数据结构实验报告二

题目：

班级：计科一班　　姓名：范文　　学号：　PB18111679　　完成日期：2019.10.31

一．　实验要求

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
3. 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。

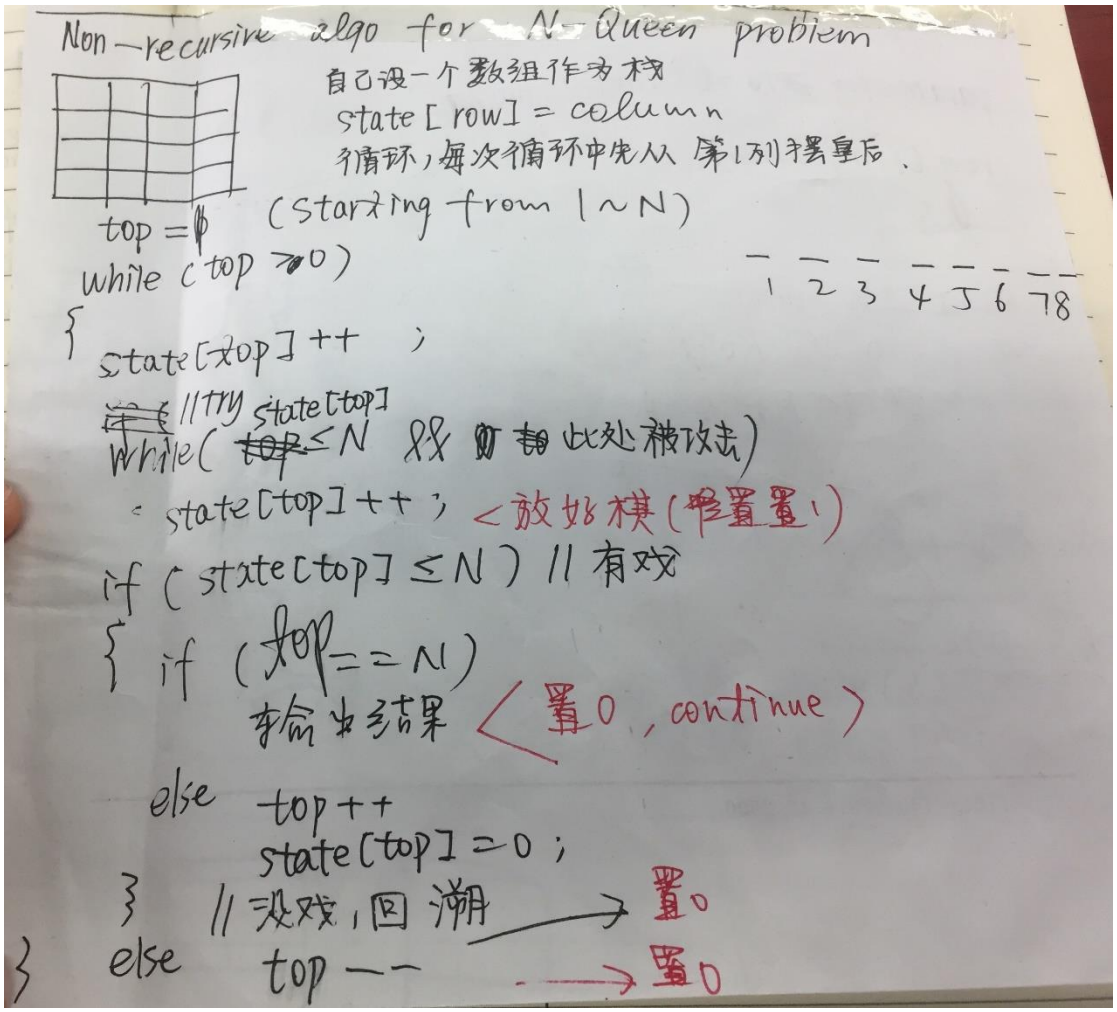
二．　设计思路

首先，自己设计一个 N*N 二维数组 state 来储存棋盘的格局，便于记录攻击情况以及输出棋盘，state[i][j] = 1 代表[i][j]位置上放了皇后，state[i][j] = 0 代表[i][j]位置上未放皇后。再使用一位数组 method（当做栈）来进行迭代的位置记录，method[row] = column 代表第 row 行的皇后置于第 column 列。

```
const int SIZE = 8;
int state[SIZE][SIZE];
int method[SIZE + 1];
```

写上几个辅助的函数，比如为了查看位置（i，j）攻击的情况，就把 i 行，j 列以及对角线上所有在棋盘上的位置都检查一遍。首先找到横竖斜三个方向在棋盘上移动的最大值 temp，然后对横竖斜三个方向上有效的位置（棋盘上）进行查找。

之后，对 method 数组设计一个 top 指针，设其初值为 1. 对其进行迭代直至 top<1，思路分析如下：



若第 top 行可以放置皇后，

当 top = N 时，说明 N 行的皇后都已经放好，因此可以输出这个格局，再把最后一次放置的皇后拿掉，继续在第 N 行下一列尝试。

否则，说明当前可能可行但还不是一组解，就让 top++，从下一行的第一列开始找。

若第 top 行不能放置皇后，即 method[top] > N，就说明在之前 top-1 行的格局无法找到正确解，因此需把第 top 行尝试放的皇后拿走，令 top--，并继续找。

迭代到最后，当后面都不满足要求，top 就会从 1 退回 0，此时查找结束。

三． 核心代码

1．在判断位置(i,j)是否被攻击时，

需要从 4 个数中求最大值的函数 max

```
    int max(int num1,int num2,int num3,int num4)
{
    int temp1 = num1 > num2 ? num1 : num2;
    int temp2 = num3 > num4 ? num3 : num4;
    return temp1 > temp2 ? temp1 : temp2;
}
```

需要判断位置（x，y）是否在棋盘上有效的函数 is_valid:

```
bool is_valid(int i,int j)
{
    if (i >= 0 && i < SIZE && j >= 0 && j < SIZE)
        return true;
    else
        return false;
}
```

因此判断（i，j）是否被攻击的函数就非常好写了。

```
bool is_attacked(int row,int column)
{
    for (int i = 0; i < SIZE;++i)
        if( state[i][column] == 1)
            return true;
```

```
     for (int j = 0; j < SIZE;++j)
        if( state[row][j] == 1)
            return true;


    int temp = max(row, column, SIZE - row, SIZE - column);
    for (int i = 1; i <= temp;++i)
    {
        if( is_valid(row + i,column + i) )
            if(state[row + i][column + i] == 1)
                return true;
         if( is_valid(row + i,column - i) )
            if(state[row + i][column - i] == 1)
                return true;
         if( is_valid(row - i,column + i) )
            if(state[row - i][column + i] == 1)
                return true;
         if( is_valid(row - i,column - i) )
            if(state[row - i][column - i] == 1)
                return true;
    }//for
    return false;
}
```

当找到一组解时，输出格局的函数如下：

```
//output a solution of the chess board if that is valid
//use '#' to represent blank space, 'Q' to represent QUEUE
//it will tell the order of this sulotion.
void print_state(int order)
{
```

```
    cout << "the " << order << "th circumstance: \n";
    for (int i = 0; i < SIZE;++i)
    {
        for (int j = 0; j < SIZE;++j)
        {
            if(state[i][j] == 0)
                cout << "#";
            else
                cout << "Q";
        }
        cout << endl;
    }
}
```

非递归查找函数如下：

```
//finding the solution in a non recursive way
void find_queen()
{
    int order = 1,top = 1;
    method[1] = 0;
    while(top > 0)
    {
        //putting a queen next to the current place.
        ++method[top];
        while(method[top] <= SIZE && is_attacked(top-1,method[top] -1) )
        {
            ++method[top];
        }
        state[top - 1][method[top] - 1] = 1;
        // the place of this queen is valid
```

```
        if(method[top] <= SIZE)
        {
            //the last row is reached
            if(top == SIZE)
            {
                print_state(order);
                ++order;
                state[top - 1][method[top] - 1] = 0;
                continue;
            }
            //go to search the solution of the next row
            else
            {
                ++top;
                method[top] = 0;
            }
        }
        // the place of this queen is not valid, backtrack.
        else
        {
            state[top - 1][method[top] - 1] = 0;
            --top;
            state[top - 1][method[top] - 1] = 0;
        }
    }//while top > 0
}
```

四． 代码测试

main 函数非常简单

```
//using clock() to count the lasting time of execution

int main()

{

    for (int i = 0; i < SIZE;++i)

        for (int j = 0; j < SIZE;++j)

            state[i][j] = 0;

    clock_t startTime = clock();

    find_queen();

    clock_t endTime = clock();

    cout << "the lasting time of non recursive process is "

        << endTime - startTime << " milliseconds"<< endl;

    return 0;

}
```

运行八皇后问题得到了 92 组解，符合要求。三次用时分别为 398，404，538ms，平均用时 446ms。

```
#####Q##
the 91th circumstance:
#######Q
##Q#####
Q#######
#####Q##
#Q######
####Q###
######Q#
###Q####
the 92th circumstance:
#######Q
###Q####
Q#######
##Q#####
#####Q##
#Q######
######Q#
####Q###
the lasting time of non recursive process is 398 milliseconds
```

```
####Q###
the lasting time of non recursive process is 404 milliseconds
```

```
the lasting time of non recursive process is 538 milliseconds
```

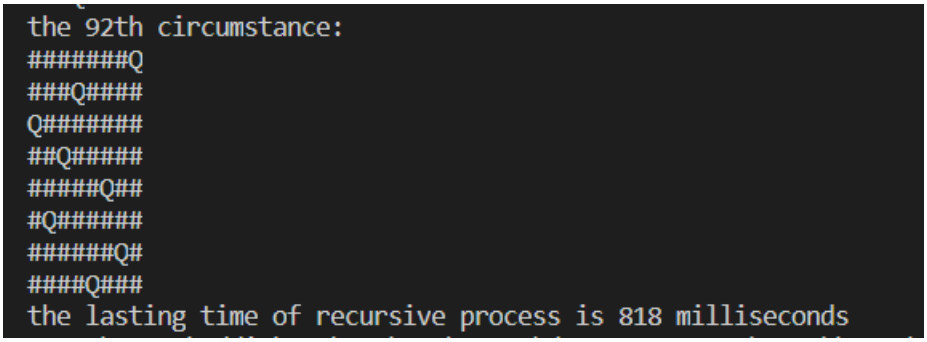
五． 拓展对比

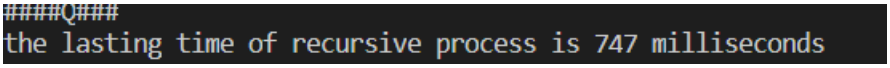
用递归版本的八皇后，在判断攻击，放置棋子，输出状态等辅助函数都基本一样的情况下，对比运行时间。

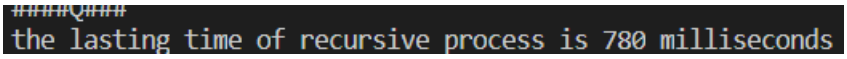
首先给出递归版八皇后的核心代码：

```
//dfs
void queen(int row,int &order)
{
    int column = 0;
    for (; column < SIZE;++column)
    {
        if( !is_attacked(row,column) )
        {
            state[row][column] = 1;
            if(row == SIZE -1)
            {
                print_state(order);
                ++order;
            }
            else
                queen(row + 1,order);
            state[row][column] = 0;
        }
    }
}
```

运行递归版本的八皇后代码，得到了 92 组解，用时分别为 818,747,780ms，平均用时 782ms。







显然可以看出，非递归的解法要快于递归的解法。这是因为函数调用的时候，每次调用时要做地址保存，参数传递等，这是通过一个递归工作栈实现的。具体是每次调用函数本身要保存的内容包括：局部变量、形参、调用函数地址、返回值。那么，如果递归调用 N 次，就要分配 N *局部变量*、*N*形参、N *调用函数地址*、*N*返回值，这势必是影响效率的，同时，这也是内存溢出的原因，因为积累了大量的中间变量无法释放。

六． 实验总结

本次实验，我在先想好递归算法的基础上想出了非递归算法，并对其进行比较，发现其实我们设计非递归算法就是在模拟编译器将递归转化成迭代的过程。同时，我们需要自己设计合适的数据结构（比如栈或者数组）来方便地存储中间变量。另外，经过对比，发现解决同样的问题，非递归算法明显快于递归算法。这体现了递归虽然比较直观，但由于函数的调用开销和重复计算而相对耗时的特点。因此我对栈和递归有了更加深刻的了解。

七． 源代码附件

recur_sol.cpp 存放 N 皇后问题的递归解法

non_recur_sol.cpp 存放 N 皇后问题的非递归解法