**作业 HW2\* 实验报告**

姓名：张嘉麟 学号：2352595 日期：2024年10月17日

# 实验报告格式要求按照模板（使用Markdown等也请保证报告内包含模板中的要素）

# 对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考vscode配色）

# 实验报告要求在文字简洁的同时将内容表示清楚

# 报告内不要大段贴代码，尽量控制在20页以内

1. **涉及数据结构和相关背景**

# 题目或实验涉及数据结构的相关背景

栈（Stack）是一种后进先出（LIFO，Last In First Out）的数据结构，具有以下特性：基本操作包括压栈（Push），将一个元素放入栈顶；弹栈（Pop），移除并返回栈顶元素；查看栈顶（Peek），返回栈顶元素但不移除它。栈的应用场景包括表达式求值，在计算机科学中，栈常用于解析和计算数学表达式（如布尔表达式），通过将操作符和操作数分别压入栈中，程序可以根据运算符的优先级依次进行计算；函数调用管理，程序中的函数调用使用栈来存储局部变量和返回地址，确保程序执行的正确性；撤销操作，在文本编辑器或图形处理软件中，栈用于记录用户的操作，以便实现撤销功能。

队列（Queue）是一种先进先出（FIFO，First In First Out）的数据结构，具有以下特性：基本操作包括入队（Enqueue），将一个元素添加到队列的尾部；出队（Dequeue），移除并返回队列的头部元素；查看队头（Front），返回队头元素但不移除它。队列的应用场景包括任务调度，在操作系统中，队列用于管理进程和任务的调度，确保先到达的任务先被执行；数据流处理，队列常用于缓冲数据流，例如网络数据包的处理和打印队列；广度优先搜索，在图算法中，队列用于实现广度优先搜索（BFS），确保按层次遍历图中的节点。1.1 栈在布尔表达式计算中的应用

在布尔表达式的计算中，栈的使用可以有效管理运算符和操作数的顺序。通过维护操作符栈和操作数栈，程序能够根据运算符的优先级，逐步计算表达式的值。尤其是在涉及括号时，栈能够确保表达式的正确解析和求值顺序。

尽管在本实验中主要使用了栈，但队列在某些相关的场景中也可能发挥作用。例如，若扩展该程序以支持多个表达式的批处理，队列可以用于存储待处理的表达式，从而实现顺序处理。

通过对栈和队列的深入理解，可以在程序设计中灵活运用这些数据结构，以解决复杂的计算问题，提高程序的效率和可维护性。

**2. 实验内容**

#### 2.1 列车进站

#### 2.1.1 问题描述

根据给定的入站序列和一系列出站序列，判断这些出站序列是否可以通过列车进站和出站的操作实现。列车可以从入口直接进入出口，也可以先进入车站再从车站进入出口。

#### 2.1.2 基本要求

输入是一个字符串，代表入站序列，以及若干字符串，代表出站序列。对于每一个出站序列，如果可以通过模拟列车进站和出站的过程实现，则输出"yes"，否则输出"no"。

#### 2.1.3 数据结构设计

为了模拟列车的进站和出站，使用了一个栈来表示车站内的列车状态。栈是一种先进后出的数据结构，非常适合用来模拟列车进站出站的行为。

stack station; // 定义车站栈

#### 2.1.4 功能说明（函数、类）

使用伪代码来描述主要的功能，包括如何模拟列车进站和出站，以及如何判断给定的出站序列是否合法。

// 检查给定的出站序列是否可以通过模拟进站和出站实现

function checkSequence(inStation, outStation):

// 初始化车站栈

station = new stack()

// 初始化入站序列的索引

index = 0

// 对于出站序列中的每一个字符

for each char c in outStation:

// 将入站序列中的列车压入车站栈，直到栈顶的列车与当前需要出站的列车匹配

while index < length(inStation) AND (station is empty OR top(station) ≠ c):

station.push(inStation[index])

index++

// 如果车站栈为空或者栈顶的列车与当前需要出站的列车不匹配，则无法实现该出站序列

if station is empty OR top(station) ≠ c THEN RETURN false

// 弹出栈顶的列车

station.pop()

// 如果车站栈不为空，则表示还有列车未出站

RETURN station is empty

#### 2.1.5 调试分析（遇到的问题和解决方法）

**栈顶列车不匹配问题**：最初的设计中，如果没有列车可以直接出站，但栈顶列车也不匹配的情况下，直接返回false。经过调整后，增加了继续压入列车直到栈顶匹配的逻辑。

**栈非空问题**：在所有列车出站后检查栈是否为空，如果不为空，则表示有列车未能出站，这也是一种不合法的状态。

#### 2.2 一元多项式的相加和相乘

#### 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

支持计算包含 V、F、&、|、! 及括号的布尔表达式。

输出格式应为“Expression X: Y”，其中 X 是表达式序号，Y 是计算结果（V 或 F）。

处理输入中的空格，并且能够识别无效字符，输出“输入错误”。

#### 2.2.3 数据结构设计

使用两个栈：

values 栈：用于存储布尔值（true 和 false）。

ops 栈：用于存储操作符（&、|、! 和括号）。

#### 2.2.4功能说明（函数、类）

int precedence(char op): 返回操作符的优先级。

bool applyOp(char op, bool a, bool b): 计算两个布尔值的双目运算结果。

bool applyNot(bool a): 计算单目运算 ! 的结果。

bool isValidChar(char c): 判断字符是否为有效的布尔表达式字符。

bool evaluateExpression(const string& expr): 评估布尔表达式，处理空格、操作符、括号及计算结果。

function evaluateExpression(expr):

initialize stacks values and ops

for each character in expr:

if character is space: continue

if character is V: push true onto values

if character is F: push false onto values

if character is '(': push '(' onto ops

if character is ')':

while top of ops is not '(':

process top of ops and update values

pop '(' from ops

if character is operator:

while top of ops has higher or equal precedence:

process top of ops and update values

push current operator onto ops

process remaining operators in ops

return final value in values

function process operator(op):

if op is '!':

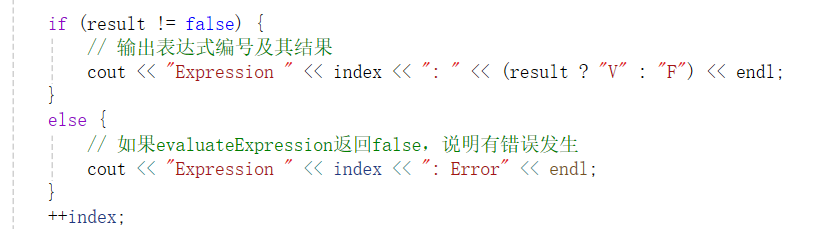
pop value from values and push applyNot(value)

else:

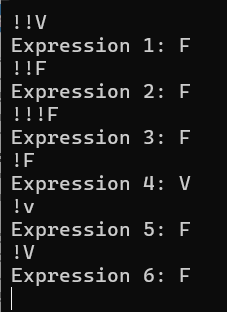
pop two values from values and push applyOp(op, a, b)

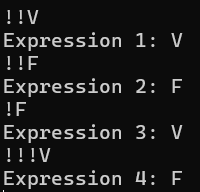
#### 2.2.5 调试分析（遇到的问题和解决方法）

1.If语句的错误使用：此时result只有为true的时候才可以进去第一句话，所以只能正常输出V而无法输出F。后奖if语句判断去掉。



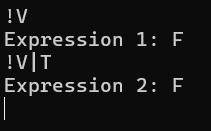
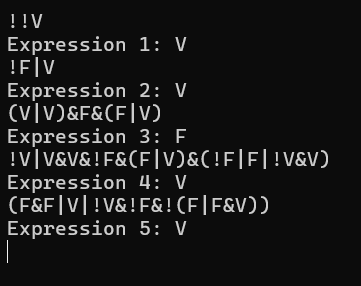
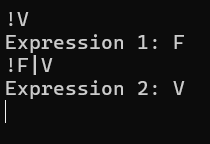
**2.优先级处理**：初始版本中未正确处理 ! 的优先级，导致错误的计算结果。通过重新设计栈的处理逻辑，确保在遇到 ! 时，先处理其后续的值。

3.连续 ! 的处理：在处理多个感叹号时，前期的设计没有考虑连续感叹号的情况。后续通过引入额外的栈来处理，只在后续操作符为括号或感叹号时才继续入栈。



更改前 更改后

4.！的优先级和出栈顺序：第三条更改后出现新的问题，即！的出栈顺序在&和|之后（如图所示），更改后正确。



更改前 更改后

**5.无效输入**：原有代码未能有效处理无效字符。在调试时添加了有效字符的检查，确保程序在遇到无效输入时输出“输入错误”。

**6.空格处理**：最初的实现未能妥善处理输入中的空格，后来通过简单的 if 判断跳过空格，确保表达式能正确解析。下图为调试空格的控制台：



#### 2.3 最长子串

#### 2.3.1 问题描述

给定一个只包含字符 '(' 和 ')' 的字符串，要求计算出最长的有效括号子串的长度及其起始位置。若存在多个最长子串，返回第一个的起始位置。如果字符串为空，则返回长度为0和起始位置0。

#### 2.3.2 基本要求

输入为一个长度为n的字符串，字符仅为'('和')'，且0 ≤ n ≤ 100,000。

输出为最长有效括号子串的长度及其起始位置。应使用栈数据结构来辅助解决问题。

#### 2.3.3 数据结构设计

使用一个栈 st 来存储左括号的索引。

初始化栈时，向栈中压入一个-1，表示假设字符串开始前有一个左括号。这有助于在没有有效左括号时进行长度计算。

#### 2.3.4 功能说明（函数、类）

main()

输入字符串 s

如果 s 的首尾为引号

去除引号

初始化栈 st

压入 -1 到栈 st // 假设字符串前有一个左括号

初始化 maxLength 为 0 // 最大有效子串长度

初始化 startIndex 为 0 // 最大有效子串起始位置

对于每个字符 s[i] 在 s 中

如果 s[i] 是 '(' // 遇到左括号

将索引 i 压入栈 st

否则（s[i] 是 ')') // 遇到右括号

弹出栈顶元素 // 试图匹配最近的左括号

如果 栈 st 为空 // 栈空表示没有匹配的左括号

压入当前右括号的索引 i 到栈 st

否则

currentLength = i - st.top() // 计算当前有效子串的长度

如果 currentLength > maxLength // 更新最大长度

maxLength = currentLength

startIndex = st.top() + 1 // 更新起始位置

输出 maxLength 和 startIndex // 输出最长有效括号子串的长度和起始位置

**2.3.5 调试分析（遇到的问题和解决方法）**

**问题1**：栈为空的处理。遇到右括号时，若栈为空，表示没有匹配的左括号。为了解决这个问题，当栈为空时，需要将当前右括号的索引压入栈中，以作为新的起始点。

**问题2**：有效子串长度的计算。初始化栈时，需要压入-1以处理有效括号串的长度计算。如果栈中有元素，计算当前有效子串长度时要减去栈顶元素的索引，以得到当前有效子串的起始位置。

**问题3**：在多次测试中，需要确保程序能够正确处理所有边界情况，如空字符串、仅包含左括号或右括号的字符串等。通过多样化的测试用例，确保算法的稳定性和准确性。

**2.4 列队的应用（数组区域数量）**

**2.4.1 问题描述**

给定一个 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（上下左右）联通的 1 算作一个区域，但如果 1 仅在矩阵边缘联通，则该区域不算作有效区域。要求计算有效区域的数量。

#### 2.4.2 基本要求

输入：第1行有两个正整数 n 和 m，表示矩阵的行数和列数。接下来有 n 行，每行 m 个整数，表示矩阵的元素（0 或 1）。

输出：输出一个整数，表示有效区域的数量。

**2.4.3 数据结构设计**

// 定义常量 MAXN 作为矩阵的最大尺寸

常量 MAXN = 1000

// 定义存储矩阵的二维数组 grid 和标记访问状态的 visited 数组

矩阵 grid[MAXN][MAXN] // 存储输入的矩阵，grid[i][j] 为 0 或 1

矩阵 visited[MAXN][MAXN] // 访问标记数组，visited[i][j] 为 true 表示该点已访问过

// 定义四个方向的数组 dx 和 dy，分别表示上下左右的偏移量

数组 dx = [-1, 1, 0, 0] // 行方向变化：上(-1), 下(+1)

数组 dy = [0, 0, -1, 1] // 列方向变化：左(-1), 右(+1)

**2.4.4功能说明（函数、类）**

1.isValid(x, y, n, m)

功能：检查坐标 (x, y) 是否在矩阵范围内，且该点是未访问的 1。

输入：坐标 x 和 y，矩阵大小 n 和 m。

输出：返回布尔值，true 表示该点是有效的 1，否则返回 false。

// 函数：isValid

输入：坐标 x, y, 矩阵大小 n, m

返回：布尔值，表示该点是否为有效的 1 且未访问

函数 isValid(x, y, n, m):

如果 x 或 y 超出矩阵边界:

返回 false

如果 grid[x][y] == 1 且 visited[x][y] == false:

返回 true

否则:

返回 false

2.bfs(x, y, n, m)

功能：从坐标 (x, y) 开始执行 BFS，遍历该区域所有相连的 1，并检查是否包含非边缘的元素。

输入：起始坐标 (x, y)，矩阵大小 n 和 m。

输出：返回布尔值，true 表示该区域包含非边缘的 1，否则返回 false。

// 函数：bfs

输入：坐标 x, y, 矩阵大小 n, m

返回：布尔值，表示该区域是否为有效区域（包含非边缘元素）

函数 bfs(x, y, n, m):

初始化队列 q, 将 (x, y) 入队

标记 visited[x][y] 为已访问

标记 hasInner = false // 标记该区域是否有非边缘的1

当队列 q 非空时:

当前坐标 = q.front() 出队

如果 当前坐标不在矩阵边缘:

标记 hasInner = true // 该区域包含非边缘的1

对于 四个方向 i (上下左右):

计算 newX = 当前坐标的 x + dx[i]

计算 newY = 当前坐标的 y + dy[i]

如果 isValid(newX, newY, n, m):

标记 visited[newX][newY] 为已访问

将 (newX, newY) 入队

返回 hasInner // 返回是否为有效区域

3.main()

功能：初始化矩阵，遍历矩阵中的每个 1，执行 BFS 来判断每个区域是否有效，最终输出有效区域的数量。

输入：矩阵大小 n 和 m 及矩阵元素。

输出：有效区域的数量。

// 函数：main

输入：矩阵大小 n, m

输出：有效区域数量

函数 main():

输入 n 和 m

初始化 grid 和 visited 数组

// 输入矩阵的每个元素

对于 i 从 0 到 n-1:

对于 j 从 0 到 m-1:

输入 grid[i][j]

初始化区域计数器 regionCount = 0

// 遍历矩阵中的每个元素

对于 i 从 0 到 n-1:

对于 j 从 0 到 m-1:

如果 grid[i][j] == 1 且 visited[i][j] == false:

如果 bfs(i, j, n, m):

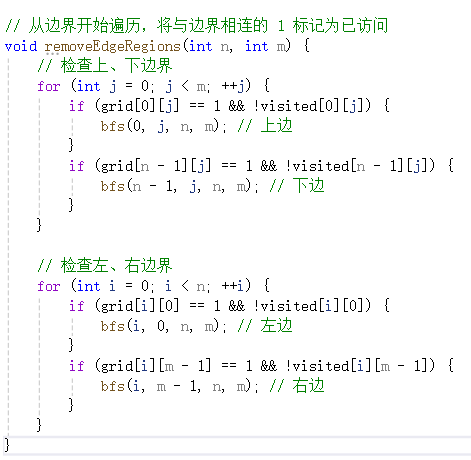
regionCount += 1 // 只有包含非边缘元素时，才计为有效区域

输出 regionCount

#### 2.4.5调试分析（遇到的问题和解决方法）

1.边缘过滤问题

问题：初始版本直接删除边缘连通的 1，导致一些区域被拆分（因为去掉边缘可能导致本来连在一起的两部分断开了），错误计数。



解决：通过先完整搜索区域，再检查是否包含非边缘元素来确保计数正确。

2.重复访问问题

问题：早期版本中，未对访问过的 1 进行标记，导致重复计数。

解决：在 BFS 搜索过程中，及时标记已访问的节点，避免重复遍历。

3.性能优化问题

问题：当矩阵规模较大（如 1000x1000），算法性能下降。

解决：通过优化 BFS 队列操作，保持算法时间复杂度为 O(n\*m)，确保大规模数据下的性能。

**2.1.6 总结和体会**

本题的难点在于如何正确识别连通区域，并过滤掉仅在边缘的区域。在初期设计中，错误地去除边缘导致区域被拆分，但通过改进算法，先遍历完整区域再判断有效性，问题得到解决。

**收获**：通过此题学会了如何使用广度优先搜索（BFS）来解决矩阵类问题，并理解了处理边界条件的重要性。

**难点**：难点在于如何有效处理边缘条件，同时保证算法的时间复杂度在大规模数据下也能满足需求。

#### 2.5 列队中的最大值

#### 2.5.1 问题描述

给定一个只包含字符 '(' 和 ')' 的字符串，计算最长的有效括号子串的长度及其起始位置。如果存在多个最长的有效子串，则输出起始位置最小的那个子串的相关信息。

#### 2.5.2 基本要求

字符串长度：0≤n≤1×1050≤*n*≤1×105。输出最长有效子串的长度及其起始位置。若字符串为空或没有有效的括号子串，则输出长度为 0，起始位置为 0。

#### 2.5.3 数据结构设计

为了有效地解决这个问题，我们可以使用栈来跟踪左括号的位置，并根据右括号来确定有效子串的长度和起始位置。

// 定义栈用于存储左括号的位置

Stack<Integer> stack;

#### 2.5.4 功能说明（函数、类）

使用栈来存储左括号的位置，并在遇到右括号时，根据栈的状态来更新最长有效子串的长度和起始位置。

function findLongestValidSubstring(inputString):

// 初始化栈并预设一个基准点

stack.push(-1)

maxLength = 0

startIndex = 0

// 遍历字符串

for i from 0 to length(inputString) - 1 do:

char = inputString[i]

if char == '(':

// 遇到左括号，将其索引压入栈中

stack.push(i)

else if char == ')':

// 遇到右括号，弹出栈顶元素

stack.pop()

if stack.isEmpty():

// 栈空，意味着没有匹配的左括号，设置新的起点

stack.push(i)

else:

// 计算当前有效子串的长度

currentLength = i - stack.peek()

if currentLength > maxLength:

maxLength = currentLength

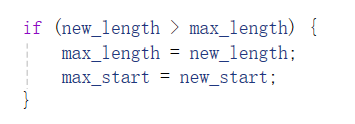
startIndex = stack.peek() + 1

return maxLength, startIndex

#### 2.5.5 调试分析（遇到的问题和解决方法）

在调试过程中，我们遇到了以下几个问题：

1.刚开始的成绩一直为25分，当时程序是后入栈先出栈的顺序。Debug时发现判断函数写的是“>”，事实上是不对的。由于后入栈的先出栈，由于题目中有说如果存在两个长度相等的取前面的，对应到程序里面就是只要前面的大于或等于后面的数就更新。所以应改为“≥”。



**2.输入处理问题**：输入字符串可能包含首尾引号，需要在处理之前去除这些引号。

**3.边界情况处理**：当字符串为空或不含有效括号时，需要正确处理这些边界情况。

**4.栈操作错误**：在某些情况下，栈操作可能导致错误的结果，例如在没有匹配的左括号时，栈可能为空，此时需要特别处理。

针对这些问题，我们采取了以下措施：

在处理输入字符串时，检查并去除首尾的引号。

对于边界情况，初始化栈时插入一个虚拟的基准点（如 -1），以方便后续的计算。

在栈操作时，增加了对栈是否为空的检查，避免了空栈操作引发的错误。

1. **实验总结**

在本次实验中，通过实现了不同的数据结构，如栈、队列，以及栈在布尔表达式求值中的具体应用，加深了我对这些数据结构及其在实际问题中的使用场景的理解。在整个实验过程中，我收获了许多宝贵的经验，尤其是在问题的分析、调试、优化过程中。

1. 数据结构的重要性

栈与队列是计算机科学中最为基础的两种数据结构，它们分别适用于后进先出（LIFO）和先进先出（FIFO）的场景。实验中的列车进站问题、布尔表达式计算问题都可以通过栈进行有效地解决，而在广度优先搜索中，队列则是重要的数据结构。在处理复杂问题时，选择合适的数据结构能显著提高解决问题的效率和代码的可维护性。

2. 栈在布尔表达式计算中的应用

在布尔表达式的求值过程中，栈用于管理操作符和操作数的顺序。通过设计两个栈分别存储操作数和操作符，能高效地处理括号和不同优先级的运算符。尤其在处理逻辑非（!）的优先级和连续取反时，我通过栈来延迟执行取反操作，确保逻辑的正确性。在实现的过程中，不仅解决了优先级的问题，还有效地处理了连续符号及无效输入。

3. 解决调试问题

本次实验涉及多个调试环节，其中，栈在布尔表达式求值中的优先级问题和连续符号处理问题，给我带来了不少挑战。通过多次测试和调试，我逐步优化了代码，使其能够正确处理复杂输入及边界情况。在最长有效括号子串问题中，使用栈来辅助计算有效子串的长度和起始位置，成功解决了括号匹配的问题。

4. 实验的收获与思考

本次实验让我更加深入地理解了数据结构的运用。对于每个问题，从分析、设计到调试和优化，我都在不断实践数据结构的理论知识，并将其应用到实际问题中。通过解决实际问题，我逐步提高了代码实现的能力和调试的技巧。同时，在编写代码时，尽量减少代码的复杂度，提升可读性和扩展性，这对今后的编程工作具有深远的意义。