C++方向编程题答案

第六周

day36

1、题目ID： 24505-Rational Arithmetic

https://www.nowcoder.com/questionTerminal/b388bdee5e3e4b1c86a79ad1877a3aa4

【题目翻译】：

24505-有理数运算

实现对两个有理数的基本运算，包括加、减、乘、除。

输入描述:

每个输入文件只包含一个测试用例，测试用例会给出一行数据，格式为“a1/b1 a2/b2”

分子分母的范围都在长整型的范围内，如果数字为负，则符号只会出现在分子的前面。分母一定是非零数。

输出描述:

针对每个测试用例，都输出四行，分别是这两个有理数的和、差、积和商，格式为“数1 操作符 数2 = 结果”。注意，所有的有理数都将遵循一个简单形式“k a/b”，其中k是整数部分，a/b是最简分数形式，如果该数为负数，则必须用括号包起来。如果除法中的除数为0，则输出“Inf”。结果中所有的整数都在long int的范围内。

【题目解析】：

本题看上去不难，但是存在几个问题：

1、除数为0，这个很好解决，做个判断即可。

2、负数的输出，这个只要一个标签即可。

3、题目中虽然没有明说，但是这个数字处理后其实是有可能不存在分数部分或者整数部分的。也就是说将数据处理完形成k a/b的格式后，有可能只有一个k，也可能只有一个a/b，也有可能两者皆有，所以要分别考虑这几种情况。

【解题思路】：

可以尝试实现一个有理数类，将数据处理后重载一下加减乘除即可。处理数据的方法就是除一下mod一下的问题，加减乘除遵循基本的分数加减乘除原则，最后求一下最大公约数，做一下约分，再处理一下数据，就OK了。

【示例代码】：

```
#include <iostream>
#include <cmath>

//long long是两个关键字拼起来的，用起来很不方便，重命名一下
typedef long long _sint64;

//有理数类的声明
```

```
class RationalNumber{
    bool m_infinate;        //处理除数为零
    bool m_negative;        //处理负数
    _sint64 m_numerator;    //分子，方便输出
    _sint64 m_denominator;  //分母
    _sint64 m_integer;      //整数部分

    _sint64 m_numeratorAll; //记录无整数分数的分子，方便进行运算
    _sint64 calcGCD(_sint64 a, _sint64 b); //求最大公约数的函数
public:

    RationalNumber(_sint64 numerator, _sint64 denominator); //构造函数
    RationalNumber operator+(RationalNumber const& o) const; //四则运算重载
    RationalNumber operator-(RationalNumber const& o) const;
    RationalNumber operator*(RationalNumber const& o) const;
    RationalNumber operator/(RationalNumber const& o) const;

    //输出流运算符重载
    friend std::ostream &operator<<(std::ostream &os, RationalNumber const& o);
};

//有理数类每个方法的实现
_sint64 RationalNumber::calcGCD(_sint64 a, _sint64 b)
{
    if (b == 0)
    {
        return a;
    }

    //辗转相除法
    return calcGCD(b, a % b);
}

RationalNumber::RationalNumber(_sint64 numerator, _sint64 denominator)
{
    m_negative = false;
    m_infinate = false;
    //处理分母为零的情况
    if (denominator == 0)
    {
        m_infinate = true;
        return;
    }

    //这里这样写，是因为在通过计算结果进行构造过程中，有可能出现分子分母均为负的情况。
    if (numerator < 0)
    {
        m_negative = !m_negative;
    }

    if (denominator < 0)
    {
        m_negative = !m_negative;
    }

    //计算整数、分子、分母。其中分母要参与下面的运算，所以不能是负的，用abs取绝对值，分子要保留原值
    m_integer = numerator / denominator;
```

```
    m_numerator = numerator - m_integer * denominator;
    m_denominator = abs(denominator);

    //约分，注意传给子函数的分子必须是正的，分母上面处理过了
    if (m_numerator)
    {
        _sint64 maxtmp = calcGCD(abs(m_numerator), m_denominator);

        if (maxtmp)
        {
            m_numerator /= maxtmp;
            m_denominator /= maxtmp;
        }
    }

    //计算约分后假分数版的分子，因为后续运算是不需要整数部分的，所以必须用假分数的分子算。
    m_numeratorAll = m_numerator + m_integer * m_denominator;
}

//以下为分数的加减乘除，统统使用m_numeratorAll（假分数的分子）进行运算。
RationalNumber RationalNumber::operator+(RationalNumber const& o) const
{
    _sint64 numerator = (m_numeratorAll * o.m_denominator) +
                        (o.m_numeratorAll * m_denominator);
    _sint64 denominator = m_denominator * o.m_denominator;

    return RationalNumber(numerator, denominator);
}

RationalNumber RationalNumber::operator-(RationalNumber const& o) const
{
    _sint64 numerator = (m_numeratorAll * o.m_denominator) -
                        (o.m_numeratorAll * m_denominator);
    _sint64 denominator = m_denominator * o.m_denominator;

    return RationalNumber(numerator, denominator);
}

RationalNumber RationalNumber::operator*(RationalNumber const& o) const
{
    _sint64 numerator = m_numeratorAll * o.m_numeratorAll;
    _sint64 denominator = m_denominator * o.m_denominator;

    return RationalNumber(numerator, denominator);
}

RationalNumber RationalNumber::operator/(RationalNumber const& o) const
{
    _sint64 numerator = m_numeratorAll * o.m_denominator;
    _sint64 denominator = m_denominator * o.m_numeratorAll;

    return RationalNumber(numerator, denominator);
}

std::ostream &operator<<(std::ostream &os, RationalNumber const& o)
{
    //分母为0的情况就不用继续了
    if (o.m_infinate)
```

```
    {
        os << "Inf";
        return os;
    }

    //整数和分子为0那干脆就是0了
    if (o.m_numerator == 0 && o.m_integer == 0)
    {
        os << "0";
        return os;
    }

    //负数打印括号和负号
    if (o.m_negative)
    {
        os << "(-";
    }

    //有整数就打整数
    if (o.m_integer)
    {
        os << abs(o.m_integer);
        if (o.m_numerator) //整数小数都有就打个空格隔开
        {
            os << " ";
        }
    }

    //有分数就打分数，分母已经abs过了，这里可以不用
    if (o.m_numerator)
    {
        os << abs(o.m_numerator) << '/' << o.m_denominator;
    }

    //负数的后半边括号
    if (o.m_negative)
    {
        os << ")";
    }

    return os;
}

int main()
{
    _sint64 n1, d1, n2, d2;
    scanf("%lld/%lld %lld/%lld", &n1, &d1, &n2, &d2);
    RationalNumber rn1(n1, d1), rn2(n2, d2);

    //轻松+愉快的使用函数时间
    std::cout << rn1 << " + " << rn2 << " = " << rn1 + rn2 << '\n';
    std::cout << rn1 << " - " << rn2 << " = " << rn1 - rn2 << '\n';
    std::cout << rn1 << " * " << rn2 << " = " << rn1 * rn2 << '\n';
    std::cout << rn1 << " / " << rn2 << " = " << rn1 / rn2 << '\n';

    return 0;
}
```

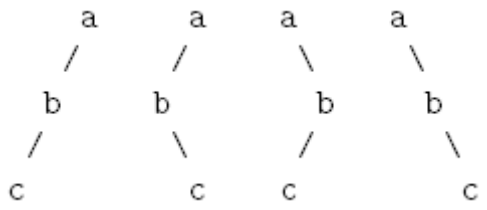
2、题目ID：23561-Pre And Post

https://www.nowcoder.com/questionTerminal/89844f1f632c475ab6f4a600f71683a8

【题目翻译】：

23561-前序和后序

我们都很熟悉二叉树的前序、中序和后序遍历。在数据结构类中，通常会遇到给定中序和后序的情况下求前序的问题，或是给定前序和中序求后序的问题。但一般情况下，当给定树的前序和后序时，并不能确定树的中序遍历。例如下面的这四个二叉树：



它们都拥有着相同的前序和后序。其实这种情况不仅仅限于二叉树，M叉树也是一样。

输入描述：

输入是由多个测试用例组成。每个用例只有一行，格式为m s1 s2，表示树是m叉树，s1是前序遍历，s2是后序遍历。所有字符串将由小写字母字符组成。对于所有的输入实例，1<=m<=20，s1和s2的长度将介于1和26之间（含1和26）。如果s1的长度是k（当然，s2也是这么长），那使用的就是字母表的前K个字母。输入一行0表示终止输入。

输出描述：

对于每个测试用例都要输出一行，表示中序遍历满足该条件的数的个数。输出的范围不会超过int的范围，对于每条用例，都保证至少有一棵树满足要求。

【题目解析】：

这道题本质上其实是一个排列组合问题。通过前序和后序我们虽然还原不出来树，但是谁是谁的子树我们还是知道的。

【解题思路】：

假设我们的前序是abejkcfghid，后序是jkebfghicda，那么我们根据前序，就能知道：

1、最多可以有13颗子树，也就是每一层都有13个可能位置

2、a是根，第一课子树的根是b

3、通过后树我们能知道，b的子树有j、k、e、b共四个结点

4、再回到前序，向前走4个结点，下一棵子树的根是 c

5、以此类推，最终得到 a 为根的下一层共有 3 棵子树

好了三颗子树长这样：

前序 bejk cfghi d

后序 jkeb fghic d

则这一层一共的可能性就是13个空位随便挑3个摆这3颗子树，那么有 C_{13}^{3} 种可能。

之后再递归处理b这棵子树，bejk|jkeb，看以b为根时下一层有多少棵子树。可以看出，只有一棵以e为根的子树，那么可能性就只有C_{13}^{1}种。再递归ejk|jke这棵树，可能情况自然是C_{13}^{2}种，递归cfghi|fghic这棵树，可能情况是C_{13}^{4}种。故而最终结果将会是：

$C_{13}^{3} * C_{13}^{1} * C_{13}^{2} * C_{13}^{4}$种。最终算出这个结果即可。

所以这道题根本上是排列组合问题，我们需要实现排列组合中的C这个方法。

【示例代码】：

```
#include <string>
#include <tuple>
#include <list>
#include <cstdio>

// 求n的阶乘
long long factorial(int n)
{
    long long r = 1;
    for (int i = 1; i <= n; i++)
    {
        r *= i;
    }

    return r;
}

// 求 n, m 的组合   C(n, m)
// 利用 C(n, m) == C(n, n - m) 的特点，计算容易的
long long combination(int n, int m)
{
    int max = m > (n - m) ? m : (n - m);
    long long numerator = 1;
    for (int i = max + 1; i <= n; i++)
    {
        numerator *= i;
    }

    return numerator / factorial(n - max);
}

// 重命名类型，类似于 typedef 作用
using PrePost = std::tuple<std::string, std::string>;

// 给出一棵树的前序+后序，利用最上面注释的原理
// 把每棵子树的前序+后序切分出来
std::list<PrePost> splitSubTrees(std::string const& pre, std::string const&
post)
{
    std::list<PrePost> list{};
    size_t preIdx = 1;
    size_t lastPost = 0;

    while (preIdx < pre.size())
    {
        char rootValue = pre[preIdx];
        size_t postIdx = post.find(rootValue);
```

```
        int countOfNode = postIdx - lastPost + 1;

        list.emplace_back(std::make_tuple(
            pre.substr(preIdx, countOfNode),
            post.substr(lastPost, countOfNode)
            )
        );

        preIdx += countOfNode;
        lastPost += countOfNode;
    }

    return list;
}

// 递归的求解每一层的可能性，直到树中只剩一个或者零个结点
long long calculateNumOfPossibilities(int m,
                                      std::string const& pre,
                                      std::string const& post)
{
    if (pre.size() == 0 || pre.size() == 1) {
        return 1;
    }

    std::list<PrePost> subTrees = splitSubTrees(pre, post);
    long long result = combination(m, subTrees.size());

    for (PrePost const& prePost : subTrees)
    {
        result *= calculateNumOfPossibilities(m,
                                              std::get<0>(prePost),
                                              std::get<1>(prePost));
    }

    return result;
}

int main()
{
    int m;
    char pre[30];
    char post[30];

    while (scanf("%d %s %s", &m, pre, post) != EOF)
    {
        printf("%lld\n", calculateNumOfPossibilities(m, pre, post));
    }

    return 0;
}
```