作业 HW2 实验报告

姓名：刘博洋 学号：2153538 日期：2022 年 10 月 11 日

1. 涉及数据结构和相关背景

栈和队列是使用频率最高的数据结构。栈和队列是指插入和删除只能在表的一端进行的线性表。从数据结构的角度来看，栈和队列也是线性表，其特殊性在于栈和队列的基本操作是线性表操作的子集，它们是操作受限的线性表，因此可称为限定性的数据结构。

栈具有后进先出的特性，如果问题解决具有先进后出的天然特性的话，则求解的算法就要使用栈。比如：进制转换，括号匹配，表达式求值，函数调用，递归调用，八皇后问题，迷宫求解等。

队列具有先进先出的特性，如果问题解决具有先进先出的特性的话，则求解的算法就要使用队列。比如：脱机打印操作，进程按等待时间创建优先级队列，信号按接受的先后顺序依次处理

学好这两种数据结构是在为后面更多更复杂的数据结构打好基础，重要是不仅要理解其思想，更要能主动去使用去实现。

2. 实验内容

2.1 最长子串

2.1.1 问题描述

已知一个长度为 n，仅含有字符'('和')'的字符串，请计算出最长的正确的括号子串的长度及起始位置，若存在多个，取第一个的起始位置。

子串是指任意长度的连续的字符序列。

例 1：对字符串 "(()()))()"来说，最长的子串是"(()())"，所以长度=6，起始位置是 0。

例 2：对字符串")())"来说，最长的子串是"()",子串长度=2，起始位置是 1。

例 3；对字符串""来说，最长的子串是"",子串长度=0，空串的起始位置规定输出 0。

字符串长度：0≤n≤1*105

对于 20%的数据：0<=n<=20

对于 40%的数据：0<=n<=100

对于 60%的数据：0<=n<=10000

对于 100%的数据：0<=n<=100000

2.1.2 基本要求

输入：

一行字符串

输出：

子串长度，及起始位置（数字）

2.1.3 数据结构设计

当时考虑了两种算法；第一种是用栈的思想

具体做法是始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」，这样的做法主要是考虑了边界条件的处理，栈里其他元素维护左括号的下标：

一旦遇到‘(’ ，将它的放入栈中，一旦遇到 ‘)’ ，如果此时栈为空，则不能统一起来，于是我们预设一个-1 的元素在栈底下方，一旦栈为空则可以统一起来减去-1，如果栈不为空，先弹出栈顶元素表示匹配了当前右括号，此时有两种情况：

1、如果栈为空，说明当前的右括号为没有被匹配的右括号，将其下标放入栈中来更新我们之前提到的「最后一个没有被匹配的右括号的下标」

2、如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」

从而一直循环从前往后遍历字符串并更新答案，需要注意的是，如果出现两个相同长度的子串，取前面一个。

第二种就是利用顺序表的思想，直接将字符串存入顺序表，再进行判断。

利用两个计数器 left 和 right 。首先，从左到右遍历字符串，对于遇到的每个‘(’，我们增加 left 计数器，对于遇到的每个 ‘)’ ，增加 right 计数器。每当 left 计数器与 right 计数器相等时，计算当前有效字符串的长度，并且记录目前为止找到的最长子字符串。当 right 计数器比 left 计数器大时，将 left 和 right 计数器同时变回 0

这样的做法贪心地考虑了以当前字符下标结尾的有效括号长度，每次当右括号数量多于左括号数量的时候之前的字符都扔掉不再考虑，重新从下一个字符开始计算，但是当左括号数量一直大于右括号时会出现问题，这时候只需要再从右往左遍历用类似的方法计算即可，只是这个时候判断条件反过来。

2.1.4 功能说明（函数、类）

```
typedef struct {
	char a;
	int num;
}elem;
typedef struct   {
	elem* base;
	elem* top;
	int stacksize;
}sqstack;
sqstack s;//定义一个栈
Status Initstack(sqstack& s)
{
	s.base = (elem*)malloc(sizeof(elem) * STACK_INIT_SIZE);
	if (!s.base) exit(OVERFLOW);
	s.top = s.base;
	s.stacksize = STACK_INIT_SIZE;
}//初始化栈
Status push(sqstack &s,char a,int i)
{
	if (s.top - s.base >= s.stacksize)
```

```
	{
		s.base = (elem*)realloc(s.base, (s.stacksize + 100) *
sizeof(elem));
		if (!s.base) return (OVERFLOW);
		  s.top = s.base + s.stacksize;
		  s.stacksize += 100;
	}
	s.top++;
	s.top->a = a;
	s.top->num = i;

}//进栈操作
Status pop(sqstack &s)
{
	if (s.top == s.base)return ERROR;
	s.top--;

}//弹出栈顶元素
void input(string &l)
{
	cin>>l;

}
int count(sqstack& s,string &l,int &n)
{
	int last=-1;
	int ans=0;
	for (int i = 0;i < l.length();i++)
	{
		if (l[i] != '(' && l[i] != ')')
			return ans;
		if (l[i] == '(')
		{
			push(s, l[i],i);

		}

		if (l[i] == ')')
		{
			if (s.base == s.top)
				last = i;
			else {
				pop(s);
				if (s.base == s.top)
```

```
                        {
                              if (i - last > ans)
                                    n = last + 1;
                              ans = max(ans, i - last);

                        }
                        else {
                              if (i - s.top->num >ans)
                                    n = s.top->num + 1;//更新位置
                               ans = max(ans, i - s.top->num);//更新最长子串
长度

                        }
                  }
            }
      }
      return ans;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

1.第一次生成的十个测试数据全对，最后 oj 给分只有 17 分，通过咨询老师得知问题在于最后答案的输出，原本的语句是 cout << count(s, 1, n) << " "<<n;
这样输出在 vs 编译器中没有问题，但是在有些编译器中可能 n 输出的是函数调用前的值，导致错误。这个和编译器有关，变量取值时机不一定是按照左右顺序的，为了保证严谨和一致性，应该采用分开输入的方式。
2.在改掉上面的错误后，还是没能得到满分，再次测试数据得到，当遇到一个字符串中两个长度相等时，取的是后面一个的位置，因此将更新的条件去掉等号，即只有新值大于原值得时候位置和长度才进行更新。

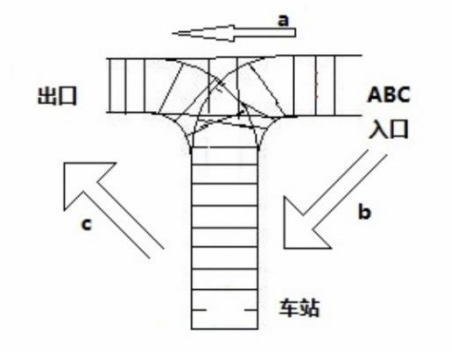
2.1.6 总结和体会

此题看似容易，实际上思维含量还是相当大的，我认为本题的难点在于如何判断遇到右括号是栈为空的情况。因为只要栈里有左括号，遇到右括号与左括号相消即可，但是如果单独出现不合法的右括号，就无法相消，比较难判断。这里需要想到在栈中预置一个-1 的下标，如果弹出栈元素后栈为空，用下标减去-1 得到的也是正确结果。此题让我收获颇多，不论是对栈的理解上，还是对代码的严谨性上都有了一定的提升。学习栈就是要用栈的思想去进行先进先出的判断，其中要抓住进栈和出栈的条件仔细斟酌。检查代码时不能仅仅依据编译器的结果来认为自己代码是正确的，要考虑到不同编译器的差异，尽量避免容易出现差异的地方。

2.2 列车进栈

2.2.1 问题描述

每一时刻，列车可以从入口进车站或直接从入口进入出口，再或者从车站进入出口。即每一时刻可以有一辆车沿着箭头 a 或 b 或 c 的方向行驶。 现在有一些车在入口处等待，给出该序列，然后给你多组出站序列，请你判断是否能够通过上述的方式从出口出来。



2.2.2 基本要求

输入：
第 1 行，一个串，入站序列。
后面多行，每行一个串，表示出栈序列
当输入=EOF 时结束
输出：
多行，若给定的出栈序列可以得到，输出 yes,否则输出 no。

2.2.3 数据结构设计

本题的栈比较明显，就是车站，列车进入车站后是先进后出，后进先出的，但是到达出口还可以直接从入口实现，所以我们要做的就是判断给定入口的序列和出口的序列，中间能否经过任意步入栈出栈或直接出口的方式得到出口的序列。对于每一步，都有三种操作：
（1）从入口直接进入出口
（2）从入口进栈
（3）从栈顶到出口
那么我们定义三个序列：入口序列（已知）出口序列（已知）结果序列（未知）
分别用三个指针指向三个序列的头。由于每一步走向出口只可能从栈顶到出口或者从入口到出口，每一次判断入口序列指针指向是否和出口序列指针指向内容相同，如果相同，那么直接将该元素从入口到出口结果序列便可以满足要求，如果不相同，那么判断栈顶元素和出口指针指向元素是否相同，相同则栈顶元素出栈去出口存入结果序列。如果某一时刻栈顶和入口都没有和出口要求相同的元素，那么使入口元素入栈，最后当入口指针为空循环结束，此时栈中可能还有元素，那么将栈中元素再出栈存入结果序列中，最后比较结果序列和给定的出口序列是否相同，便可判断能否从出口得到此序列。此题是一种贪心的思想，只要满足给出的结果序列就匹配移动。由于只有一个栈，如果按这种规则最后不能得到给定的序列，说明不能完成此操作。

2.2.4 功能说明（函数、类）

```
typedef struct {
      char* base;
      char* top;
      int stacksize;
}sqstack;
sqstack L;//定义一个栈
Status Initstack(sqstack &L)
{
```

```
	L.base = (char*)malloc(sizeof(char) * 500);
	if (!L.base)return (OVERFLOW);
	L.top = L.base;
	L.stacksize = STACK_INITSIZE;
}//初始化栈
Status push(sqstack& L,char m)
{
	if (L.top - L.base >= L.stacksize)
		L.base = (char*)realloc(L.base, (STACK_INITSIZE + 10) *
sizeof(char));
	if (!L.base)return (OVERFLOW);
	L.top++;
	*L.top = m;

}//进栈操作
void input(char s[])
{
	cin >> s;
}
void judge(char s[], char l[], sqstack& L)
{
	char out[100]="";
	char* p,*q,*k;
	p = &(l[0]);
	q = &(s[0]);
	k = &(out[0]);//定义三个指针
	while (*p!='\0' && *q!='\0')
	{
		if (*p == *q)
		{
			*k = *q;
			q++;
			p++;
			k++;
		}//相等直接挪
		else if (L.top != L.base && (*L.top == *p))
		{
			*k = *L.top;
			k++;
			p++;
			L.top--;
		}栈顶相等直接挪
		else {
			push(L, *q);
```

```
                q++;
        }//否则进栈
    }

    for (int j = 0;L.top != L.base;j++)
    {
        char* x;
        x = out;
        while (*x != '\0')
            x++;
        *x = *L.top;
        L.top--;
    }//把剩余栈中元素挪到结果序列
    bool flag = 1;
    for (int i = 0;out[i] != '\0';i++)
    {
        if (out[i] != l[i])
            flag = 0;

    }
    if(flag==1)
        cout << "yes"<<endl;
    else
        cout << "no"<<endl;
}//逐位比较
```

2.2.5 调试分析（遇到的问题和解决方法）

此题代码量不大但是思维含量大，要搞清楚出栈和进栈和直接跳过栈的条件，不能遗漏情况。此题调试分析时没有遇到很大问题，主要在于最后输入结束符 EOF 的判断，由于 oj 的输入是重定向到文本文档里的,所以这里使用 cin.eof()==1 来判断是否读到了 eof 文件末尾。

2.2.6 总结与收获

此题是一个用栈为工具来实现顺序改变的问题，要弄清楚进栈和出栈的条件，并且将每一步的三种情况分析清楚。这是一种贪心的思想，每一步都按照目标字符串进行操作，最后得到结果。但是这种方法在本题适用，当栈的个数增加时便不一定正确，如果有多个车站，那贪心算法求出的只是一种局部最优，不一定局部最优能保证整体能达到目标，可能某几步不是最优的操作最后反而能形成目标字符串。本题我完成的速度相比之前有了较大的提高，这说明我对这种数据结构的理解在加深，并且代码的正确率和熟练度有了提高。

2.3 题目三

2.3.1 问题描述

计算如下布尔表达式（V | V）& F & (F | V) 其中 V 表示 True，F 表示 False，| 表示 or，&表示 and，！表示 not（运算符优先级 not> and > or），用栈来求解。

2.3.2 基本要求

输入：
文件输入，有若干（A<=20）个表达式，其中每一行为一个表达式。 表达式有（N<=100）个符号，符号间可以用任意空格分开，或者没有空格，所以表达式的总长度，即字符的个数，是未知的。
对于 20%的数据，有 A<=5，N<=20，且表达式中包含 V、F、&、|
对于 40%的数据，有 A<=10，N<=50，且表达式中包含 V、F、&、|、!
对于 100%的数据，有 A<=20，N<=100，且表达式中包含 V、F、&、|、!、(、)
所有测试数据中都可能穿插空格
输出：对测试用例中的每个表达式输出“Expression”，后面跟着序列号和“: ”，然后是相应的测试表达式的结果（V 或 F），每个表达式结果占一行（注意冒号后面有空格）。

2.3.3 数据结构设计

本题实际上是用栈来求解表达式的值，我们建立两个运算栈，OPTR 栈存放运算符，OPND 栈存放运算数，输入一个表达式，从开头往后读，读到运算数则直接进入 OPND 栈。开始时先向 OPTR 栈进一个“#”，表示运算开始，读入表达式时直接在字符串后加一个“#”，第二次读到“#”则表示运算结束。如果读入的是运算符，首先要判断读入运算符和栈顶运算符的优先级，首先需要用一个函数确定优先级，这里只有三个运算符！，&，|和括号所以直接用条件判断就可以，当运算符比较多的时候可以用 map 映射函数表示优先级。需要注意的是括号也有优先级，因为总是先运算括号内的内容，所以规定：当左右括号相遇时两者运算优先级相同，当运算符在左括号右边时优先级大于左括号，在左括号左边时大于左括号，在右括号左边时优先级小于右括号，在右括号右边时优先级小于右括号。这样的话可以保证遇到括号总是先运算括号内内容。

据此绘制一个表格更加直观:规定 a 在 b 左边,则有

a \ b	#	!	&	|	(	)
#	>	<	<	<	<	<
!	>	>	>	>	>	<
&	>	<	>	>	>	<
|	>	<	<	>	>	<
(	>	<	<	<	>	=
)	>	>	>	>	=	>

如此，当运算符进栈时，如果要进栈的操作符优先级大于栈顶的操作符，那么进栈，如果要进栈的操作符优先级小于栈顶的操作符，那么先将栈顶运算符运算，要进栈的运算符不动，指针也不遍历，等到下一次循环的时候再判断；此时要运算，要先判断运算符的类型，如果是单目运算符！，那么建立一个函数，如果 OPND 栈顶是 V 返回 F，如果是 F 返回 V，如果是双目运算符，那么在 OPND 栈中 pop 出两个元素并用临时变量储存，运算后返回值再 push 进入 OPND 栈。如果要进栈的运算符优先级等于栈顶运算符，那么有两种情况，第一是两运算符都是#，表示运算结束，我们把这种情况放入循环终止调节中，于是只剩下一种情况，即左右括号相遇，那么表明括号内的运算式子已经运算完成，此时在 OPTR 栈中弹出左括号，指针++，跳过右括号即可。

2.3.4 功能说明（函数、类）

```
typedef struct {
```

```
	char* base;
	char* top;
	int stacksize;
}sqstack;
sqstack OPTR,OPND;
Status Initstack(sqstack &L)
{
	L.base = (char*)malloc(sizeof(char) * STACK_INITSIZE);
	if (!L.base) return OVERFLOW;
	L.top = L.base;
	L.stacksize = STACK_INITSIZE;
}//栈的类型定义及初始化

bool in(char c)
{
	if (c == '&' || c == '|' || c == '!' || c == '(' || c == ')'||c=='#')
		return true;
	else return false;
}//判断读入的是否是运算符
Status push(sqstack &L, char m)
{
	if (L.top - L.base >= L.stacksize)
		L.base = (char*)realloc(L.base, (STACK_INITSIZE + 10) *
sizeof(char));
	if (!L.base)return (OVERFLOW);
	L.top++;
	*L.top = m;
}//进栈操作
Status pop(sqstack &L)
{
	if (L.base == L.top)
		return ERROR;
	L.top--;
}//出栈操作
char priority(char a,char b)
{
	if (a == '(' && b == ')')return '=';
	else if (a == '#' && b == '#')return '=';
	else if (a == ')' && (b == '!' || b == '&' || b == '|'))return '>';
	else if (b == ')' && (a == '!' || a == '&' || a == '|'))return '>';
	else if (a == '(' && (b == '!' || b == '&' || b == '|'))return '<';
	else if (b == '(' && (a == '!' || a == '&' || a == '|'))return '<';
	else if (a == '!' && b == '&')return '>';
	else if (a == '!' && b == '|')return '>';
```

```
	else if (a == '&' && b == '|')return '>';
	else if (a == '&' && b == '!')return '<';
	else if (a == '|' && b == '!')return '<';
	else if (a == '|' && b == '&')return '<';
	else if ((b == '(' || b == ')' || b == '!' || b == '&' || b == '|') &&
a == '#')return '<';
	else if ((a == '(' || a == ')' || a == '!' || a == '&' || a == '|') &&
b == '#')return '>';
	else if ((a!='#'&& b != '#')&&a == b)return '<';
}//判断优先级
char operate1(char a)
{
	if (a == 'V')
		return 'F';
	else if (a == 'F')return 'V';
	else return ERROR;

}//单目运算符操作
char operate2(char a, char b, char n)
{
	if (a == 'V' && b == 'V')
	{
		if (n == '&')
			return 'V';
		if (n == '|');
		return 'V';
	}
	else if (a == 'V' && b == 'F')
	{
		if (n == '&')
			return 'F';
		if (n == '|');
		return 'V';
	}
	else if (a == 'F' && b == 'V')
	{
		if (n == '&')
			return 'F';
		if (n == '|');
		return 'V';
	}
	else if (a == 'F' && b == 'F')
	{
		if (n == '&')
```

```
                return 'F';
            if (n == '|');
            return 'F';
        }
}//双目运算符操作
void Caculate(char* exp,sqstack &OPTR,sqstack &OPND)
{
        push(OPTR, '#');
        while ( ( * exp != '#' || *OPTR.top != '#'))//两个#相遇即结束
        {
            if ((int)*exp == 32)
                exp++;
            else{
            if (in(*exp) == 0)
            {
                push(OPND, *exp);
                exp++;
            }
            else {
                switch (priority(*OPTR.top, *exp))//判断优先级
                {
                case'>':
                    if (*OPTR.top == '!')
                    {
                        char x = operate1(*OPND.top);
                        pop(OPTR);
                        pop(OPND);
                        push(OPND, x);
                    }//判断是否为单目运算符

                    else {
                        char m = *OPND.top;
                        pop(OPND);
                        char n = *OPND.top;
                        pop(OPND);
                        char ch;
                        ch = operate2(m, n, *OPTR.top);
                        pop(OPTR);
                        push(OPND, ch);

                    }//双目运算符则需取 pop 两次再 push
                    break;
                case'<':
                    push(OPTR, *exp);
```

```
                        exp++;break;

                    case'=':
                        pop(OPTR);
                        exp++;
                        break;
                    }
                }
                }
        }
}//执行运算的大循环
int main()
{
        Initstack(OPTR);
        Initstack(OPND);
        int i = 1;
        while(1)
        {
        char exp[500];
        char* p = exp;
        cin.getline(exp,500);//读入时忽略空格
        if(cin.eof() == 1)break;//判断文件结尾 eof
        strcat_s(exp, "#");//读入字符串时自动在末尾加#，方便运算时判断结束
        Caculate(exp, OPTR,OPND );
        cout <<"Expression " <<i<<": "<< * OPND.top<<endl;
        i++;
        }
//主函数
```

2.3.5 调试分析（遇到的问题和解决方法）

此题过程比较复杂，涉及到多种进栈出栈操作。在开始编写程序判断运算顺序和括号的优先级让我遇到了一些困难，在最后画图的辅助并结合一个实例遍历了一次运算流程后，弄清了两个栈实现表达式运算的原理和关键的判断条件。在最后测试数据时，发现有一部分的数据一直出现答案无法显示的情况，进行单步调试，归纳多组数据后发现是我开始没有定义两个左括号相遇时的优先级，导致运算到那一步陷入死循环。个人认为我这种判断优先级的方法不太好，当运算符较多时，容易漏掉情况，当运算达到没有考虑到的情况时则会无法判断而陷入死循环。

2.3.6 总结与收获

作为栈的典型应用问题，本题让我收获了栈这种数据结构典型的思想，以及针对较为复杂过程各个条件的判断。以及在做完题之后调试出错误的同时，也感觉到自己代码需要优化，鲁棒性比较低，一旦出现遗漏的情况就会陷入死循环。此题是分治的思想，将运算符和运算数分别放入两个栈，等到运算的时候再将他们取出运算，这样让栈的结构清晰明了。

2.4 队列的应用（求矩阵联通域）

2.4.1 问题描述

输入一个 n*m 的 0 1 矩阵，1 表示该位置有东西，0 表示该位置没有东西。所有四邻域联通的 1 算作一个区域，仅在矩阵边缘联通的不算作区域。求区域数。

2.4.2 基本要求

输入：第 1 行 2 个正整数 n，m，表示要输入的矩阵行数和列数

第 2—n+1 行为 n*m 的矩阵，每个元素的值为 0 或 1。

输出：

1 行，代表区域数

2.4.3 数据结构设计

首先读题明确什么是题目要求的区域

0	1	0	1	1
0	0	0	0	0

这两个 1 处于边界处，不构成一个区域

0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	1	0	0
0	1	0	1	1	1
0	0	0	0	0	0

上下左右四联通的 1 构成一个区域

0	0	0	0	0	1
0	0	0	0	0	0
0	0	0	1	0	0
0	1	0	1	1	1
0	0	0	0	0	0

单独的一个 1 也是一个区域

首先开辟一个 m*n 的二维数组，并输入表示给定的矩阵。再开辟一个 bool 类型的二维数组，大小也为 m*n，作为标记矩阵，首先令所有元素都为 0，表示没有访问过。

由于仅仅在边界上联通的不能算区域，所以从（1,1）开始遍历，经过每个不为 0 的点，计数器 count++，都向上下左右四个方向去探索，然后进行递归，对于探到是 1 的点继续执行以上操作，知道所有的探索停止，并且将所有探到的点标记为 1 并进队，表示已经访问过。

然后返回（1,1）继续依次遍历每一个不为 0 的点，直到结束。最后的 count 即为区域数。

2.4.4 功能实现（函数、类）

```
struct nod
{
    int x, y;
}node;

int X[] = { 0,0,1,-1 };        //方向增量数组
int Y[] = { 1,-1,0,0 };
int Map[1000][1000];

bool inq[1000][1000] = { false };              //判断元素是否入队

bool judge(int x, int y,int m,int n)
{
    if (x >= m || x < 0 || y >= n || y < 0) return false;
    if (inq[x][y] == true || Map[x][y] == 0) return false;
    //都不符合情况， 返回真
    return true;
}
void bfs(int x, int y,int m,int n)
{
    node.x = x, node.y = y;    //实参传递
    push(q);//进队
    inq[x][y] = true;
    while (!q.empty())
    {
        nod top = q.front();
        pop(q);//出队
        for (int i = 0; i < 4; i++)
        {
            int newx = top.x + X[i];
            int newy = top.y + Y[i];//向四个方向遍历
            if (judge(newx, newy,m,n))
            {
                node.x = newx, node.y = newy;
                q.push(node);
                inq[newx][newy] = true;
            }
        }
    }
nt main()
{
    int m, n;
    cin >> m >> n;
```

```
    for (int i = 0;i < m;i++)
    {
        for (int j = 0;j < n;j++)
            cin >> Map[i][j];
    }
    int ans = 0;
    for (int i = 1; i < m-1; i++)
        for (int j = 1; j < n-1; j++)
        {
            if (Map[i][j] == 1 && inq[i][j] == false)
            {
                ans++;
                bfs(i, j,m,n);
            }
        }//进行递归操作直到找不到
    cout << ans ;
    return 0;
}
```

2.4.5 调试与分析

本题实际上用的队列比较少，主要用来标记每个元素是否进过队，即是否被访问过，本题更多是一种搜索和回溯的算法，即规定一个目标，直到没有达到这个目标就回到原点，再换一个点进行下一次遍历，在这种搜索中，不遗漏不重复是关键，此题调试过程中没有遇到太多问题，只是做题之前对题目的理解和思考算法耗时较长。

2.4.6 总结与收获

通过此题收获了一种深度搜索的思想，即把目标放在优先位置，一旦达到目标则继续探索，直到所有的分支都无法达到目标，回溯到原来的位置，再挪动进行下一次操作。

2.5 队列中的最大值

2.5.1 问题描述

此题给定一个队列，要求实现有下列 3 个基本操作：

（1）Enqueue(v): v 入队

（2）Dequeue(): 使队首元素删除，并返回此元素

（3）GetMax(): 返回队列中的最大元素

请设计一种数据结构和算法，让 GetMax 操作的时间复杂度尽可能地低。

要求运行时间不超过一秒

2.5.2 基本要求

输入：

第 1 行 1 个正整数 n, 表示队列的容量(队列中最多有 n 个元素)

接着读入多行，每一行执行一个动作。

若输入"dequeue"，表示出队，当队空时，输出一行“Queue is Empty”;否则，输出出队的元素；

若输入"enqueue m"，表示将元素 m 入队,当队满时(入队前队列中元素已有 n 个)，输出

"Queue is Full"，否则，不输出；
若输入"max",输出队列中最大元素，若队空，输出一行“Queue is Empty”。
若输入"quit",结束输入，输出队列中的所有元素

输出：
多行，分别是执行每次操作后的结果

2.5.3 数据结构设计

此题入队操作和从队首出队操作比较容易实现，属于队列的基本操作。但是注意此题要求使 getmax 函数的时间复杂度尽可能低，我们注意到，队列 S 的元素可能被出队（删除）。如果我们记录一个 max_value，并且把 max_value 作为当前队列的最大值，在每一次入队的时候都与 max_value 比较，如果小于 maxvalue 就不入队，就可以在只有入队操作的情况下，用 O(1)的时间复杂度查询到当前队内最大的元素。但当出现出队操作时，如果原先记录的最大值被从 SS 中删除了，就需要重新遍历队列以寻找新的最大值。我们必须要放弃传统的遍历整个队列求最大值的方法，自己设计一种数据结构。这里我考虑构造一个递减队列，维护这个队列的队头始终为最大值。

既然始终维护所有数的最大值比较难以实现，我们可以考虑维护某一个区间上的最大值。

如一个队列为

元素	1	8	6	2	5	7	3	4	5	13	2	1	3	5
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13

观察知

元素	1	8	6	2	5	7	3	4	5	13	2	1	3	5
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13

在区间[0,6]上的最大值为 8，区间[1,6]上的最大值也为 8
在区间左端不端+1，即元素不断出队时，

元素	1	8	6	2	5	7	3	4	5	13	2	1	3	5
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13

区间[2,10]上的最大值为 13，

元素	1	8	6	2	5	7	3	4	5	13	2	1	3	5
下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13

区间[2,8]上的最大值为 7
把这些最大值标记出来，发现其变化规律为 8->7->13->5

画出图示

元素	1	8	6	2	5	7	3	4	5	13	2	1	3	5

下标	0	1	2	3	4	5	6	7	8	9	10	11	12	13

可以发现蓝色框内的元素，右边都存在比它大的元素。可以说明，如果当前队列 S 中两个元素的下标为 i,j，且 i<j,S[i]<S[j]（即只要 S[i]的右边存在比它大的元素），S[i]不可能成为未来（任意次入队出队之后）某一时刻的队内最大元素：

如果比 S[i]下标更大的元素是当前最大元素，那么直到 S[i]出队，S[i]都不可能成为最大元素。
如果当前的最大元素比 S[i]的下标要小,那在其出队之后,新的最大元素将会在其右侧，最后会变为第一种情况，即最大元素在 S[i]右侧。

所以在这个队列中把不可能成为最大元素的元素都删去，最后得到的就是分别为第一个最大值、第二个值、第三个最大值的序列。
具体操作是，每一次入队时，直接进入存放所有元素的原队列，而对于辅助求最大值的辅助队列，如果要入队的元素大于队尾元素，就一直把队尾元素删掉，直到遇到一个比它大的元素或者队列为空，将这个元素插入队尾，最后可得一个非递增的包含所有可能最大值的序列。

2.5.4 功能说明（函数、类）

```
typedef int Status;
typedef struct QNode{
	int data;
	struct QNode* next;
	struct QNode* prior;
}QNode, *queueptr;
typedef struct {
	queueptr front;
	queueptr rear;
}LinkQueue;
LinkQueue Q,Q1;//Q 为原队列，Q1 为辅助队列
int coun = 0;记录原队列有多少个数，便于判断是否满
Status Initqueue(LinkQueue& Q, int n)
{
	Q.front = Q.rear =(queueptr)malloc(sizeof(QNode));
	  if (!Q.front) exit(OVERFLOW);
	Q.front->next = NULL;
	return OK;
}//初始化队列
void enqueue(int m, LinkQueue& Q,int n)
{
	if (coun == n)
	{
		cout << "Queue is Full" << endl;
```

```
	}
	else {
		queueptr p = (queueptr)malloc(sizeof(QNode));
		if (!p)exit(OVERFLOW);
		p->data = m;
		p->next = NULL;
		Q.rear->next = p;
		Q.rear = p;
		coun++;
	}
}//原队列进队，需要判断是否满，进队成功 coun++
void enqueueQ1(int m, LinkQueue& Q, int n)
{
		queueptr p = (queueptr)malloc(sizeof(QNode));
		if (!p)exit(OVERFLOW);
		p->data = m;
		p->next = NULL;
		Q.rear->next = p;
		Q.rear = p;
}//辅助队列进队
void DequeueQ(LinkQueue &Q)
{
	int a;
	if (Q.front == Q.rear)
	{
		cout << "Queue is Empty" << endl;
	}
	else {
		queueptr p = Q.front->next;
		Q.front->next = p->next;
		  a = p->data;
		if (Q.rear == p)
			Q.rear = Q.front;
		free(p);
		cout << a << endl;
		coun--;
	}

}//原队列出队，需要输出出队元素和 coun--
void DequeueQ1(LinkQueue& Q)
{
	int a;
	if (Q.front == Q.rear)
	{
```

```
            cout << "Queue is Empty" << endl;

      }
      else {
            queueptr p = Q.front->next;
            Q.front->next = p->next;
            a = p->data;
            if (Q.rear == p)
                  Q.rear = Q.front;
            free(p);
      }

}//辅助队列出队
void DequeueQ1last(LinkQueue& Q)
{
      queueptr q = Q.front;
      queueptr p = Q.rear;
      while (q->next != p)
      q = q->next;
      q->next = NULL;
      Q.rear = q;
      free(p);

}//删除队尾元素
int main()
{
      int n;
      cin >> n;
      Initqueue(Q,n);
      Initqueue(Q1, n);
      string s;
      while (1)
      {
            cin >> s;
            if (s == "dequeue")
            {
if(Q.front!=Q.rear&&Q1.front!=Q1.rear&&Q.front->next->data==
Q1.front->next->data)
                  {
                        DequeueQ1(Q1);   //出队时如果原队列出队元素和辅助队
列队头相等，则辅助队列也出队，表示后面不存在这个最大值了
                  }
                  DequeueQ(Q);
```

```
}
else if (s == "enqueue")
{
	int m;
	cin >> m;

		if (coun < n)
		{
			if (Q1.front == Q1.rear)
			{
				enqueueQ1(m, Q1, n);
			}
			else {
				if (Q1.front != Q1.rear)
				{
					while (Q1.rear->data < m)
					{
						DequeueQ1last(Q1);
						if (Q1.front == Q1.rear)
							break;
					}//一直删除直到找到比它大的或为空

				}
				enqueueQ1(m, Q1, n);
			}
	}
		enqueue(m, Q, n);
}
else if (s == "max")
{
	if (Q.front == Q.rear)
		cout << "Queue is Empty" << endl;
	else cout << Q1.front->next->data << endl;
}//max值即为辅助队列队头元素
else if (s == "quit")
{
	if (Q.front != Q.rear)
	{
		queueptr t = Q.front->next;
		while (t != NULL)
		{
			cout << t->data << " ";
			t = t->next;
		}
```

```
                }
                break;
            }

        }
        return 0;
}
```

2.5.5 调试分析（遇到的问题及解决办法）

此题要设计一个数据结构使求最大值的时间复杂度最小，要设定两个队列来辅助求解，对于辅助队列的入队和出队的条件是本题的难点，当时遇到的问题主要是在队满之后再插入，虽然原队列没有入队，但是辅助队列却仍然入队，这是因为辅助队列的队列缺了判断队满这一个判断条件，导致整体答案出现问题，一般这种多次判断出入条件的，我喜欢拿纸画图并把从头到尾的过程在脑海中遍历一遍，寻找哪里出了差错，如果还找不出来就需要利用单步调试，到关键的判断条件时查看队列的值，检查问题所在。

2.5.6 总结与收获

本题是要求我们设计一种数据结构，其实按往常的做法，求 max 只需要遍历就行，基本上不存在什么思维含量，但是我们通过构造辅助队列的方法成功将这种算法的时间复杂度降低，说明即使是最常用的代码也不一定是最好最快的，在针对不同问题的要求时，以及在针对不同实际情况时，我们不能生搬硬套，要有不断精益求精不断优化的创新精神，往往能取得更好的效果。

3.实验总结

栈和队列也是属于线性数据结构，在数据结构中用途广泛且方便理解，在各种实际问题中有着广泛的应用。本次实验我总体的做题速度相比第一次实验有了显著的提升，说明我在逐渐适应这一课程的学习，但是在代码准确度上还是要有一定的缺陷，经常会出现逻辑漏洞和考虑不周全的地方，而且我每个题目改错的时间还是相对较长，调试能力还需要在不断的练习中加强。通过这五个典型的问题我对栈和队列的理解和代码的熟练度也不断提升，收获颇丰。