

作业要求

4			9					3
	8				1		9	
				2		7		
	3							4
		6	7			5		
2							6	
		7		3		6		
	5		6					
1					9			2

必答题

如上图所示的数独，如果去掉其中任意两个数字，解的个数将会变多，请编程求取：去掉哪两个数字（位置），解的个数最多，为多少个？

开放题

9*9 的数独一共有多少种解？

作业要求：

- 1、提交源代码文件和 word 说明文档。
- 2、源代码文件为可以编译运行的 c 语言代码，需包含代码注释。
- 3、word 说明文档需要包含解题思路描述、求解的答案以及程序运行结果的截图。

作业内容

必答题：

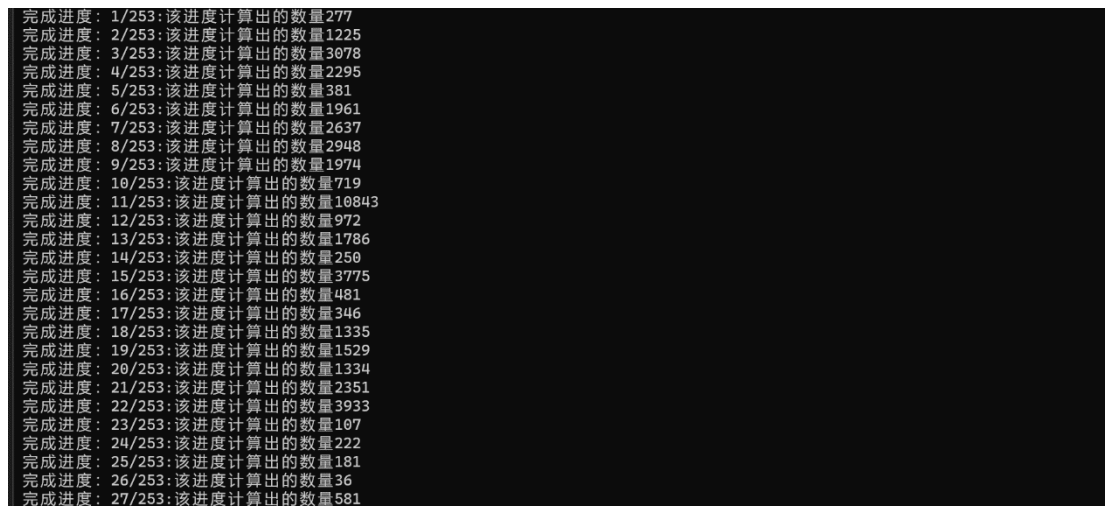
首先，需要事先说明的是，此程序已经被整体打包压缩，放在了附件里，名字叫做 TheHomeworkMustBeFinished，老师可以直接解压运行这个程序查看结果，然后对于开放题，会有另外一个压缩包，参见后面阐述。

随后，我将以截图顺序简要阐述这个程序的效果实例：



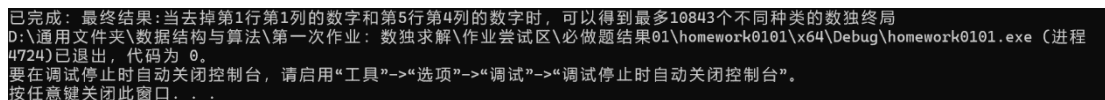
```
D:\通用文件夹\数据结构与算法 > 400900003
080001090
000020700
030000004
006700500
200000060
007030600
050600000
100009002
```

首先，逐行地输入数独，然后换行，就会开始自动计算



```
完成进度：1/253:该进度计算出的数量277
完成进度：2/253:该进度计算出的数量1225
完成进度：3/253:该进度计算出的数量3078
完成进度：4/253:该进度计算出的数量2295
完成进度：5/253:该进度计算出的数量381
完成进度：6/253:该进度计算出的数量1961
完成进度：7/253:该进度计算出的数量2637
完成进度：8/253:该进度计算出的数量2948
完成进度：9/253:该进度计算出的数量1974
完成进度：10/253:该进度计算出的数量719
完成进度：11/253:该进度计算出的数量10843
完成进度：12/253:该进度计算出的数量972
完成进度：13/253:该进度计算出的数量1786
完成进度：14/253:该进度计算出的数量250
完成进度：15/253:该进度计算出的数量3775
完成进度：16/253:该进度计算出的数量481
完成进度：17/253:该进度计算出的数量346
完成进度：18/253:该进度计算出的数量1335
完成进度：19/253:该进度计算出的数量1529
完成进度：20/253:该进度计算出的数量1334
完成进度：21/253:该进度计算出的数量2351
完成进度：22/253:该进度计算出的数量3933
完成进度：23/253:该进度计算出的数量107
完成进度：24/253:该进度计算出的数量222
完成进度：25/253:该进度计算出的数量181
完成进度：26/253:该进度计算出的数量36
完成进度：27/253:该进度计算出的数量581
```

然后就会开始计算，显示出计算过程（由于所有的完成进度一共有 253 个，所以没有全部放出来，只是放出了前面的一部分显示）



```
已完成：最终结果:当去掉第1行第1列的数字和第5行第4列的数字时，可以得到最多10843个不同种类的数独终局
D:\通用文件夹\数据结构与算法\第一次作业：数独求解\作业尝试区\必做题结果01\homework0101\x64\Debug\homework0101.exe (进程
4724)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .
```

最终，实现了对于这个数独的操作：找出了去除其中哪两个数字可以使得整个数独的终局数量最多。

为了验证这个程序的可靠性，我们尝试把原来的数独的最后一个数字 2 改成 0，看一下这个程序能不能正确运行。（注：这个的输入数独改了，不是题目中的数独，是用来验证的）

```
DA\通用文件夹\数据结构与算法 > + v
400900003
080001090
000020700
030000004
006700500
200000060
007030600
050600000
100009000
```

首先，我们把修改之后的数独输入进去（把数独的最后一个数字 2 改成 0），然后回车

```
完成进度：1/231:该进度计算出的数量7943
完成进度：2/231:该进度计算出的数量28969
完成进度：3/231:该进度计算出的数量29333
完成进度：4/231:该进度计算出的数量38184
完成进度：5/231:该进度计算出的数量12494
完成进度：6/231:该进度计算出的数量24184
完成进度：7/231:该进度计算出的数量26743
完成进度：8/231:该进度计算出的数量16352
完成进度：9/231:该进度计算出的数量32867
完成进度：10/231:该进度计算出的数量32085
完成进度：11/231:该进度计算出的数量38740
完成进度：12/231:该进度计算出的数量29402
完成进度：13/231:该进度计算出的数量31103
完成进度：14/231:该进度计算出的数量8367
完成进度：15/231:该进度计算出的数量32209
完成进度：16/231:该进度计算出的数量17203
完成进度：17/231:该进度计算出的数量15697
完成进度：18/231:该进度计算出的数量14526
完成进度：19/231:该进度计算出的数量18038
完成进度：20/231:该进度计算出的数量45001
完成进度：21/231:该进度计算出的数量25231
```

可以发现，这个程序仍然正确运行，而且总进度也根据数独的变化做出调整

```
Microsoft Visual Studio 调试 > + v
完成进度：207/231:该进度计算出的数量3252
完成进度：208/231:该进度计算出的数量2798
完成进度：209/231:该进度计算出的数量7624
完成进度：210/231:该进度计算出的数量3982
完成进度：211/231:该进度计算出的数量16829
完成进度：212/231:该进度计算出的数量17534
完成进度：213/231:该进度计算出的数量14085
完成进度：214/231:该进度计算出的数量18229
完成进度：215/231:该进度计算出的数量51088
完成进度：216/231:该进度计算出的数量20320
完成进度：217/231:该进度计算出的数量6648
完成进度：218/231:该进度计算出的数量4038
完成进度：219/231:该进度计算出的数量5532
完成进度：220/231:该进度计算出的数量12706
完成进度：221/231:该进度计算出的数量8841
完成进度：222/231:该进度计算出的数量5302
完成进度：223/231:该进度计算出的数量7813
完成进度：224/231:该进度计算出的数量11283
完成进度：225/231:该进度计算出的数量7021
完成进度：226/231:该进度计算出的数量6223
完成进度：227/231:该进度计算出的数量20183
完成进度：228/231:该进度计算出的数量5435
完成进度：229/231:该进度计算出的数量12726
完成进度：230/231:该进度计算出的数量8119
完成进度：231/231:该进度计算出的数量20540
已完成：最终结果:当去掉第7行第3列的数字和第9行第1列的数字时，可以得到最多51088个不同种类的数独终局
D:\通用文件夹\数据结构与算法\第一次作业：数独求解\作业尝试区\必做题结果01\homework0101\x64\Debug\homework0101.exe (进程
15700)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。...
```

程序给出了结果，证明这个程序是可靠的，在改变了原有数独的情况下仍然给出了结果，也证明了这个程序不仅可以解决作业总那个数独，还可以解决任何一个数独的问题。

下面，我再简要介绍一下这个程序的特点和整体设计框架：

程序特点：

- 1. 之前老师提供的程序没有输入数独的功能，只能直接求解代码中已经写好的数独。这个程序则可以允许自由输入任意标准数独进行求解，而且使用了用字符串转换数字的方法进行读入，可以不用麻烦我们在一行输入长长的数独，而是一行换一行的输入，可以让我们更方便地进行输入。
- 2. 中间加入了完成进度显示的新功能，可以让我们清楚地看到整个程序运行的情况。而且这

个程序会自动根据输入数独计算总进度的数量、已完成进度的数量和该进度计算出的数量，大大提高了程序运行的观感，避免我们长时间等待程序运行而无法得知程序运行情况，而且也方便调试进一步升级功能。

3. 整个程序并没有使用之前课上给的线性表的接口方法，主要是因为感觉在实现这个功能的时候觉得这个程序功能比较简单，而且线性表不只一个，写线性表可能会造成不必要的冗余，所以直接在 main 函数里面使用了两个结构体直接实现了。

整体设计框架：

算法：

整个程序其实是基于老师所提供的回溯法求解数独的代码上进行修改完善实现的，为了减少冗余并增加可读性，没有使用额外的函数来做接口，在开头位置新增了一个读取数独的功能，这里使用了读取字符的方法来存储数独，因为如果是读取数字的话由于系统缓冲区的设置，没办法实现逐行输入的功能。读取字符后再用 ASCII 码的特点转换为数字并存储到对应数组中。由于需要遍历任意去除两个数字的情况，所以随后需要进行一个数字搭配的循环，我创建了两个一模一样的循环，第一个循环的功能是用来求出总的数字搭配的数量，以求出“总进度”，然后第二个循环的功能是用来正式地在每一次数字搭配中求出数独数量并且记录对应求解信息。再求出所有数独求解信息后，我使用了之前 C++ 教过的“打擂台”的方法求出所有数独求解信息中对应解的数量最大那一个，然后将其其它信息也一起打印出来。

数据结构：

整个程序中的数独分为两类，一类是以字符形式储存的二维数组，一类是以数字形式储存的二维数组。然后有两个记录数字信息的数据类型，一类是储存已知数字相关信息的结构体，其中有存储行列信息以及数字信息，并进一步定义了一个对应数组，一类是储存去掉数字信息和其对应解数量的结构体，其中有存储第一个和第二个交换数字的行列信息以及数独终局数量，并进一步定义了一个结构体对象和一个数组，分别代表最终结果和每种情况的集合。

最后，把代码放上来，其中基本每一个重要位置都做了注释：

```
#include <iostream>
using namespace std;

int numm = 0; //统计一个数独解的数量（变量）
int total = 0; //统计不同已知数字搭配的总数量，同时也是进度总数量（变量）

struct mask_1
{
    int rowdata;
    int valdata;
    int numdata;
}mask_1s[81]; //一个储存已知数字相关信息的数组

struct mask_2
{
    int firstonerow;
```

```

    int firstoneval;
    int secondonerow;
    int secondoneval;
    int number;
}mask_2s[1000], result;
//一个储存去掉数字信息和其对应解数量的数组
//一个储存去掉数字信息和其对应解数量的结果的结构体

void GetCanFillNumber(int sudu[9][9], int row, int col, int mask[])
{
    for (int i = 0; i < 9; ++i)
    {
        int val = sudu[row][i];
        if (val == 0)
            continue;

        mask[val - 1] = 1;
    }
    for (int i = 0; i < 9; ++i)
    {
        int val = sudu[i][col];
        if (val == 0)
            continue;

        mask[val - 1] = 1;
    }

    int leftCol = col / 3 * 3;
    int topRow = row / 3 * 3;;
    for (int i = topRow; i < topRow + 3; ++i)
        for (int j = leftCol; j < leftCol + 3; ++j)
        {
            int val = sudu[i][j];
            if (val == 0)
                continue;

            mask[val - 1] = 1;
        }
}
//老师给的得到数独中可填数字的函数

bool IsFinished(int sudu[9][9])
{
    for (int i = 0; i < 9; ++i)

```

```

        for (int j = 0; j < 9; ++j)
        {
            if (sudu[i][j] == 0)
            {
                return false;
            }
        }

        return true;
    }
}

```

//老师给的判断数独是否完成的函数

```

void PrintfSudu(int sudu[9][9])
{
    for (int i = 0; i < 9; ++i)
    {
        for (int j = 0; j < 9; ++j)
        {
            printf("%d,", sudu[i][j]);
        }
        printf("\r\n");
    }
    printf("\r\n");
}

```

//老师给的打印数独的函数，但是在这道题实际上不会调用这个函数

```

void FillSudu(int sudu[9][9])
{
    for (int i = 0; i < 9; ++i)
        for (int j = 0; j < 9; ++j)
        {
            if (sudu[i][j] == 0)
            {
                int mask[9] = { 0 };
                GetCanFillNumber(sudu, i, j, mask);

                for (int k = 0; k < 9; ++k)
                {
                    if (mask[k] == 0)
                    {
                        sudu[i][j] = k + 1;
                        FillSudu(sudu);
                        sudu[i][j] = 0;
                    }
                }
            }
        }
}

```

```

        }
    }
    return;
}
}

```

```

if (IsFinished(sudu))
{
    ++numm;
    return;
}
}

```

//老师给的完成数独的函数，做了一些修改，将打印数独结果修改成数独结果统计。

```

int main()
{
    int sudu[9][9]; //定义了一个存储数独的二维数组
    char suduchar[9][9]; //定义了一个用于存储数独字符形式的二维数组
    int row = 0, val = 0, t = 0; //初始化了行、列和统计变量的值
    int sumtotal = 0; //初始化了统计变量的值，下面会用到
    for (row = 0; row <= 8; ++row)
    {
        for (val = 0; val <= 8; ++val)
        {
            cin >> suduchar[row][val];
            sudu[row][val] = suduchar[row][val] - '0';
        }
    } //这里利用了输入字符转换为数值的方法，确保自由输入任意数独，而且支持换行
    for (row = 0; row <= 8; ++row)
    {
        for (val = 0; val <= 8; ++val)
        {
            if (sudu[row][val] == 0)
            {
                continue;
            }
            else
            {
                mask_ls[sumtotal].numdata = sudu[row][val];
                mask_ls[sumtotal].rowdata = row;
                mask_ls[sumtotal].valdata = val;
                ++sumtotal;
            }
        }
    }
}

```

```

    }
}
} //这里对于初始数独进行了扫描，将其中已知数字信息存储mask_1s中
int i0 = 0, j0 = 0;
for (i0 = 0; i0 <= sumtotal - 1; ++i0)
{
    for (j0 = i0 + 1; j0 <= sumtotal - 1; ++j0)
    {
        ++total;
    }
} //这里事先统计了已知数字搭配的总数量，得到了总进度的大概数字。
for (i0 = 0; i0 <= sumtotal - 1; ++i0)
{
    for (j0 = i0 + 1; j0 <= sumtotal - 1; ++j0)
    {
        sudu[mask_1s[i0].rowdata][mask_1s[i0].valdata] = 0;
        sudu[mask_1s[j0].rowdata][mask_1s[j0].valdata] = 0;
        FillSudu(sudu);
        //求解数独
        mask_2s[t].number = numm;
        mask_2s[t].firstonerow = mask_1s[i0].rowdata;
        mask_2s[t].firstoneval = mask_1s[i0].valdata;
        mask_2s[t].secondonerow = mask_1s[j0].rowdata;
        mask_2s[t].secondoneval = mask_1s[j0].valdata;
        printf("完成进度: %d/%d:该进度计算出的数量%d\n", t + 1, total,
numm);

        //打印已完成进度和总进度的关系，并且打印每次进度计算出的数量
        numm = 0;
        ++t;
        sudu[mask_1s[i0].rowdata][mask_1s[i0].valdata] =
mask_1s[i0].numdata;
        sudu[mask_1s[j0].rowdata][mask_1s[j0].valdata] =
mask_1s[j0].numdata;
    }
}

} //这里计算每种数字搭配的解的信息并存储到mask_2s中
result.number = 0;
for (int w = 0; w <= t - 1; ++w)
{
    if (mask_2s[w].number > result.number)
    {
        result.number = mask_2s[w].number;
        result.firstonerow = mask_2s[w].firstonerow;
        result.firstoneval = mask_2s[w].firstoneval;
    }
}

```



```

        result.seconddonerow = mask_2s[w].seconddonerow;
        result.seconddoneval = mask_2s[w].seconddoneval;
    }

    }//比较所有解中最大的那一个数字，并且将相关信息存储result中
    printf("已完成：最终结果:当去掉第%d行第%d列的数字和第%d行第%d列的数字时，可
    以得到最多%d个不同种类的数独终局", result.firstonerow + 1, result.firstoneval +
    1, result.seconddonerow + 1, result.seconddoneval + 1, result.number);
    //打印结果
}

```

开放题：

对于此道开放题，我尝试把已知数独直接改为全部为0的数独，但是发现程序长时间加载不出来，所以我怀疑是9x9数独已知终局数量太多，导致计算机也无法计算出来，所以我去网上搜索了相关资料。

经过网上资源的搜索，找到两个可靠的信源：

第一个，来自于[数独 - 搜狗百科 \(sogou.com\)](http://sogou.com)，中间有提到：

数独中的数字排列千变万化，那么究竟有多少种终盘的数字组合呢？
 6, 670, 903, 752, 021, 072, 936, 960（约为 6.67×10 的 21 次方）种组合，2005 年由 Bertram Felgenhauer 和 Frazer Jarvis 计算出该数字，并将计算方法发布在他们网站上，如果将等价终盘（如旋转、翻转、行行对换，数字对换等变形）不计算，则有 5, 472, 730, 538 个组合。数独终盘的组合数量都如此惊人，那么数独题目数量就更加不计其数了，因为每个数独终盘又可以制作出无数道合格的数独题目。

可以发现，这个数字基本上是一个天文数字了，经过我的计算，目前所有的个人计算机都无法在运用简单回溯算法的情况下，在可供接受的时间内完成所有数独终盘组合的计算。

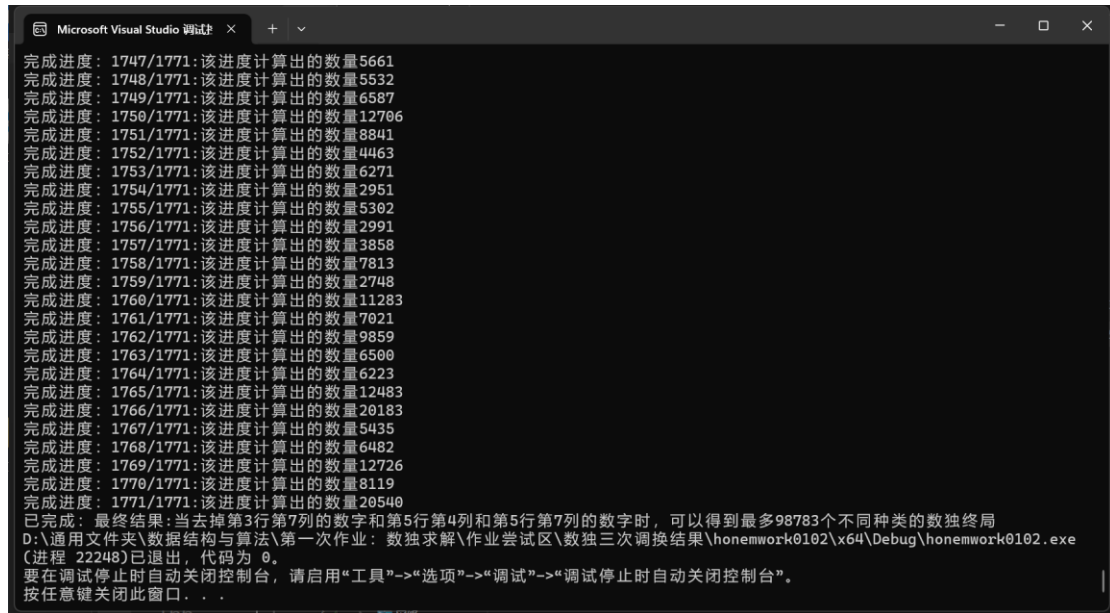
第二个，来自于[总共有多少个数独（转载） - busyfruit - 博客园 \(cnblogs.com\)](http://cnblogs.com)，提供了这个数独组合严谨的数学计算过程，非常遗憾的是，笔者数学知识有限，并不能看懂繁琐的计算过程，但是最终结果显示与第一个信源是一模一样的数字。

那么，我们的程序设计就到此为止了吗？当然不是，因为仔细研究会发现，想要解决这个问题，有两种办法，第一种办法，不能使用原来的简单回溯法进行计算，而是使用形式繁琐但是效率极高的数独计算方法进行计算，可能就可以在可接受的时间完成所有计算了；第二种办法，我们不妨从第一个问题，即必答题那里入手，我们已经得到了去掉两个数字之后数独的数量，以及最多的情况。那么，我们为什么不能进一步去掉三个、四个……向着我们的去掉所有已知结果不断逼近？

那么，由于后者的办法可行度更高，所以我们从后者开始进行探索和研究。我们需要对原有程序进行更改，使得原有程序可以处理去掉三个已知数字的结果。修改完成之后，会发现一个问题，在程序运行至最后几位的时候，会出现内存溢出的情况。经过我几天的摸索后发现，是因为我所定义的整型变量已经无法存储得出的大数字了，所以，我将变量

全部修改为 long long int 类型，以避免内存溢出。具体能够成功运行的代码。[我放在](#)

了名字叫做selectablework的压缩包里面的homework0102文件夹了，老师可以直接打开输入已知数独并且运行，操作方法和必选题中的操作没有差别。



```
Microsoft Visual Studio 调试  x  +  v
完成进度: 1747/1771:该进度计算出的数量5661
完成进度: 1748/1771:该进度计算出的数量5532
完成进度: 1749/1771:该进度计算出的数量6587
完成进度: 1750/1771:该进度计算出的数量12706
完成进度: 1751/1771:该进度计算出的数量8841
完成进度: 1752/1771:该进度计算出的数量4463
完成进度: 1753/1771:该进度计算出的数量6271
完成进度: 1754/1771:该进度计算出的数量2951
完成进度: 1755/1771:该进度计算出的数量5302
完成进度: 1756/1771:该进度计算出的数量2991
完成进度: 1757/1771:该进度计算出的数量3858
完成进度: 1758/1771:该进度计算出的数量7813
完成进度: 1759/1771:该进度计算出的数量2748
完成进度: 1760/1771:该进度计算出的数量11283
完成进度: 1761/1771:该进度计算出的数量7021
完成进度: 1762/1771:该进度计算出的数量9859
完成进度: 1763/1771:该进度计算出的数量6500
完成进度: 1764/1771:该进度计算出的数量6223
完成进度: 1765/1771:该进度计算出的数量12483
完成进度: 1766/1771:该进度计算出的数量20183
完成进度: 1767/1771:该进度计算出的数量5435
完成进度: 1768/1771:该进度计算出的数量6482
完成进度: 1769/1771:该进度计算出的数量12726
完成进度: 1770/1771:该进度计算出的数量8119
完成进度: 1771/1771:该进度计算出的数量20540
已完成: 最终结果: 当去掉第3行第7列的数字和第5行第4列和第5行第7列的数字时, 可以得到最多98783个不同种类的数独终局
D:\通用文件夹\数据结构与算法\第一次作业: 数独求解\作业尝试区\数独三次调换结果\honework0102\x64\Debug\honework0102.exe
(进程 22248)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

这就是最后运行的结果了，总进度是1771个数字搭配，然后整个程序大概花费了一个多小时的时间运行出结果。我也把代码直接拷在下面，由于代码与之前的功能几乎没有什么变化，只是在搭配数量上加了1，所以我就没有对于注释做出大的调整，所有与必答题中不一样的地方都用红色字体进行标注。

```
#include <iostream>
using namespace std;

int numm = 0; //统计一个数独解的数量（变量）
int total = 0; //统计不同已知数字搭配的总数量，同时也是进度总数量（变量）

struct mask_1
{
    int rowdata;
    int valdata;
    int numdata;
}mask_1s[81]; //一个储存已知数字相关信息的数组

struct mask_2
{
    int firstonerow;
    int firstoneval;
```

```

    int secondonerow;
    int secondoneval;
    int thirdonerow;
    int thirdoneval; //这里增加了第三个数字的行列信息
    long long int number; //就是这里需要改为long long int, 要不然会导致内存溢出
}mask_2s[10000], result;
//一个储存去掉数字信息和其对应解数量的数组
//一个储存去掉数字信息和其对应解数量的结果的结构体

```

```

void GetCanFillNumber(int sudu[9][9], int row, int col, int mask[])
{
    for (int i = 0; i < 9; ++i)
    {
        int val = sudu[row][i];
        if (val == 0)
            continue;

        mask[val - 1] = 1;
    }
    for (int i = 0; i < 9; ++i)
    {
        int val = sudu[i][col];
        if (val == 0)
            continue;

        mask[val - 1] = 1;
    }

    int leftCol = col / 3 * 3;
    int topRow = row / 3 * 3;;
    for (int i = topRow; i < topRow + 3; ++i)
        for (int j = leftCol; j < leftCol + 3; ++j)
        {
            int val = sudu[i][j];
            if (val == 0)
                continue;

            mask[val - 1] = 1;
        }
}

```

//老师给的得到数独中可填数字的函数

```

bool IsFinished(int sudu[9][9])
{

```

```

        for (int i = 0; i < 9; ++i)
            for (int j = 0; j < 9; ++j)
            {
                if (sudu[i][j] == 0)
                {
                    return false;
                }
            }

        return true;
    }
}

```

//老师给的判断数独是否完成的函数

```

void PrintfSudu(int sudu[9][9])
{
    for (int i = 0; i < 9; ++i)
    {
        for (int j = 0; j < 9; ++j)
        {
            printf("%d, ", sudu[i][j]);
        }
        printf("\r\n");
    }
    printf("\r\n");
}

```

//老师给的打印数独的函数，但是在这道题实际上不会调用这个函数

```

void FillSudu(int sudu[9][9])
{
    for (int i = 0; i < 9; ++i)
        for (int j = 0; j < 9; ++j)
        {
            if (sudu[i][j] == 0)
            {
                int mask[9] = { 0 };
                GetCanFillNumber(sudu, i, j, mask);

                for (int k = 0; k < 9; ++k)
                {
                    if (mask[k] == 0)
                    {
                        sudu[i][j] = k + 1;
                        FillSudu(sudu);
                    }
                }
            }
        }
}

```

```

        sudu[i][j] = 0;
    }
}
return;
}

}

if (IsFinished(sudu))
{
    ++numm;
    return;
}
}

//老师给的完成数独的函数，做了一些修改，将打印数独结果修改成数独结果统计。

int main()
{
    int sudu[9][9]; //定义了一个存储数独的二维数组
    char suduchar[9][9]; //定义了一个用于存储数独字符形式的二维数组
    int row = 0, val = 0, t = 0; //初始化了行、列和统计变量的值
    int sumtotal = 0; //初始化了统计变量的值，下面会用到
    for (row = 0; row <= 8; ++row)
    {
        for (val = 0; val <= 8; ++val)
        {
            cin >> suduchar[row][val];
            sudu[row][val] = suduchar[row][val] - '0';
        }
    }
    //这里利用了输入字符转换为数值的方法，确保自由输入任意数独，而且支持换行
    for (row = 0; row <= 8; ++row)
    {
        for (val = 0; val <= 8; ++val)
        {
            if (sudu[row][val] == 0)
            {
                continue;
            }
            else
            {
                mask_ls[sumtotal].numdata = sudu[row][val];
                mask_ls[sumtotal].rowdata = row;
                mask_ls[sumtotal].valdata = val;
            }
        }
    }
}

```

```

        ++sumtotal;
    }
}
} //这里对于初始数独进行了扫描，将其中已知数字信息存储mask_1s中
int i0 = 0, j0 = 0, k0 = 0;
for (i0 = 0; i0 <= sumtotal - 1; ++i0)
{
    for (j0 = i0 + 1; j0 <= sumtotal - 1; ++j0)
    {
        for (k0 = j0 + 1; k0 <= sumtotal - 1; ++k0)
        {
            ++total;
        }
    }
}
} //这里事先统计了已知数字搭配的总数量，得到了总进度的大概数字。
for (i0 = 0; i0 <= sumtotal - 1; ++i0)
{
    for (j0 = i0 + 1; j0 <= sumtotal - 1; ++j0)
    {
        for (k0 = j0 + 1; k0 <= sumtotal - 1; ++k0)
            //这里增加了调换三个数字的循环操作
        {
            sudu[mask_1s[i0].rowdata][mask_1s[i0].valdata] = 0;
            sudu[mask_1s[j0].rowdata][mask_1s[j0].valdata] = 0;
            sudu[mask_1s[k0].rowdata][mask_1s[k0].valdata] = 0;
            //这里增加了第三个数字的操作
            FillSudu(sudu);
            //求解数独
            mask_2s[t].number = numm;
            mask_2s[t].firstonerow = mask_1s[i0].rowdata;
            mask_2s[t].firstoneval = mask_1s[i0].valdata;
            mask_2s[t].secondonerow = mask_1s[j0].rowdata;
            mask_2s[t].secondoneval = mask_1s[j0].valdata;
            mask_2s[t].thirdonerow = mask_1s[k0].rowdata;
            mask_2s[t].thirdoneval = mask_1s[k0].valdata;
            //同样的，这里增加了第三个数字的操作
            printf("完成进度: %d/%d:该进度计算出的数量%d\n", t + 1, total,
numm);

            //打印已完成进度和总进度的关系，并且打印每次进度计算出的数量
            numm = 0;
            ++t;
            sudu[mask_1s[i0].rowdata][mask_1s[i0].valdata] =
mask_1s[i0].numdata;
            sudu[mask_1s[j0].rowdata][mask_1s[j0].valdata] =

```

```

mask_1s[j0].numdata;
        sudu[mask_1s[k0].rowdata][mask_1s[k0].valdata] =
mask_1s[k0].numdata;
        //同样的，这里增加了第三个数字的操作
    }
}

} //这里计算每种数字搭配的解的信息并存储到mask_2s中
result.number = 0;
for (int w = 0; w <= t - 1; ++w)
{
    if (mask_2s[w].number > result.number)
    {
        result.number = mask_2s[w].number;
        result.firstonerow = mask_2s[w].firstonerow;
        result.firstoneval = mask_2s[w].firstoneval;
        result.seconderow = mask_2s[w].seconderow;
        result.secdoneval = mask_2s[w].secdoneval;
        result.thirdonerow = mask_2s[w].thirdonerow;
        result.thirdoneval = mask_2s[w].thirdoneval;
    }

    //比较所有解中最大的那一个数字，并且将相关信息存储result中
    printf("已完成：最终结果：当去掉第%d行第%d列的数字和第%d行第%d列和第%d行第%d
列的数字时，可以得到最多%d个不同种类的数独终局", result.firstonerow + 1,
result.firstoneval + 1, result.seconderow + 1, result.secdoneval + 1,
result.thirdonerow + 1, result.thirdoneval + 1, result.number);
    //打印结果
}

```

同样地，我们可以继续更改程序，成为去掉四个数字的数独计算程序，**那么这个程序我也是同样放在了selectablework的压缩包里，文件夹名字为homework0103。**

```
D:\通用文件夹\数据结构与算法 >
完成进度: 2053/8855:该进度计算出的数量30959
完成进度: 2054/8855:该进度计算出的数量13533
完成进度: 2055/8855:该进度计算出的数量29858
完成进度: 2056/8855:该进度计算出的数量10845
完成进度: 2057/8855:该进度计算出的数量10420
完成进度: 2058/8855:该进度计算出的数量15778
完成进度: 2059/8855:该进度计算出的数量54219
完成进度: 2060/8855:该进度计算出的数量18914
完成进度: 2061/8855:该进度计算出的数量16122
完成进度: 2062/8855:该进度计算出的数量3785
完成进度: 2063/8855:该进度计算出的数量23852
完成进度: 2064/8855:该进度计算出的数量14476
完成进度: 2065/8855:该进度计算出的数量4172
完成进度: 2066/8855:该进度计算出的数量13033
完成进度: 2067/8855:该进度计算出的数量6646
完成进度: 2068/8855:该进度计算出的数量15379
完成进度: 2069/8855:该进度计算出的数量10627
完成进度: 2070/8855:该进度计算出的数量11144
完成进度: 2071/8855:该进度计算出的数量15796
完成进度: 2072/8855:该进度计算出的数量10047
完成进度: 2073/8855:该进度计算出的数量16815
完成进度: 2074/8855:该进度计算出的数量109379
完成进度: 2075/8855:该进度计算出的数量35571
完成进度: 2076/8855:该进度计算出的数量32902
完成进度: 2077/8855:该进度计算出的数量8182
完成进度: 2078/8855:该进度计算出的数量24291
完成进度: 2079/8855:该进度计算出的数量13861
完成进度: 2080/8855:该进度计算出的数量3942
完成进度: 2081/8855:该进度计算出的数量16738
```

那么我们会发现，这个程序在运行了整整一个下午加上晚上大概八九个小时的时间，都没有运算出结果，经过我对其五个进度的计算得到，这个程序运行结束需要三十多个小时，所以我就直接结束了这个程序。由于这个程序没有做出太大的改动，只是增加了一些数字搭配算法，所以我就不在word里放代码了。

```
Microsoft Visual Studio 调试 >
050600000
10000002
完成进度: 1/23:该进度计算出的数量144
完成进度: 2/23:该进度计算出的数量7
完成进度: 3/23:该进度计算出的数量72
完成进度: 4/23:该进度计算出的数量175
完成进度: 5/23:该进度计算出的数量159
完成进度: 6/23:该进度计算出的数量11
完成进度: 7/23:该进度计算出的数量246
完成进度: 8/23:该进度计算出的数量148
完成进度: 9/23:该进度计算出的数量69
完成进度: 10/23:该进度计算出的数量57
完成进度: 11/23:该进度计算出的数量108
完成进度: 12/23:该进度计算出的数量933
完成进度: 13/23:该进度计算出的数量73
完成进度: 14/23:该进度计算出的数量68
完成进度: 15/23:该进度计算出的数量8
完成进度: 16/23:该进度计算出的数量234
完成进度: 17/23:该进度计算出的数量23
完成进度: 18/23:该进度计算出的数量44
完成进度: 19/23:该进度计算出的数量172
完成进度: 20/23:该进度计算出的数量93
完成进度: 21/23:该进度计算出的数量111
完成进度: 22/23:该进度计算出的数量150
完成进度: 23/23:该进度计算出的数量189
已完成: 最终结果: 当去掉第5行第4列的数字时, 可以得到最多933个不同种类的数独终局
D:\通用文件夹\数据结构与算法\第一次作业: 数独求解\作业尝试区\数独一次调换结果\homework0104\x64\Debug\homework0104.exe (
进程 29036)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . . .
```

同样地，我们会发现，我们还漏了只去掉一个数字的情况，我们把程序修改成只去掉一个数字的程序，再运行出上面这个结果，[具体代码参见selectablework的压缩包homework0104文件夹。](#)由于代码也是比较小的改动，就不在word放出代码了。

下面，我们就成功完成了去掉一个数字、两个数字、三个数字和四个数字（部分）的情况了，明显，这与我们想要的去掉所有数字的情况相去甚远，所以，我准备使用matlab工具对于数据进行处理分析，希望通过我们已知的情况推导出一个相关函数，并且求解出在去

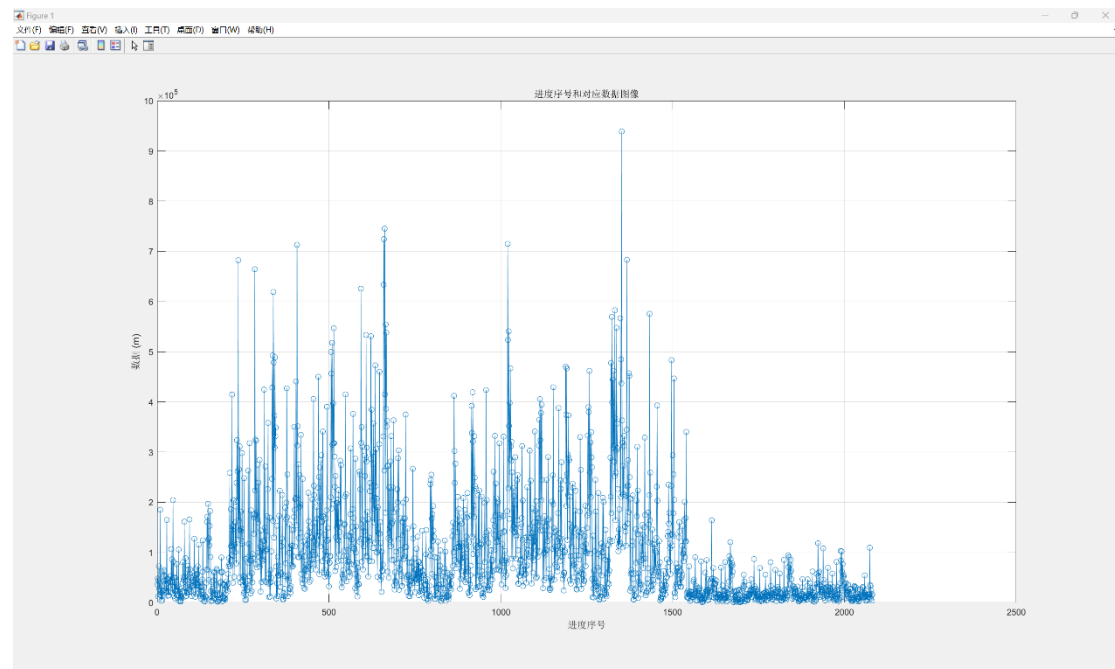
掉所有数字的情况下，数独的最终数字。

对于第一次调试到第三次调试而言，我们实际上不需要使用matlab程序，因为它已经实际上算出了最多的终局数量，它们分别是933、51088、98783。那么比较麻烦的是第四次调试，因为第四次调试没有算出最终结果，但是我们可以使用matlab分析一下它已经生成数字的最大值。

我们在matlab输入源代码：

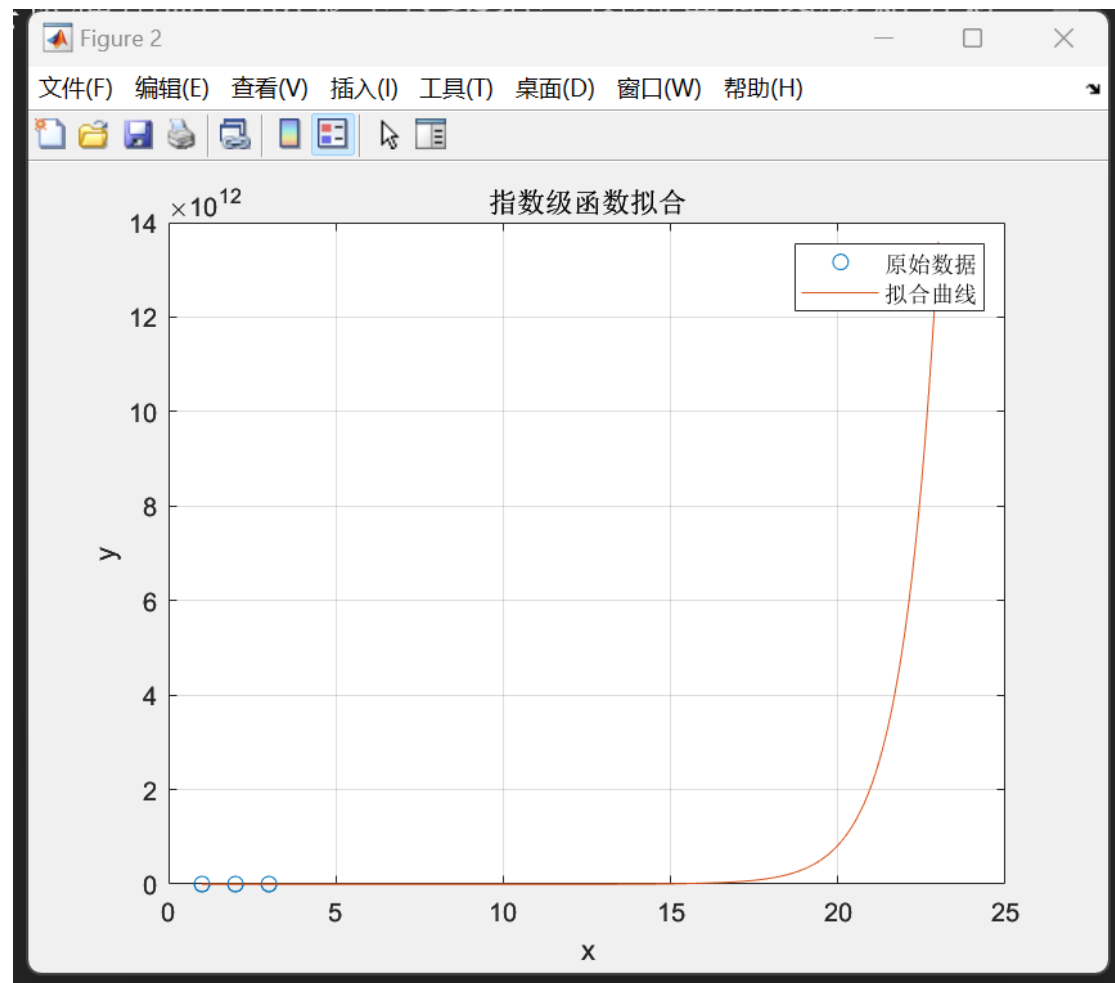
```
fileID = fopen('四次调换结果.txt', 'r'); % 替换 'your_data.txt' 为实际的文件路径
if fileID == -1
    error('无法打开文件');
end
data = textscan(fileID, '完成进度：%d/8855:该进度计算出的数量%d', 'Delimiter',
'\n');
fclose(fileID);
n = data{1};
m = data{2};
plot(n, m, '-o');
title('进度序号和对应数据图像');
xlabel('进度序号');
ylabel('数据 (m)');
grid on;
```

随后，我们就可以得到我们已经求出的四次调换数据图像



可以发现，最大值为938936，而且整个数据呈现一种极不规律的状态进行，这意味着我们难以估计出四次调换的数据最大值，这意味着第四次调换数据无法使用了。这同时意味着，我们依靠四个数据拟合函数的方法是很难得到结果的，所以，我们只能退而求其次，尝试只使用调换三次的数据得到拟合的函数图像及其方程，或许能做出一定的进展。由于已知数字的总数量是23，而最终量级达到了10的21次方，所以我猜测拟合函数是一个指数

级函数，我尝试使用matlab求出这条拟合函数曲线图像和方程。



以上就是拟合出来的图像了，可以发现，虽然绘制出来了，但是数量级仍然差了10次方左右，matlab源代码如下：

```
x = [1, 2, 3];
y = [933, 51088, 98783];
model = @(a, b, x) a * exp(b * x);
startParams = [1, 1];
fittedModel = fit(x', y', model, 'StartPoint', startParams);
xValues = 1:0.1:23;
yFit = feval(fittedModel, xValues);
figure;
plot(x, y, 'o', xValues, yFit);
title('指数级函数拟合');
xlabel('x');
ylabel('y');
legend('原始数据', '拟合曲线');
grid on;
xValue = 23;
yValue = feval(fittedModel, xValue);
fprintf('在 x = %d 时的值为: %e\n', xValue, yValue);
```

另外，我们求取这个函数的方程源代码：

```
a = fittedModel.a;
b = fittedModel.b;
fitFunction = @(x) a * exp(b * x);
xValue = 23;
yValue = fitFunction(xValue);
fprintf('拟合的函数是: y(x) = %e * exp(%ex)\n', a, b);
fprintf('在 x = %d 时的值为: %e\n', xValue, yValue);
可以得到，方程如下图：
```

```
拟合的函数是: y(x) = 6.096845e+03 * exp(9.358826e-01x)
在 x = 23 时的值为: 1.359639e+13
>>
```

总的来看，数量级还是没达到要求，看来后者的方法还是无法成立。

我后来又尝试了前者的方法，就是使用更高效率的数独算法进行求解。根据我在网上资料的搜索得知，对于可以用回溯法解决的问题，实际上可以通过最小剩余值（MRV）算法进行改进，这个算法看上去很高端，但底层逻辑却是十分简单，因为它仅仅只是在回溯法的基础上进行了小小的改动。就拿我们的数独求解问题而言，一般的回溯法是从左上角的粗宫格开始，然后往后一步步进行穷举运算，在计算机程序中我们往往会习惯地使用这种方法，这种算法简单但是效率低。我们不妨从一个真正的人的角度去思考一下如何解决数独问题，对于此，我也去网上查询了一下，那么我发现了很好的方法，就是我们纵观整个数独，然后在其中找到候选数最小的那一个小宫格，然后再一个个去试，根据候选数进行穷举是比顺序穷举更好的办法，这种办法被网上称之为最小剩余值算法，英文简称是MRV，那么，我们如何在真正的程序实现这个方法呢？

我的想法是，整个程序需要重新编写，我们的数独验证部分需要改成把所有的粗宫格和行列都验证一边，然后回溯的初始位置也需要一个算法来判断我们首先运算的最小候选的格子是哪个，它的具体信息又是什么，于是，就有下面的程序：

这个程序详见文件夹homework0105

```
#include <iostream>
using namespace std;

const int N = 9; // 数独9x9的大小

struct Cell {
    int row;
    int col;
    int numCandidates;
    bool candidates[10];

    Cell() : row(0), col(0), numCandidates(N) {
        for (int i = 0; i < 10; i++) {
            candidates[i] = true;
        }
    }
}
```

```

    }

    Cell(int r, int c) : row(r), col(c), numCandidates(N) {
        for (int i = 0; i < 10; i++) {
            candidates[i] = true;
        }
    }
    // 构造函数，用于初始化单元格的信息
};

// 检查在指定位置 (row, col) 放入数字 num 是否是安全的
bool isSafe(int grid[N][N], int row, int col, int num) {
    // 检查行是否安全
    for (int x = 0; x < N; x++) {
        if (grid[row][x] == num) {
            return false;
        }
    }

    // 检查列是否安全
    for (int y = 0; y < N; y++) {
        if (grid[y][col] == num) {
            return false;
        }
    }

    // 检查3x3子格是否安全
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }

    return true;
}

// 查找剩余的空单元格中具有最少候选数字的单元格，并更新 cell 变量
bool findEmptyLocation(int grid[N][N], Cell& cell) {
    int minCandidates = N + 1;
    for (int i = 0; i < N; i++) {

```

```

        for (int j = 0; j < N; j++) {
            if (grid[i][j] == 0) {
                // 创建一个新的 Cell 对象来追踪当前单元格的信息
                Cell newCell(i, j);
                for (int num = 1; num <= 9; num++) {
                    if (isSafe(grid, i, j, num)) {
                        newCell.candidates[num] = false;
                        newCell.numCandidates--;
                    }
                }
                if (newCell.numCandidates < minCandidates) {
                    minCandidates = newCell.numCandidates;
                    cell = newCell;
                }
            }
        }
    }
    return minCandidates < N + 1;
}

// 递归解数独
bool solveSudoku(int grid[N][N]) {
    Cell cell;

    if (!findEmptyLocation(grid, cell)) {
        // 如果没有空位置了，数独已解决
        return true;
    }

    for (int num = 1; num <= 9; num++) {
        if (cell.candidates[num]) {
            grid[cell.row][cell.col] = num;
            if (solveSudoku(grid)) {
                return true;
            }
            grid[cell.row][cell.col] = 0;
        }
    }

    // 无法找到解决方案
    return false;
}

int main() {

```

```

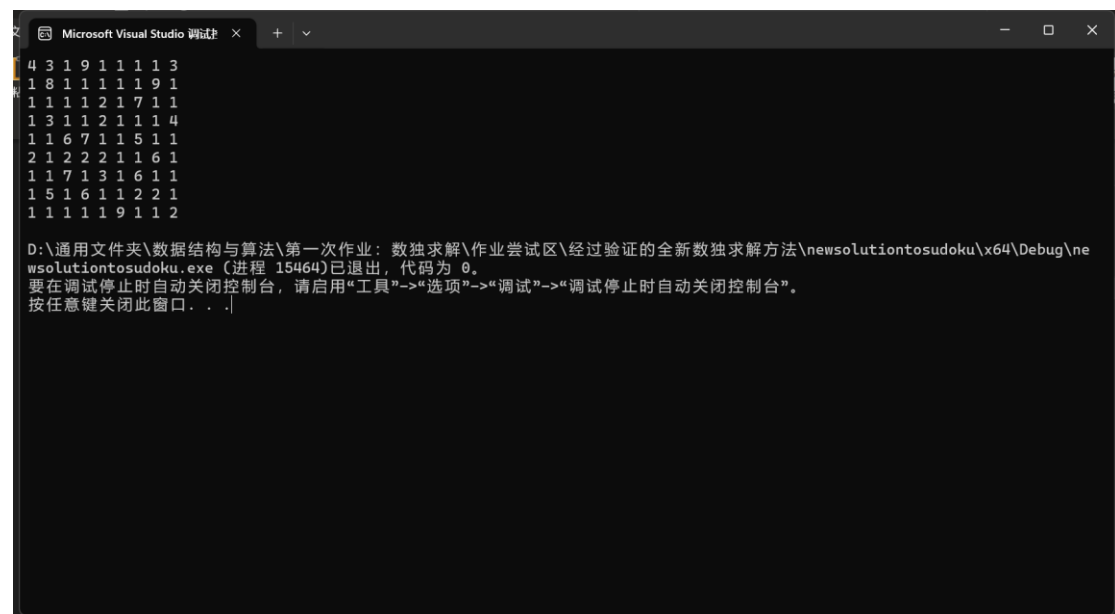
int grid[N][N] = {
    {4, 0, 0, 9, 0, 0, 0, 0, 3},
    {0, 8, 0, 0, 0, 1, 0, 9, 0},
    {0, 0, 0, 0, 2, 0, 7, 0, 0},
    {0, 3, 0, 0, 0, 0, 0, 0, 4},
    {0, 0, 6, 7, 0, 0, 5, 0, 0},
    {2, 0, 0, 0, 0, 0, 0, 6, 0},
    {0, 0, 7, 0, 3, 0, 6, 0, 0},
    {0, 5, 0, 6, 0, 0, 0, 0, 0},
    {1, 0, 0, 0, 0, 9, 0, 0, 2}
};

if (solveSudoku(grid)) {
    // 打印解决方案
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << grid[i][j] << " ";
        }
        cout << endl;
    }
}
else {
    cout << "无解" << endl;
}

return 0;
}

```

那么，这个是运行结果



```

Microsoft Visual Studio 调试
4 3 1 9 1 1 1 1 3
1 8 1 1 1 1 1 9 1
1 1 1 1 2 1 7 1 1
1 3 1 1 2 1 1 1 4
1 1 6 7 1 1 5 1 1
2 1 2 2 2 1 1 6 1
1 1 7 1 3 1 6 1 1
1 5 1 6 1 1 2 2 1
1 1 1 1 9 1 1 2

D:\通用文件夹\数据结构与算法\第一次作业：数独求解\作业尝试区\经过验证的全新数独求解方法\newsolutiontosudoku\x64\Debug\newsolutiontosudoku.exe (进程 15464) 已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。 . .

```

运行数独明显变快了，但是有一个小缺陷，就是没办法显示全部的解，而且受限于时间关系，还没来得及加入我之前的一些特殊功能，比如高效率输入的办法。

在我准备进一步把这个程序改造成可以解决最后一个问题之前，我准备去网上再确定一下这个算法到底能加快多少，能不能成功解决那个难题。但是很遗憾的是，我查到的资料显示，使用MRV算法只能使数独求解数独提高百分之三十至百分之五十，虽然提高了很多，但是按照百分之五十计算，我们还是没办法在有生之年在我们的个人计算机上算出这个数字，因为即使是去除四个数字的运算就需要两天的时间，而去除所有的已知数字则是指数级的增加，没有完成的可能性。

以上就是我尝试解决开放题的全部过程，总的来说，在前文中的严密的数学证明里其实已经证明出了这个开放题的答案就是 6.67×10^{21} 次方，但是在后续的计算机程序求解的过程中，却发现以现有计算机的算力和笔者有限的算法知识难以得到最终结果。