Q # # # # #
724
0.032090 seconds

N = 12:


问题 输出 调试控制台 终端
Q #
Q # #
Q # # # # # #
Q # # # # # # # #
Q # # # # # # # # # # #
Q # # # #
Q # # # # # # # # #
Q
Q # # # # #
Q # # # # # # #
Q # #
14200
0.843138 seconds

N = 13


问题 输出 调试控制台 终端
Q # # #
Q
Q # # # # # # # #
Q # # # # # # # # # # #
Q # # # # # # # # #
Q # # # # # # # # # # # # #
Q # #
Q # # # #
Q # # # # # #
Q # # # # # # # # #
Q # # # # #
73712
5.098938 seconds

2.递归实现：

N = 7:


问题 输出 调试控制台 终端
Q # #
Q # # # # # #
Q #
Q # # # #
Q
Q # # #
Q
Q #
Q # # #
Q # # # # # #
Q
Q # #
Q # # # #
40
0.001445 seconds

N = 8 :


问题 输出 调试控制台 终端
Q #
Q # # # # #
Q # #
Q
Q # # #
Q
Q # # #
Q # # # # # # #
Q # # # #
Q #
Q # # # # #
Q
Q # #
92
0.003071 seconds

N = 11:


问题 输出 调试控制台 终端
Q # # #
Q # # # # # # # #
Q
Q #
Q # # #
Q # # # # #
Q # # # # # # #
Q # # # # # # # # # #
Q
Q # #
Q # # # #
Q # # # # # #
Q # # # # # # # #
2680
0.115545 seconds

N = 10:


问题 输出 调试控制台 终端
Q
Q # # #
Q # # # # # #
Q
Q #
Q # # # #
Q # # # # # #
Q # # # # # # # # #
Q # # #
Q # # # # # # #
Q
Q # #
Q # # # # #
724
0.027082 seconds

N = 12:


问题 输出 调试控制台 终端
Q # #
Q
Q #
Q # #
Q # # # # # #
Q # # # # # # # #
Q # # # # # # # # # # #
Q # # # #
Q # # # # # # # # #
Q
Q # # # # #
Q # # # # # # #
Q # #
14200
0.679841 seconds

N = 13:

为了进行比较，在程序中引入了 time.h 库中的 clock 函数，统计了程序运行的时间，如每张截图最下方所示。

比较 N = 7,8,10 三种情况下两种方法使用的时间，

N	非递归（s）	递归（s）
7	0.001244	0.001445
8	0.003157	0.003071
10	0.032090	0.027082

12	0.843138	0.679841
13	5.098938	4.000765

通过比较可以发现，开始时两种方法用时基本一致，随着N 变大，递归法会略快一点，原因可能如下：

(1).非递归法中调用函数更为频繁，且 N 变大后，在堆内存上为 SqStack 分配空间的过程会变长

(2).递归时在系统栈上进行操作更为迅速

六．实验总结

1. 通过本次实验深入理解了栈特性，领会它们各自的应用背景。
2. 熟练掌握栈和队列在不同存储结构、不同的约定中，它们基本操作的实现方法与差异。
3.熟悉了递归法与自己构造栈实现的异同。

七．附录

1. **“八皇后_非递归.cpp”**
2. **“八皇后_递归.cpp”**