

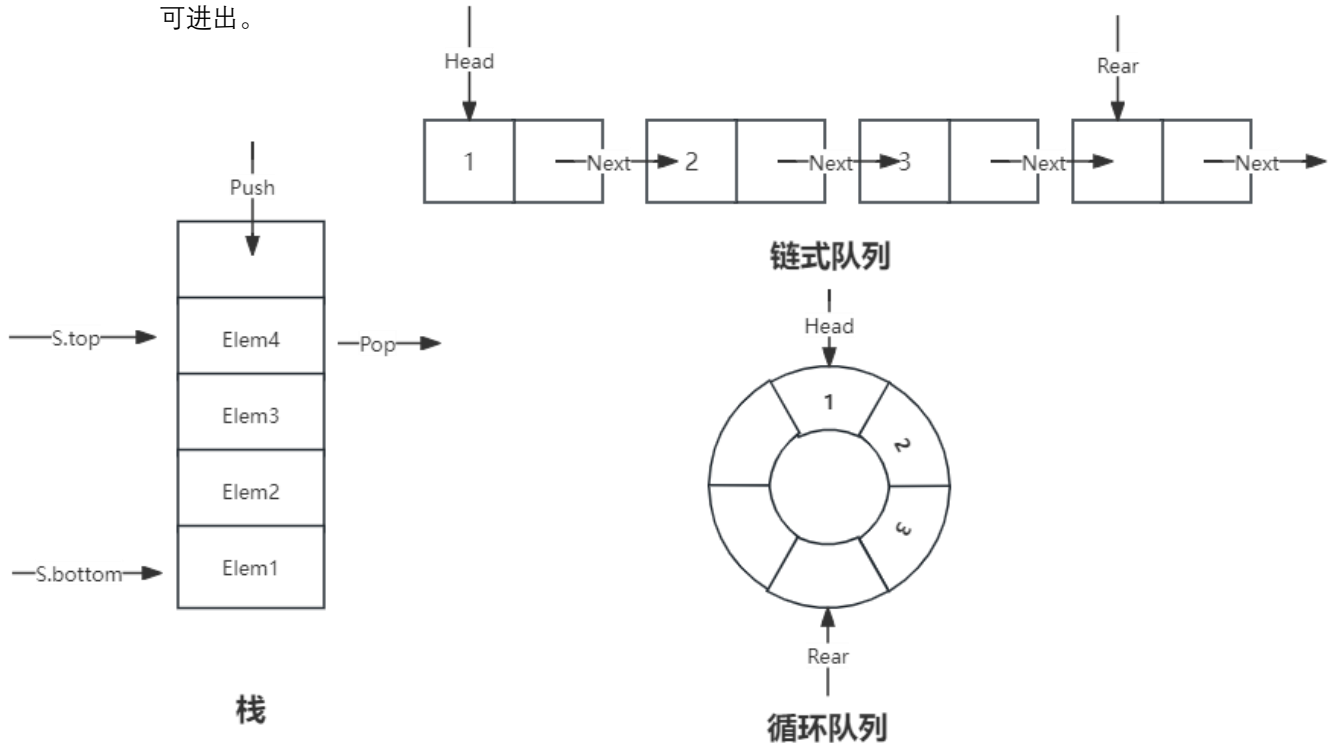
数据结构作业 HW2 实验报告

姓名：段威呈 学号：2252109 日期：2023 年 11 月 2 日

1. 涉及数据结构和相关背景

栈 队列的基本应用

- ① 栈特点：线性序列，先进后出。元素只能从栈顶进出。
- ② 队列特点：线性序列，先进先出，队尾只能进，队首只能出。存在特殊的双端队列，两端均可进出。



2. 实验内容

2.1 列车进栈

2.1.1 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。

2.1.2 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。

2.1.3 数据结构设计：栈

1) 定义栈的结构：栈顶指针、栈底指针以及栈长度

```
typedef char SElemtype;
typedef struct
{
    SElemtype* base;           //栈底指针，在栈构造之前和销毁之后，base的值为NULL
    SElemtype* top;            //栈顶指针
    int stacksize;             //栈的容量,即当前已分配的存储空间
} SqStack;
```

2) 初始化栈：动态内存申请

```
int InitStack(SqStack* S)
{
    (*S).base = (SElemtype*)malloc(STACK_INIT_SIZE * sizeof(SElemtype)); //分配存储空间
    if (!(*S).base) exit(OVERFLOW); //存储分配失败
    (*S).top = (*S).base; //当为空栈时，栈顶和栈底相同
    (*S).stacksize = STACK_INIT_SIZE; //将栈的容量设置为初始分配容量
    return 1;
}
```

3) 栈的各项功能

① 清空栈：栈顶、栈底指针对齐，长度归零

```
int ClearStack(SqStack* S)
{
    (*S).top = (*S).base; //栈顶等于栈底，且栈的容量为0
    (*S).stacksize = 0;
    return 0;
}
```

② 判断栈是否为空

```
int StackEmpty(SqStack S)
{
    if (S.base == S.top) //若栈顶与栈底相同，则说明栈为空，则返回1，否则返回0
        return 1;
    else
        return 0;
}
```

③ 元素入栈

```
int Push(SqStack& S, SElemtype e)
{
    if (S.top - S.base >= S.stacksize) //栈满，追加存储空间
    {
        S.base = (SElemtype*)realloc(S.base, (S.stacksize + STACKINCREMENT) * sizeof(SElemtype));
        if (!S.base) exit(OVERFLOW); //存储分配失败
        S.top = S.base + S.stacksize; //调整栈顶指针位置，确保其指在栈顶端
        S.stacksize += STACKINCREMENT; //为插入的位置留出空间
    }
    *S.top++ = e; //将元素插入至栈顶位置，且栈顶+1，指向下一位置
    return 1;
}
```

④ 元素出栈

```
char Pop(SqStack& S)
{
    if (S.top == S.base) return '0';
    return *--S.top; //将栈顶元素取出，且栈顶-1
}
```

⑤ 获取栈顶元素

```
SElementype GetTop(SqStack S)
{
    if (S.top == S.base) return '0';
    return *(S.top - 1); //因为栈顶指针始终指向栈顶元素的下一个位置上，所以我们要让栈顶指针减一，并用e传回栈顶元素的值
}
```

2.1.4 功能说明（函数、类）

1) 核心功能函数

函数名称：Judge()

功能：判断读入的出站序列是否满足要求

返回值：int 型，返回 1 代表满足，返回 0 代表不满足

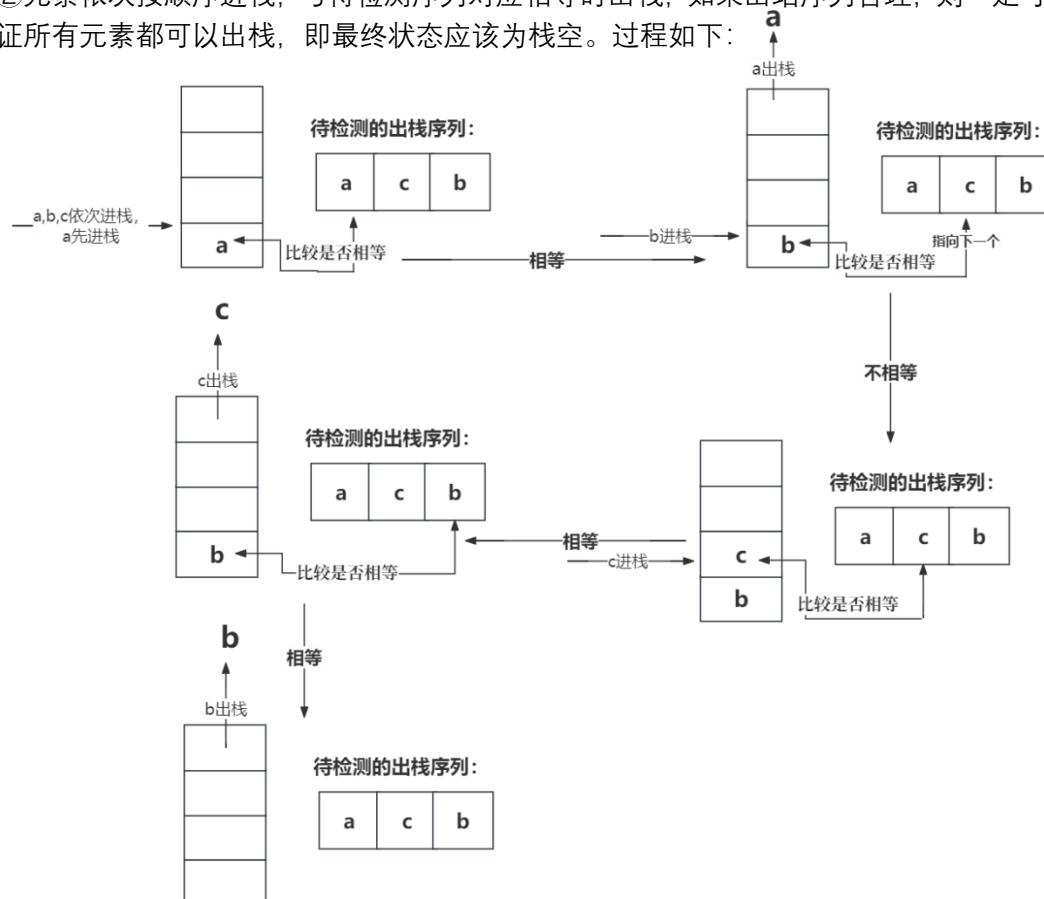
传入形参：栈 S，列车的个数 length，待检测的出站序列 c[]，列车的进站序列 b[]

```
int Judge(SqStack &S, int length, vector<char> c, vector<char> b) { //判断读入的出站序列是否合规
    //开始模拟进与出的过程
    int n = 0;
    ClearStack(&S);
    for (int r = 0; r < length; r++) { //入栈序列过程
        Push(S, b[r]); //按进站顺序依次一个个进入辅助栈
        while (GetTop(S) == c[n]) { //如果辅助栈栈顶的元素和出站顺序中现在所指向待出站的元素相同，则辅助栈中的元素出栈
            Pop(S);
            n++;
        }
    }
    if (StackEmpty(S) == 1) //如果栈空，意味着该出站顺序恰好存在与入站的列车——出栈的对应，即该出站顺序可能发生
        return 1;
    else
        return 0;
}
```

2) 算法思想：模拟进栈出栈过程

①将车站看做一个栈，列车进站可以当做元素入栈，列车出站可以当做元素出栈。”

②元素依次按顺序进栈，与待检测序列对应相等时出栈，如果出站序列合理，则一定可以保证所有元素都可以出栈，即最终状态应该为栈空。过程如下：



3) 其他功能模块

读取入站序列以及各个待检测的出栈序列：

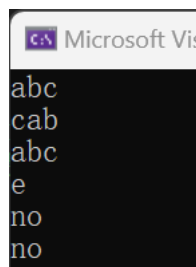
```
while((input = getchar())!='\n') { //b存放的是进站顺序
    b.push_back(input);
    ++length;
}
while ((input = getchar()) != EOF) { //在输入内容不为EOF时
    if (input != '\n')
        a.push_back(input);
    else
        ++count; //待验证序列的个数加1
}
for (int k = 0; k < count; k++) {
    c.clear(); //清空c
    for (int h = k * length; h < k*length + length; h++) {
        c.push_back(a[h]); //每次c只存一个待检查序列
    }
}
```

2.1.5 调试分析（遇到的问题解决方法）

本题可以借助输入输出重定向的方式调试

在最初的调试过程中，发现如果输入了一个错误的，则后面出现的正确的也会被判断为错误的。

在 VS 中检查出现如下情况：



Microsoft Vis
abc
cab
abc
e
no
no

经仔细检查代码发现，每次判断前要保证是空栈的初始状态，若里面还有上一次判断未清空的值，则会影响本次判断的正确性。因此需要在每次判断前清空辅助栈。

```
ClearStack(&S);
for (int r = 0; r < length; r++) { //入栈序列过程
    Push(S, b[r]); //按进站顺序依次一个个进入辅助栈
    while (GetTop(S) == c[n]) { //如果辅助栈栈顶的元素和出站顺序中现在所指向待出站的元素相同，则辅助栈中的元素出栈
        Pop(S);
        n++;
    }
}
```

2.1.6 总结和体会

本题目的模拟进栈算法复杂度为 $O(n)$ ，即只需要模拟一遍入栈和弹栈的过程即可，效率较高。此外本题的关键难点在于对数据的读取和切割，需要谨慎注意边界值的划分，否则很容易出现漏数据、读取不完整等情况。

2.2 布尔表达式

2.2.1 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)

2.2.2 基本要求

文件输入, 有若干 ($A \leq 20$) 个表达式, 其中每一行为一个表达式。表达式有 ($N \leq 100$) 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 $A \leq 5$, $N \leq 20$, 且表达式中包含 V、F、&、|

对于 40% 的数据, 有 $A \leq 10$, $N \leq 50$, 且表达式中包含 V、F、&、|、!

对于 100% 的数据, 有 $A \leq 20$, $N \leq 100$, 且表达式中包含 V、F、&、|、!、(、)

2.2.3 数据结构设计：栈

1) 栈结构设计：

①运算符栈：存放运算符及其相关信息（优先级、左右结合性）

```
struct Elem {
    char fuhao; // '(', ')', '!', '|', '&', '\n'
    int priority;
    int L_R; // 为1代表左结合, 包括 '(', ')', '|', '&'; 0代表右结合, 有 '!' ; 为2表示 '\n'
};

typedef struct
{
    Elem* base;           // 栈底指针, 在栈构造之前和销毁之后, base的值为NULL
    Elem* top;            // 栈顶指针
    int stacksize;        // 栈的容量, 即当前已分配的存储空间
} SqStack;
```

②VF 字符栈

元素都是 char 类型：

```
typedef struct
{
    char* base;           // 栈底指针, 在栈构造之前和销毁之后, base的值为NULL
    char* top;            // 栈顶指针
    int stacksize;        // 栈的容量, 即当前已分配的存储空间
} SqStack2;
```

2) 栈功能设计

两个栈功能类似, 下面以符号栈为例剖析其功能设计：

- ① 清空栈
- ② 判断栈是否为空
- ③ 弹出栈顶元素
- ④ 把元素加入栈顶
- ⑤ 获取栈 顶元素
- ⑥ 修改栈顶元素

以上①~⑤的设计与题目 1 中的完全相同

对于⑥修改栈顶元素函数 `Change_Top_xxx()`, 只需要访问栈顶指针下的一个位置的地址,

并修改即可：

```
void Change_Top_Priority(SqStack &S,int e) {  
    (*(S.top-1)).priority = e;  
}
```

2.2.4 功能说明（函数、类）

1) 表达式运算顺序规则：

①VF 字符：读入 V、F 字符依次使之入参数栈，留待计算；

②运算符号：读入符号，如果：

a)优先级高于符号栈的栈顶的符号，则直接进栈；

b)优先级低于符号栈的栈顶的符号，则栈顶元素开始进行计算；计算完成后，再比较读入符号与当前栈顶计算符的优先级，规则与前相同。

c)优先级等于符号栈栈顶的符号，若符号是左结合（&、|、）），情况同 b），即需要使栈顶元素开始计算；若符号是右结合（!），则情况同 a），直接入栈。

以下表格展示了各个运算符的优先级、左右结合情况以及运算方法：

符号形式	优先级（相对顺序）	左/右结合	运算方式
“!”	2	右	取参数栈栈顶元素，进行取反的逻辑运算，再放回参数栈栈顶
“&”	3	左	依次从参数栈的栈顶取出两个元素，进行逻辑取并运算，再放回参数栈栈顶
“ ”	4	左	依次从参数栈的栈顶取出两个元素，进行逻辑取或运算，再放回参数栈栈顶
“ (”	进入前为 1, 进入后为 2		运算符栈栈顶为“(”时，和“(”结合（即使“(”从栈中弹出）
“)”	5	左	

2) 功能函数：

①取并(&)运算：

```
void calculate_and(SqStack2* VF_stack) {  
    char VF1 = GetTop2(*VF_stack);  
    Pop2(*VF_stack);  
    char VF2 = GetTop2(*VF_stack);  
    Pop2(*VF_stack);           //依次取两个栈顶参数  
    char VF;  
    if (VF1 == 'V' && VF2 == 'V') { //都为真时才能为真  
        VF = 'V';  
    }  
    else {  
        VF = 'F';  
    }  
    Push2(*VF_stack, VF);      //运算结果返回参数栈  
}
```

② 取或(())运算

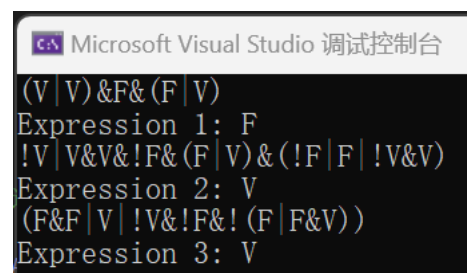
```
void calculate_or(SqStack2* VF_stack) {
    char VF1 = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    char VF2 = GetTop2(*VF_stack);
    Pop2(*VF_stack);           //依次取两个栈顶参数
    char VF;
    if (VF1 == 'F' && VF2 == 'F') { //都为假时才能为假
        VF = 'F';
    }
    else {
        VF = 'V';
    }
    Push2(*VF_stack, VF);      //运算结果返存回参数栈
}
```

③ 取否(!)运算

```
void calculate_not(SqStack2* VF_stack) {
    char VF = GetTop2(*VF_stack); //取一个栈顶参数
    Pop2(*VF_stack);
    if (VF == 'V')
        VF = 'F';
    else
        VF = 'V';                //取反操作
    Push2(*VF_stack, VF);        //运算结果返存回参数栈
}
```

2.2.5 调试分析（遇到的问题解决方法）

测试样例：



Microsoft Visual Studio 调试控制台

```
(V|V)&F&(F|V)
Expression 1: F
!V|V&V&!F&(F|V)&(!F|F|!V&V)
Expression 2: V
(F&F|V|!V&!F&!(F|F&V))
Expression 3: V
```

一开始在进行样本输入输出重定向测试的时候，发现部分！取否情况出现错误，检查后发现是在 calculate_not() 中逻辑判断上出了问题：

```
if (VF == 'V')
    VF = 'F';
if (VF == 'F')
    VF = 'V';
```

这样一套操作下来所有运算符都会被取成 V。应改成：

```
if (VF == 'V')
    VF = 'F';
else
    VF = 'V';
```

问题解决。

2.2.6 总结和体会

本题难点重在理清运算符栈的进栈的规则，即理清各个符号的运算优先级，注意左括号“（”在进栈前后会发生一个优先级变化。解决好“是否入栈”的问题，接下来就是解决“怎么算”的问题，即&，|，! 的运算规则。此外，对于括号的处理要格外小心，因为括号并不参与实际的符号运算，但会变相改变其他运算符的运算顺序，因此这里也把括号当做运算符进栈，并赋予其优先级。

2.3 最长子串

2.3.1 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。

2.3.2 基本要求

最长的字符串长度： $0 \leq n \leq 1 \times 10^5$

对于 20%的数据： $0 \leq n \leq 20$

对于 40%的数据： $0 \leq n \leq 100$

对于 60%的数据： $0 \leq n \leq 10000$

2.3.3 数据结构/算法设计

方法一：栈方法

1) 栈中的元素结构

```
struct Elem {  
    int RL; // 0左括号，1为右括号  
    int order; // 代表该括号的序号，从0开始  
};
```

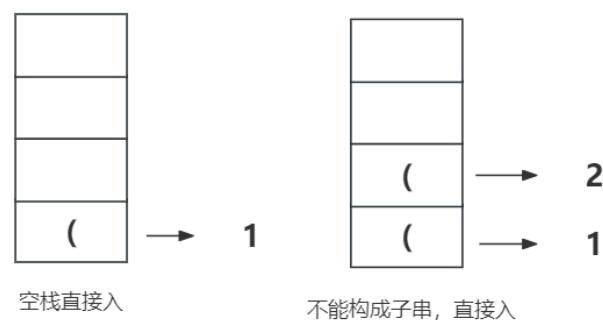
2) 栈的结构设计

(设计栈的方法同题目一与题目二)

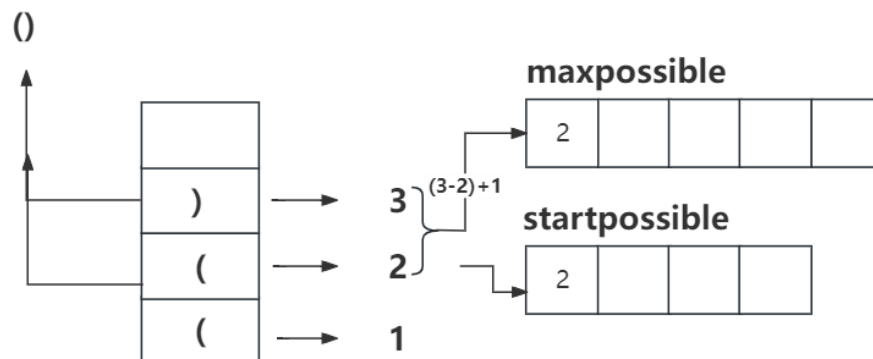
3) 实现算法

模拟入栈过程：

① 括号依次入栈，每个元素入栈的时候，记录保存自己在栈中的次序。



- ② 当入栈为右括号且栈顶元素为左括号的时候（可构成子串），则返回位置差以及初始位置到数组中存储（数组中存储的是所有子串的长度以及初始位置），同时将括号都弹出。



能构成子串，弹出并存储

- ③ 编写函数 `findmax()` 整理合并保存的子串信息，在 `maxpossible[]` 中选择最大值，同时找到 `startpossible[]` 中下标对应的数，即为最大子串的起始位置。

4) 具体实现代码

① 模拟入栈过程

```
//接下来模拟入栈过程
if (s.empty()) { //空栈直接入一个
    s.push(elem);
}
else { //若已经有元素则需要入一下并且进行比较
    if (elem.RL == 0) { //左括号直接入
        s.push(elem);
    }
    if (elem.RL == 1) {
        if (s.top().RL == 0) { //当右括号碰上左括号
            maxpossible.push_back(elem.order - s.top().order + 1);
            startpossible.push_back(s.top().order);
            s.pop();
        }
    }
}
s.push(elem);
```

② 选择函数

```
maxinformation findmax(vector<int> maxpossible, vector<int> startpossible) {
    maxinformation MAX;
    MAX.max = maxpossible[0];
    MAX.startpoint = startpossible[0];
    for (int i = 0; i <= maxpossible.size() - 1; i++) {
        if (maxpossible[i] > MAX.max) {
            MAX.max = maxpossible[i];
            MAX.startpoint = startpossible[i];
        }
    }
    return MAX;
}
```

③ 对保存的序列进行合并整理 findmax() :

该函数分为三部分：

i) 先将保存的配对序列按起始位置序号从小到大排序整理（方便后续合并）：

冒泡排序：

```
//①现将初始位置序列按照从小到大顺序排列()

for (int i = 0; i < startpossible.size() - 1; i++)
{
    for (int j = 0; j < startpossible.size() - i - 1; j++)
    {
        if (startpossible[j+1] < startpossible[j])
        {
            swap(startpossible[j+1], startpossible[j]);
            swap(maxpossible[j+1], maxpossible[j]);
        }
    }
}
```

ii) 在保存的序列中，只是把所有配对括号的信息保存下来了，但是会有如下几个问题：

a) (()) 被当做三部分保存，需要将其内部合并当做一层（内部合并）

```
//②合并(内层)
for (int i = 0; i < startpossible.size() - 1; i++) {
    if (startpossible[i] > startpossible[i+1] - maxpossible[i]) { //被包在后面的括号里面
        maxpossible.erase(maxpossible.begin() + i+1);
        startpossible.erase(startpossible.begin() + i+1);
        --i;
    }
}
```

b) 内部合并后，()(()) 仍被当做()和(())两部分，因此需要将两者合为一体保存（外部合并）

```
//③合并(外层)
for (int i = 0; i < startpossible.size() - 1; i++) {
    if (startpossible[i] == startpossible[i+1] - maxpossible[i]) {
        maxpossible[i] += maxpossible[i+1];
        maxpossible.erase(maxpossible.begin() + i + 1);
        startpossible.erase(startpossible.begin() + i + 1);
        --i;
    }
}
```

iii) 筛选出最大的 max 值：

```
//筛选最大值
for (int i = 0; i <= maxpossible.size() - 1; i++) { //遍历序列
    if (maxpossible[i] > MAX.max) { //遇到更大的则更新MAX
        MAX.max = maxpossible[i];
        MAX.startpoint = startpossible[i];
    }
}
return MAX;
```

2.2.5 调试分析（遇到的问题解决方法）

在 OJ 平台上提交检验，发现很多测试点位出现超时的情况：

```
result8.html | Submit ID: 8 | Username: 2252109 | Problem: 3

Test 1
ACCEPT
Test 2
ACCEPT
Test 3
ACCEPT
Test 4
ACCEPT
Test 5
ACCEPT
Test 6
ACCEPT
Test 7
Time Limit Exceeded
Test 8
Time Limit Exceeded
Test 9
Time Limit Exceeded
Test 10
Time Limit Exceeded
Test 11
Time Limit Exceeded
Test 12
Runtime Error
```

猜想原因：冒泡排序空间复杂度为 $O(n^2)$ ，时间效率太低了，因此这里猜想，选择使用效率更高的排序方法。

- 1) 使用快速排序法：快速排序法的排序效率为 $O(n \log_2 n) \sim O(n^2)$

```
void QuickSort(vector<int> &A, vector<int> &B, int low, int high) //快排母函数
{
    if (low < high)
    {
        int pivot = Partition1(A, B, low, high);
        QuickSort(A, B, low, pivot - 1);
        QuickSort(A, B, pivot + 1, high);
    }
}
```

提交到 OJ 平台后，发现最后一个测试点仍超时：

```
result20.html | Submit ID: 20 | Username: 2252109 | Problem: 3

Test 1
ACCEPT
Test 2
ACCEPT
Test 3
ACCEPT
Test 4
ACCEPT
Test 5
ACCEPT
Test 6
ACCEPT
Test 7
ACCEPT
Test 8
ACCEPT
Test 9
ACCEPT
Test 10
ACCEPT
Test 11
ACCEPT
Test 12
Runtime Error
```

- 2) 使用堆排序法：快速排序法的排序效率为 $O(n \log_2 n)$ 但最后一个测试点仍未通过。
(由于堆排序需要用到下一章树结构，这里不作详细展开)

于是认为，本题一旦开始使用排序，必然无法通过最后一个测试点，即借助进栈出栈模拟的方式无法将本题拿到满分。于是进一步思考，本题怎样才能避免排序呢？

想到可以直接查找右括号")"，若存在配对的左括号"("，直接存储到对应下标的数组下，即可以避免排序整理。为了实现这一想法，需要用到动态规划：

具体思想是，对一个子列，从内往外看，如对子列"(())"，相当于在内部长度为 2 的子列"()"的基础上再套上了一个"()"的“外衣”，因此长度再加 2。

因此从起始位置开始遍历序列，找到一个配对序列后，开始找他外层的“外衣”，每找到一个外衣则在原先子列基础上加 2，并新序列长度保存到对应下标的数组中。

① 将序列读入到 string s[]中：

```
string s;  
getline(cin, s);
```

② 定义一个数组 dp[]，其保存的是以第 i 个位置为终止右括号的子列的长度：

则有转移方程 $dp[i] = dp[i-1] + 2$;

③ 判断某个右括号是否存在配对的左括号：

若存在配对左括号，则从 i 位置开始，往前推 $dp[i] - 2 = dp[i-1]$ 个位置应该是左括号，即 $s[i - dp[i-1] - 1] = '('$ 。

④ 合并并列子列：在该子列左侧并列的子列的最右端括号的位置为 $i - dp[i-1] - 2$ ，因此： $dp[i] += dp[i - dp[i-1] - 2]$ 即可。

```
int longestValidParentheses(string s, int &start) {  
    vector<int> dp(s.size(), 0);  
    int maxValue = 0;  
    int prev;  
    for (int i = 1; i < s.size(); i++) {  
        if (s[i] == ')') {  
            prev = i - 1 - dp[i-1];  
            if (prev >= 0 && s[prev] == '(') {  
                dp[i] = dp[i-1] + 2;  
                if (prev - 1 >= 0)  
                    dp[i] += dp[prev - 1];  
            }  
  
            if (dp[i] > maxValue)  
                start = i - dp[i] + 1;  
            maxValue = max(maxValue, dp[i]);  
        }  
    }  
    return maxValue;  
}
```

该算法时间复杂度仅为 $O(n)$ 。

提交测试，全部样例点均通过。

2.3.6 总结和体会

该题目综合性很强，解法也很多，对于不同的任务需求需要选择不同的算法完成。用动态规划算法设计到动态规划的思想，需要推导相应的关系式，但是时间复杂度低。用栈的算法思路简单，但涉及到排序的操作，很容易超时；此外也注意到，不同排序算法时间复杂度差异也很大，冒泡排序是一种效率很低的排序算法，在实际问题解决中应尽量避免使用冒泡排序。

2.4 队列的应用

2.4.1 问题描述

输入一个 $n \times m$ 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。此算法在细胞计数上会经常用到。

2.4.2 基本要求

对于所有数据， $0 \leq n, m \leq 1000$ 。

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 $n \times m$ 的矩阵，每个元素的值为 0 或 1。输出：区域数

2.4.3 数据结构设计

1) 定义数据的结构

链式队列：



① 定义队列中的元素，存放每个节点的 x, y 坐标

```
struct ElemBox { //每个节点的元素域保存的是坐标值和父节点的索引
    int x;
    int y;
};
```

② 定义队列的结构

```
struct quene {
    elem* base; //初始化的动态分配存储空间
    int front; //头指针
    int rear; //尾指针
};
```

③ 初始化队列

```
bool InitQuene(quene &Q) { //初始化生成队列
    Q.base = (elem*)malloc (MAXQSIZE * sizeof(elem));
    Q.front = Q.rear = 0;
    return 1;
}
```

2) 队列的功能

① 在队尾添加元素

```
bool EnQuene(quene& Q, ElemBox e) {
    Q.base[Q.rear] = e;
    Q.rear += 1;
    return 1;
}
```

② 在队尾删除元素

```
bool DeQuene(queue&Q, ElemBox &e) {
    e = Q.base[Q.front];
    Q.front += 1;
    return 1;
}
```

由此可见，此删除不是真正的释放内存空间，只是将数据剔除出指针所指定的范围。



③ 判断队列是否为空，是空返回 1，反之返回 0

```
bool Is_Empty(queue& Q) { //是空返回1，不是空返回0
    if (Q.front == Q.rear) {
        return 1;
    }
    else {
        return 0;
    }
}
```

以上构造的为一个简单的线性队列，其并不能直接访问队列中的某个元素，同时队列的一端只能负责进或出一种操作。利用队列这种先进先出的特点，可以实现对区域的搜索。

2.4.4 功能说明（函数、类）

1) Input_Matrix()读入矩阵：

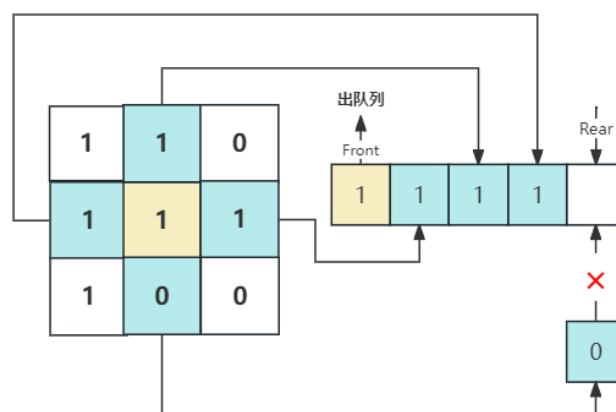
为了更好地检测边缘，需要在原矩阵的外沿加上一圈“围墙”，围墙的值为-2.

```
void Input_Matrix(int (*Matrix)[1003],int &n,int &m) {
    int k;
    cin >> n;
    cin >> m;
    for (int i = 1; i <= n + 2; i++) {
        for (int j = 1; j <= m + 2; j++) {
            if (i == 1 || i == n + 2 || j == 1 || j == m + 2) {
                Matrix[i - 1][j - 1] = -2; //边界围墙
            }
            else {
                cin >> k;
                Matrix[i - 1][j - 1] = k; //依次读入k并写到矩阵中
            }
        }
    }
}
```

2)搜索区域 Area_Search ()

功能：从起点开始，遍历一遍所有联通的 1 区域，每个被遍历过的位置都从 1 修改为-1.

① 算法设计:广度搜索算法



- a) 选择一个区域最左上端的位置为入口（二维数组从小到大遍历即可），使其入队列；
- b) 而后从队头开始依次出元素，将出列元素四周的四个格子中元素为 1 的元素入队列。
- c) 循环以上过程，

该方法可以保证所有 1 元素都可以入队列，从而实现对所有元素的遍历，遍历后该区域的 1 都被标注为-1，从而使得被统计过的区域带上了标记，不会被二次记数。

② 算法实现 Area_Search()

函数名称：Area_Search()

函数功能：判断联通（还可能是仅边缘联通），是则返回，并把所有 1 改为-1

函数形参：矩阵 Matrix[]，队列 Q，入口的初始坐标 x0,y0

函数返回值：biaozhi 是否是联通区的标志，是则返回 1，不是则返回 0

按照题目要求，这里需要对边缘进行单独记数，如果所有 1 都在边缘，则也不被记数。

```
while (1) {
    DeQueue(Q, e);
    x = e.x;
    y = e.y;
    x2 = e.x;
    y2 = e.y;
    for (int i = 1; i <= 4; i++) {
        MoveOption = i;
        x = x2;
        y = y2;
        switch (MoveOption) {
            case 1: //向左读一格
                x -= 1;
                break;
            case 2: //向右读一格
                x += 1;
                break;
            case 3: //向下读一格
                y -= 1;
                break;
            case 4: //向上读一格
                y += 1;
                break;
        }
        if (Is_Stay(Matrix, x, y) == 1) { //如果满足联通序列的部分则入队列
            e.x = x;
            e.y = y;
            EnQueue(Q, e);
            Matrix[x][y] = -1; //将1改为-1
            ++count1; //统计区域总点数
            if (Matrix[x - 1][y] == -2 || Matrix[x][y - 1] == -2 || Matrix[x + 1][y] == -2 || Matrix[x][y + 1] == -2) { //统计边缘区的点个数
                ++count2;
            }
        }
        else {
        }
    }
    if (Is_Empty(Q) == 1) { //如果队空了，说明区域内所有点都被遍历过了
        break;
    }
}
if (count1 == count2) //如果所有点都是边缘点，则认为不是联通区
    biaozi = 0;
return biaozi;
```

2.4.5 调试分析（遇到的问题和解决方法）

①一开始使用样例测试，发现出现如下的结果：



本题由于是一个图论相关问题，因此在调试过程中不妨每一步都把矩阵打印出来，从而观察为何会出现该问题。每次遍历经过如下过程：

发现在区域遍历过程中，有的点没有被遍历到。

经过问题查证，发现是在从出队的点向四周开始搜索格子前没有恢复初始化变量，导致不是所有的 1 都能入队列。规定好每次循环入队前初始化，则问题解决。

```
for (int i = 1; i <= 4; i++) {
    MoveOption = i;
    x = x2;
    y = y2;
    switch (MoveOption) {
        case 1: //向左读一格
            x -= 1;
            break;
        case 2: //向右读一格
            x += 1;
            break;
        case 3: //向下读一格
            y -= 1;
            break;
        case 4: //向上读一格
            y += 1;
            break;
    }
}
```

② 提交 OJ 平台验证后，矩阵大小超过 100×100 的测试样例无法通过。发现是矩阵开小了，但是在 main 函数中矩阵要是开到 1000×1000 会提示爆栈。但是如果把 Matrix 放在 main 函数外，设成全局静态变量，则不会报错。再次提交，所有测试点均通过。

```
static int Matrix[1003][1003];
```

2.4.6 总结和体会

本题利用队列实现广度搜索算法，是仿照迷宫问题算法进行迁移。

同时本题也可以用栈来实现。栈相比于队列还有一大优势是栈更方便回溯，即对于迷宫问题需要返回路径的问题，利用栈会更方便。

实际上利用队列也可以解决迷宫问题，只需要对每个入队的元素设为结构体，包含横纵位置坐标的同时也包括了其上一个遍历结点的序号，由于出队并不是真正的释放内存空间，因此可以根据上一个结点标号去返回路径（类似树结构的子节点与父节点，只是这里是用迭代的方式写出来的）。

2.5 队列中的最大值

2.5.1 问题描述

- (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。

2.5.2 基本要求

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)

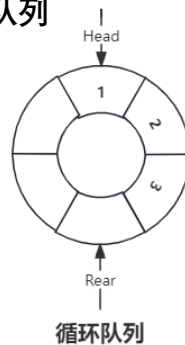
接着读入多行，每一行执行一个动作。

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

2.5.3 数据结构设计：双端环形队列



1) 定义队列结构

① 定义队列的元素

```
struct ElemBox { // 每个节点的元素域保存的是入队元素的数值
    long long value;
};
typedef ElemBox elem;
```

② 定义队列结构

```
struct quene {
    elem* base; // 初始化的动态分配存储空间
    int front; // 头指针
    int rear; // 尾指针
};
```

③ 初始化队列

```
bool InitQuene(quene& Q, int n) { // 初始化生成队列
    Q.base = (elem*)malloc(n * sizeof(elem));
    Q.front = Q.rear = 0;
    return 1;
}
```

3) 队列功能

① 在队尾添加元素

```
Q.base[Q.rear].value = e;
Q.rear = (Q.rear + 1) % n; // 注意不要忘记模n，表示循环
P.base[P.rear].value = e;
P.rear = (P.rear + 1) % n;
```

② 在队首/尾删元素

a) 队首删

```
e = Q.base[Q.front].value;
Q.front = (Q.front + 1) % n;
```

b) 队尾删

```
e = Q.base[(Q.rear-1)%n].value;  
Q.rear = (Q.rear- 1)%n;
```

③ 判断是否为空

```
bool Is_Empty(quene Q) { //是空返回1, 不是空返回0  
    if (Q.front == Q.rear) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}
```

④ 判断是否为满

```
bool Is_Full(quene Q, int n) { //是满返回1, 不是满返回0  
    if (Q.front == (Q.rear+1)%n) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}
```

2.5.4 功能说明（函数、类）

1) 算法设计思想

获取队列中最大值元素：

方法一：从队列中遍历，时间复杂度 $O(n)$ ，不建议使用，后续样例必定会超时。

方法二：构造最大值序列。

除了一个正常的保存入队元素的队列，再构建一个最大值元素一直在队首的最大值队列，则每次获得最大值的时候只需要访问该队列队首即可。

为了构建此最大值队列，可以如下操作：

一个元素入队的时候，如果队尾元素比该元素大，则先让队尾元素出队，直到队尾元素比待入队元素大或者队列为空时才允许入队。

这样可以保证越靠近队首的元素越大，队首元素为最大值。

该算法的时间复杂度为 $O(1)$

2) 算法实现代码

① 构建最大值队列以及正常存储的队列 Creat_Queue()

```
void Creat_Queue(quene &Q, quene &P, long long value, int n) { //Q为最大值单减列, P为正常存储列  
    long long e;  
    if (Is_Full(P, n) != 1) {  
        while (Q.base[(Q.rear - 1) % n].value < value) { //小于等于前才可进最大值队列, 否则删除队尾元素, 直到符合要求  
            DeQueueRear(Q, e, n);  
            if (Q.rear == Q.front) {  
                break;  
            }  
        }  
        EnQueue(Q, P, value, n, 0);  
    }  
}
```

② 获得最大值 Get_Max()

```
void Get_Max(quene Q, quene P, int n) {  
    if (Is_Empty(P) == 1) {  
        cout << "Queue is Empty" << endl;  
    } else {  
        cout << Q.base[Q.front].value << endl;  
    }  
}
```

2.5.5 调试分析（遇到的问题解决方法）

本题重点在于对循环队列指针的调控，即不要直接比较两个指针的大小关系，应该通过比较相对位置来判断队列状态。

如队列打印函数 **Print_Queue()**时，一开始设计的时候直接令 i 从 `front` 开始打印到 `rear-1` 的位置。但是发现如果队列指针出现过一次循环之后，打印开始变得混乱甚至不打印内容。这是因为在循环队列中，经过一次或多次循环后，`rear` 指针指向的位置完全有可能在 `front` 指针之前的位置。

因此要通过指针相对位置来判断状态：

如 `Q.front=Q.rear` 时候表示队列空；

`Q.front=(Q.rear+1)%n` 时候表示队列满；

[注]在循环队列中，一旦出现指针加减务必要模 n ，循环队列一直是原地的。

综上所述，`Print_Queue()`函数应如下编写：

```
void Print_Queue(queue Q, int n) {
    while (((Q.front) % n) != Q.rear) {
        cout << Q.base[Q.front].value << " ";
        Q.front = (Q.front + 1) % n;
    }
}
```

2.5.6 总结和体会

本题设计的最大值队列算法的时间复杂度仅为 $O(1)$ ，其中用到的思想是一边读一边对数据做整理，因此实际上只对所有数据进行了一次遍历。因此借助此思想，如果可以设计一个算法。在一开始数据读入过程中就将数据按照要求组织好，则可以避免二次遍历的整理。

此外，本题运用了双端循环队列的数据结构。循环队列是一种原地数据结构，相比链式队列，空间利用效率更高。其中循环队列使用时，一旦出现首/尾指针的移动，一定要模 n ；同时不要直接比较首位指针的绝对大小，要关注两者的相对位置关系。

3.实验总结

本次上机实验运用到的主要线性数据结构是栈和队列。栈的特点是元素先进后出，队列的特点是先进先出。此外为了适应更多的任务要求，队列还包括双端队列，即在首位均可以进出的队列形式（更接近顺序表的形式，但是有两个指针，可以对首尾都进行访问）。

栈数据结构的应用十分广泛。因为其可以存储数据，并且根据栈先进后出的特性，其常见的特性是把数据存进栈中后，可以自后向前依次弹出数据再做使用（数据回溯）。根据这个特性，可以通过模拟进栈的方式对某个过程序列进行分析。如**题目一列车出站**、**题目二布尔表达式**以及**题目三最长子串**，均可以借助模拟进栈的方式完成。**题目四队列的应用**也可以用栈来完成。但是栈只能解决顺序相关的问题，如果是需要从中间增删的则需要用链表或顺序表。

队列结构的特点是先进先出。队列有两种形式，链式结构以及循环结构。其中链式结构可以采用链表或者顺序表两种方式实现，如果需要对队列中元素进行直接访问，则应该通过顺序表的方式实现，如借助队列解决迷宫问题，需要用到下标回溯检索，因此需要用到顺序表。事实上，链式队列可以当做链表和顺序表的退化形式，队列可以进一步进化成双端队列。如**题目五队列最大值**则使用了双端队列。

此外关于本次实验对于算法的时间复杂度都提出了一定的要求。如在**题目三最长子串**用栈解决时需要用到排序算法，如果使用复杂度为 $O(n^2)$ 的冒泡算法则效率太低会有很多样例点无法通过，因此可以考虑效率更高的快排或者堆排序。