C++方向编程题答案

答案说明：

> 大家如果对本次题目或者答案有问题，可以联系下方的出题老师答疑。
>
> **出题老师：**
>
> 选择题：吴都 qq：1226631755
>
> 代码题：时亮益 qq：569334855

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】：

```cpp
#include <iostream>
using namespace std;

typedef long long int64;

class Rational
{
public:
    Rational(int64 n, int64 d)
    {
        negetive = false;
        isZero = false;

        // 在输入时分母永远不可能为0，但是经过运算之后分母可能为0
        if(0 == d)
        {
            isZero = true;
            return;
        }

        // 分子小于0，表示为负数
        if(n < 0)
        {
            negetive = !negetive;
        }

        // 在输入时分母一定不会小于0， 但是经过计算之后分母也可能会小于0
        if(d < 0)
        {
            negetive = !negetive;
        }

        // 如果分数是假分数，必须要将其化简为真分数  比如：5 / 3----> 1 2/3
        integer = n / d;
        numerator = n - integer*d;
        denominator = abs(d);

        // 如果不是最简的分数，还需要将其化简为最简的分数：  10 / 15 ----> 2 / 3
        // 只需给分子和分母分别除分子和分母最大公约数
        if(numerator < -1 || numerator > 1)
        {
            int gcd = CalcGCD(abs(numerator), denominator);
            if(gcd)
            {
                numerator /= gcd;
                denominator /= gcd;
            }
        }

        totalnumerator = integer*denominator + numerator;
    }

    Rational operator+(const Rational& r)const
```

```
    {
        int64 n = totalnumerator*r.denominator + r.totalnumerator*denominator;
        int64 d = denominator * r.denominator;
        return Rational(n, d);
    }

    Rational operator-(const Rational& r)const
    {
        int64 n = totalnumerator*r.denominator - r.totalnumerator*denominator;
        int64 d = denominator * r.denominator;
        return Rational(n, d);
    }

    Rational operator*(const Rational& r)const
    {
        int64 n = totalnumerator*r.totalnumerator;
        int64 d = denominator * r.denominator;
        return Rational(n, d);
    }

    Rational operator/(const Rational& r)const
    {
        int64 n = totalnumerator*r.denominator;
        int64 d = denominator * r.totalnumerator;
        return Rational(n, d);
    }
private:
    // 求最大公约数：辗转相除
    int64 CalcGCD(int64 a, int64 b)
    {
        if(0 == b)
            return a;

        return CalcGCD(b, a%b);
    }

    friend ostream& operator<<(ostream& _cout, const Rational& r)
    {
        if(r.isZero)
        {
            _cout<<"Inf";
            return _cout;
        }

        if(0 == r.integer && 0 == r.numerator)
        {
            _cout<<"0";
            return _cout;
        }

        // 如果是负数，需要用()括起来
        if(r.negetive)

        {
```

```
            _cout<<"(-";
        }

        // 输出有理数：整数 + 分数
        // 整数：可能存在也可能不存在
        if(r.integer)
        {
            _cout<<abs(r.integer);

            // 如果分数部分存在，整数和分数之间有一个空格
            if(r.numerator)
            {
                _cout<<" ";
            }
        }

        // 分数：可能存在也可能不存在
        if(r.numerator)
        {
            _cout<<abs(r.numerator)<<"/"<<r.denominator;
        }

        if(r.negetive)
        {
            _cout<<")";
        }
        return _cout;
    }

private:
    int64 numerator;  // 分子
    int64 denominator;  // 分母
    int64 integer;      // 整数部分
    bool negetive;      // 负数
    bool isZero;        // 分母是否为0

    int64 totalnumerator;   // 参与运算的分子：原分子 + 整数部分
};

int main()
{
    int64 n1,d1,n2,d2;
    while(scanf("%lld/%lld %lld/%lld", &n1,&d1,&n2,&d2) != EOF)
    {
        Rational r1(n1,d1);
        Rational r2(n2,d2);
        cout<< r1 <<" + " << r2 << " = " << r1 + r2 << endl;
        cout<< r1 <<" - " << r2 << " = " << r1 - r2 << endl;
        cout<< r1 <<" * " << r2 << " = " << r1 * r2 << endl;
        cout<< r1 <<" / " << r2 << " = " << r1 / r2 << endl;
    }
    return 0;

}
```

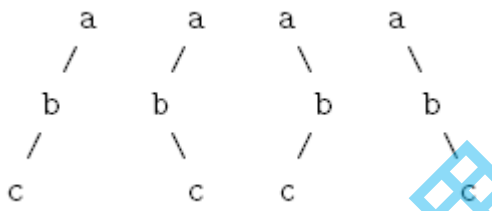
2、题目ID：23561-Pre And Post

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树的个数。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，第一棵子树的根是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 <iostream>
using namespace std;

#include <string>
#include <vector>


// 保存子树的前序和后序遍历结果
struct SubTree
{
    SubTree(const string& pre, const string& post)
        : _pre(pre)
        , _post(post)
    {}

    string _pre;
    string _post;
};


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

    return f;
}


// 求: C(n,m) = C(n, n-m)
long long CalcCom(int n, int m)
{
    m = m < (n-m)? m : (n-m);
```

```
    long long r = 1;
    for(int i = n; i >= n-m+1; i--)
    {
        r *= i;
    }

    return r / Fac(m);
}


// 找根节点的所有子树
vector<SubTree> CalcSubTree(const string& pre, const string& post)
{
    size_t subRootPreIdx = 1;

    size_t postFirst = 0;   // 子树在后序遍历结果中第一个元素的位置

    vector<SubTree> v;

    while(subRootPreIdx < pre.size())
    {
        // 确定子树的根节点
        char subRoot = pre[subRootPreIdx];

        // 确定根在后序遍历结果中的位置
        char subRootPostIdx = post.find(subRoot);

        // 计算该子树中节点的个数
        size_t subTreeNodeCount = subRootPostIdx - postFirst + 1;

        // 找到该棵子树前序遍历结果 - 找到该棵子树后序遍历结果
        SubTree subTree(pre.substr(subRootPreIdx, subTreeNodeCount), post.substr(postFirst,
subTreeNodeCount));

        v.push_back(subTree);

        // 根新下一课子树根在前序遍历结果中的下标
        subRootPreIdx += subTreeNodeCount;

        // 更新下一棵子树中第一个节点在后序遍历结果中的下标
        postFirst += subTreeNodeCount;
    }

    return v;
}

long long CalcTreePossible(int m, const string& pre, const string& post)
{
    // 如果树中只有1个节点---树的可能性就是唯一的
    if(1 == pre.size())
        return 1;
```

```
    // 先找出根节点的所有子树
    vector<SubTree> v = CalcSubTree(pre, post);

    // 计算根节点子树的可能性---组合
    long long result = CalcCom(m, v.size());

    for(auto& e : v)
        result *= CalcTreePossible(m, e._pre, e._post);

    return result;
}

int main()
{
    int m;
    string pre, post;
    // 循环输入来处理每组测试用铝
    while(cin>>m>>pre>>post)
    {
        if(0 == m)
            break;

        cout<<CalcTreePossible(m, pre, post)<<endl;
    }

    return 0;
}
```