【华为OD机考 统一考试机试C卷】最长合法表达式（C++ Java Python javaScript）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题。 可以先继续刷B卷， C卷和D卷的题目会放在现在大家购买的专栏内，不需要重新购买，请大家放心。

专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）

华为OD面试真题精选：华为OD面试真题精选

在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述：最长合法表达式（本题分值200）

提取字符串中的最长合法简单数学表达式字符串长度最长的，并计算表达式的值。如果没有返回 0

简单数学表达式只能包含以下内容

0-9 数字，符号±*

说明:

1.所有数字，计算结果都不超过 long

2.如果有多个长度一样的，请返回第一个表达式的结果

3.数学表达式，必须是最长的，合法的

4.操作符不能连续出现，如 ±-+1 是不合法的

输入描述

字符串

输出描述

表达式值

示例一

输入

1-2abcd

输出

-1

输入描述

字符串

输出描述

表达式值

用例

输入

```
1-2abcd
```

输出

```
-1
```

解题思路

1. 首先，我们需要从输入的字符串中提取出所有的合法数学表达式。在这里，一个合法的数学表达式是指由数字和操作符（+、-、*）组成的字符串，且操作符不能连续出现。我们通过遍历输入字符串的每个字符，并使用一个变量 `start` 来记录当前表达式的开始索引，来实现这一步。当我们遇到一个数字字符时，如果 `start` 为-1（表示我们还没有开始记录一个新的表达式），我们就将 `start` 设置为当前索引；当我们遇到一个操作符字符时，如果前一个字符不是操作符，我们就将当前索引加1；否则，我们就提取从 `start` 到当前索引的子字符串作为一个表达式，并将 `start` 重置为-1。如果我们遇到一个既不是数字也不是操作符的字符，并且 `start` 不为-1，我们也会提取从 `start` 到当前索引的子字符串作为一个表达式，并将 `start` 重置为-1。最后，如果 `start` 不为-1，我们会提取从 `start` 到字符串末尾的子字符串作为一个表达式。
2. 然后，我们需要对提取出的所有表达式按长度进行排序，并选择长度最长的表达式。我们使用Python的 `sorted` 函数来实现这一步，其中 `key=len` 表示按长度排序，`reverse=True` 表示按降序排序。
3. 接下来，我们需要计算选出的表达式的结果。我们首先使用正则表达式来分割表达式，得到一个由数字和操作符组成的列表。然后，我们遍历这个列表，先进行所有的乘法操作，然后进行所有的加法和减法操作。在这个过程中，我们使用一个变量 `result` 来记录当前的计算结果。
4. 最后，我们打印出计算结果。如果没有提取出任何表达式，我们就打印0。

C++

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <cctype>
using namespace std;
// 提取合法表达式的函数
vector<string> extractExpressions(const string &line) {
    vector<string> expressions;
    int start = -1;

    for (size_t i = 0; i < line.length(); ++i) {
        char cur = line[i];

        // 如果当前字符是数字
        if (isdigit(cur)) {
            // 如果开始索引为-1，设置开始索引为当前索引
            if (start == -1) {
                start = i;
            }
        } else if (start != -1 && (cur == '+' || cur == '-' || cur == '*')) {
            // 如果当前字符是操作符，并且前一个字符不是操作符
            // 则将当前索引加1
            if (i == 0 || !(line[i - 1] == '+' || line[i - 1] == '-' || line[i - 1] == '*')) {
                ++i;
            } else {
                // 否则，提取从开始索引到当前索引的子字符串作为一个表达式
                // 并将开始索引重置为-1
                expressions.push_back(line.substr(start, i - start));
                start = -1;
            }
        } else {
            // 如果当前字符既不是数字也不是操作符
            // 并且开始索引不为-1，则提取从开始索引到当前索引的子字符串作为一个表达式
            // 并将开始索引重置为-1
            if (start != -1) {
                expressions.push_back(line.substr(start, i - start));
                start = -1;
            }
        }
```

```
    }

    // 如果开始索引不为-1，则提取从开始索引到字符串末尾的子字符串作为一个表达式
    if (start != -1) {
        expressions.push_back(line.substr(start));
    }

    // 返回提取的表达式列表
    return expressions;
}

// 计算表达式结果的函数
int calc(const string &str) {
    vector<string> tokens;
    string number;

    for (char ch : str) {
        // 如果当前字符是数字，则将其添加到当前数字的末尾
        if (isdigit(ch)) {
            number += ch;
        } else {
            // 否则，将当前数字和当前操作符添加到元素列表中
            // 并开始构建新的数字
            tokens.push_back(number);
            tokens.push_back(string(1, ch));
            number.clear();
        }
    }
    // 将最后一个数字添加到元素列表中
    tokens.push_back(number);

    // 遍历元素列表，先进行所有的乘法操作
    for (size_t i = 0; i < tokens.size(); ++i) {
        if (tokens[i] == "*") {
            int result = stoi(tokens[i - 1]) * stoi(tokens[i + 1]);
            tokens[i - 1] = to_string(result);
            tokens.erase(tokens.begin() + i, tokens.begin() + i + 2);
            --i;
        }
    }
```

```
    // 初始化结果为元素列表的第一个数字
    int result = stoi(tokens[0]);
    // 遍历元素列表，进行所有的加法和减法操作
    for (size_t i = 1; i < tokens.size(); i += 2) {
        if (tokens[i] == "+") {
            result += stoi(tokens[i + 1]);
        } else {
            result -= stoi(tokens[i + 1]);
        }
    }

    // 返回计算结果
    return result;
}

int main() {
    // 创建一个字符串来处理输入
    string line;
    // 读取一行输入
    getline(cin, line);

    // 调用extractExpressions函数提取输入中的合法表达式
    vector<string> expressions = extractExpressions(line);

    // 使用C++的sort函数对表达式按长度进行排序
    sort(expressions.begin(), expressions.end(), [](const string &s1, const string &s2) {
        return s2.length() < s1.length();
    });

    // 如果表达式列表不为空，则计算并打印最长表达式的结果
    // 否则，打印0
    if (!expressions.empty()) {
        cout << calc(expressions[0]) << endl;
    } else {
        cout << 0 << endl;
    }

    return 0;
}
```

Java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        // 创建一个Scanner对象来处理输入
        Scanner in = new Scanner(System.in);
        // 读取一行输入
        String line = in.nextLine();
        // 关闭Scanner对象
        in.close();

        // 调用extractExpressions方法提取输入中的合法表达式
        List<String> expressions = extractExpressions(line);

        // 使用Java 8的Lambda表达式对表达式按长度进行排序
        expressions.sort((s1, s2) -> Integer.compare(s2.length(), s1.length()));

        // 如果表达式列表不为空，则计算并打印最长表达式的结果
        // 否则，打印0
        if (!expressions.isEmpty()) {
            System.out.println(calc(expressions.get(0)));
        } else {
            System.out.println(0);
        }
    }

    // 提取合法表达式的方法
    public static List<String> extractExpressions(String line) {
        // 创建一个列表来存储提取的表达式
        List<String> expressions = new ArrayList<>();
        // 初始化开始索引为-1
        int start = -1;

        // 遍历输入字符串的每个字符
        for (int i = 0; i < line.length(); i++) {
            char cur = line.charAt(i);

```

```
            // 如果当前字符是数字
            if (Character.isDigit(cur)) {
                // 如果开始索引为-1，设置开始索引为当前索引
                if (start == -1) {
                    start = i;
                }
            } else if (start != -1 && "+-*".contains(String.valueOf(cur))) {
                // 如果当前字符是操作符，并且前一个字符不是操作符
                // 则将当前索引加1
                if (!"+-*".contains(String.valueOf(line.charAt(i - 1)))) {
                    i++;
                } else {
                    // 否则，提取从开始索引到当前索引的子字符串作为一个表达式
                    // 并将开始索引重置为-1
                    expressions.add(line.substring(start, i));
                    start = -1;
                }
            } else {
                // 如果当前字符既不是数字也不是操作符
                // 并且开始索引不为-1，则提取从开始索引到当前索引的子字符串作为一个表达式
                // 并将开始索引重置为-1
                if (start != -1) {
                    expressions.add(line.substring(start, i));
                    start = -1;
                }
            }
        }

        // 如果开始索引不为-1，则提取从开始索引到字符串末尾的子字符串作为一个表达式
        if (start != -1) {
            expressions.add(line.substring(start));
        }

        // 返回提取的表达式列表
        return expressions;
    }

    // 计算表达式结果的方法
    public static int calc(String str) {
```

```
        // 创建一个列表来存储表达式的每个元素（数字或操作符）
        List<String> tokens = new ArrayList<>();
        // 创建一个StringBuilder对象来构建数字
        StringBuilder sb = new StringBuilder();

        // 遍历表达式的每个字符
        for (char ch : str.toCharArray()) {
            // 如果当前字符是数字，则将其添加到当前数字的末尾
            if (Character.isDigit(ch)) {
                sb.append(ch);
            } else {
                // 否则，将当前数字和当前操作符添加到元素列表中
                // 并开始构建新的数字
                tokens.add(sb.toString());
                tokens.add(String.valueOf(ch));
                sb.setLength(0);
            }
        }
        // 将最后一个数字添加到元素列表中
        tokens.add(sb.toString());

        // 遍历元素列表，先进行所有的乘法操作
        for (int i = 0; i < tokens.size(); i++) {
            if ("*".equals(tokens.get(i))) {
                int result = Integer.parseInt(tokens.get(i - 1)) * Integer.parseInt(tokens.get(i + 1));
                tokens.set(i - 1, String.valueOf(result));
                tokens.remove(i);
                tokens.remove(i);
                i--;
            }
        }

        // 初始化结果为元素列表的第一个数字
        int result = Integer.parseInt(tokens.get(0));
        // 遍历元素列表，进行所有的加法和减法操作
        for (int i = 1; i < tokens.size(); i += 2) {
            if ("+".equals(tokens.get(i))) {
                result += Integer.parseInt(tokens.get(i + 1));
            } else {
                result -= Integer.parseInt(tokens.get(i + 1));
```

```
            }
        }

        // 返回计算结果
        return result;
    }
}
```

javaScript

```javascript
const readline = require('readline');

// 创建一个readline.Interface实例来处理输入
const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.on('line', (line) => {
    // 调用extractExpressions函数提取输入中的合法表达式
    let expressions = extractExpressions(line);

    // 使用JavaScript的sort函数对表达式按长度进行排序
    expressions.sort((s1, s2) => s2.length - s1.length);

    // 如果表达式列表不为空，则计算并打印最长表达式的结果
    // 否则，打印0
    if (expressions.length > 0) {
        console.log(calc(expressions[0]));
    } else {
        console.log(0);
    }

    rl.close();
});

// 提取合法表达式的函数
function extractExpressions(line) {
    let expressions = [];
    let start = -1;
```

```
    for (let i = 0; i < line.length; i++) {
        let cur = line.charAt(i);

        // 如果当前字符是数字
        if (!isNaN(cur)) {
            // 如果开始索引为-1，设置开始索引为当前索引
            if (start === -1) {
                start = i;
            }
        } else if (start !== -1 && "+-*".includes(cur)) {
            // 如果当前字符是操作符，并且前一个字符不是操作符
            // 则将当前索引加1
            if (!"+-*".includes(line.charAt(i - 1))) {
                i++;
            } else {
                // 否则，提取从开始索引到当前索引的子字符串作为一个表达式
                // 并将开始索引重置为-1
                expressions.push(line.substring(start, i));
                start = -1;
            }
        } else {
            // 如果当前字符既不是数字也不是操作符
            // 并且开始索引不为-1，则提取从开始索引到当前索引的子字符串作为一个表达式
            // 并将开始索引重置为-1
            if (start !== -1) {
                expressions.push(line.substring(start, i));
                start = -1;
            }
        }
    }

    // 如果开始索引不为-1，则提取从开始索引到字符串末尾的子字符串作为一个表达式
    if (start !== -1) {
        expressions.push(line.substring(start));
    }

    // 返回提取的表达式列表
    return expressions;
}
```

```
// 计算表达式结果的函数
function calc(str) {
    let tokens = [];
    let number = '';

    for (let ch of str) {
        // 如果当前字符是数字，则将其添加到当前数字的末尾
        if (!isNaN(ch)) {
            number += ch;
        } else {
            // 否则，将当前数字和当前操作符添加到元素列表中
            // 并开始构建新的数字
            tokens.push(number);
            tokens.push(ch);
            number = '';
        }
    }
    // 将最后一个数字添加到元素列表中
    tokens.push(number);

    // 遍历元素列表，先进行所有的乘法操作
    for (let i = 0; i < tokens.length; i++) {
        if (tokens[i] === "*") {
            let result = parseInt(tokens[i - 1]) * parseInt(tokens[i + 1]);
            tokens[i - 1] = String(result);
            tokens.splice(i, 2);
            i--;
        }
    }

    // 初始化结果为元素列表的第一个数字
    let result = parseInt(tokens[0]);
    // 遍历元素列表，进行所有的加法和减法操作
    for (let i = 1; i < tokens.length; i += 2) {
        if (tokens[i] === "+") {
            result += parseInt(tokens[i + 1]);
        } else {
            result -= parseInt(tokens[i + 1]);
        }

```

```
    }

    // 返回计算结果
    return result;
}
```

Python

```python
# 导入需要的模块
import re

# 提取合法表达式的函数
def extract_expressions(line):
    # 创建一个列表来存储提取的表达式
    expressions = []
    # 初始化开始索引为-1
    start = -1

    # 遍历输入字符串的每个字符
    for i in range(len(line)):
        cur = line[i]

        # 如果当前字符是数字
        if cur.isdigit():
            # 如果开始索引为-1，设置开始索引为当前索引
            if start == -1:
                start = i
        elif start != -1 and cur in "+-*":
            # 如果当前字符是操作符，并且前一个字符不是操作符
            # 则将当前索引加1
            if line[i - 1] not in "+-*":
                i += 1
            else:
                # 否则，提取从开始索引到当前索引的子字符串作为一个表达式
                # 并将开始索引重置为-1
                expressions.append(line[start:i])
                start = -1
        else:
            # 如果当前字符既不是数字也不是操作符
            # 并且开始索引不为-1，则提取从开始索引到当前索引的子字符串作为一个表达式
```

```
            # 并将开始索引重置为-1
            if start != -1:
                expressions.append(line[start:i])
                start = -1

    # 如果开始索引不为-1，则提取从开始索引到字符串末尾的子字符串作为一个表达式
    if start != -1:
        expressions.append(line[start:])

    # 返回提取的表达式列表
    return expressions

# 计算表达式结果的函数
def calc(expression):
    # 使用正则表达式分割字符串，得到数字和操作符
    tokens = re.split('([-+*])', expression)
    tokens = [token for token in tokens if token]

    # 遍历元素列表，先进行所有的乘法操作
    while '*' in tokens:
        index = tokens.index('*')
        result = int(tokens[index - 1]) * int(tokens[index + 1])
        tokens = tokens[:index - 1] + [str(result)] + tokens[index + 2:]

    # 初始化结果为元素列表的第一个数字
    result = int(tokens[0])
    # 遍历元素列表，进行所有的加法和减法操作
    for i in range(1, len(tokens), 2):
        if tokens[i] == '+':
            result += int(tokens[i + 1])
        else:
            result -= int(tokens[i + 1])

    # 返回计算结果
    return result

# 读取一行输入
line = input()

# 调用extract_expressions函数提取输入中的合法表达式
```

```
expressions = extract_expressions(line)

# 使用Python的sorted函数对表达式按长度进行排序
expressions = sorted(expressions, key=len, reverse=True)

# 如果表达式列表不为空，则计算并打印最长表达式的结果
# 否则，打印0
if expressions:
    print(calc(expressions[0]))
else:
    print(0)
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师