

Given a string representing an expression of fraction addition and subtraction, you need to return the calculation result in string format. The final result should be [irreducible fraction](#). If your final result is an integer, say **2**, you need to change it to the format of fraction that has denominator **1**. So in this case, **2** should be converted to **2/1**.

Example 1:

Input: "-1/2+1/2"  
Output: "0/1"

Example 2:

Input: "-1/2+1/2+1/3"  
Output: "1/3"

Example 3:

Input: "1/3-1/2"  
Output: "-1/6"

Example 4:

Input: "5/3+1/3"  
Output: "2/1"

- Note:
- The input string only contains **'0'** to **'9'**, **'/'**, **'+'** and **'-'**. So does the output.
  - Each fraction (input and output) has format **numerator/denominator**. If the first input fraction or the output is positive, then **'+'** will be omitted.
  - The input only contains valid **irreducible fractions**, where the **numerator** and **denominator** of each fraction will always be in the range [1,10]. If the denominator is 1, it means this fraction is actually an integer in a fraction format defined above.
  - The number of given fractions will be in the range [1,10].
  - The numerator and denominator of the **final result** are guaranteed to be valid and in the range of 32-bit int.

## Solution

### Approach 1: Using LCM

The first obvious step to be undertaken is to split the given string into individual fractions. We split the string based on **+** and **-** sign. We store the signs in the order in which they appear in the string in *sign* array. Further, after getting the individual fractions, we further split the fractions based on **/** sign. Thus, we obtain the individual numerator and denominator parts. We store the same in *num* and *den* arrays respectively.

Now, we've got the data ready to be worked upon. In order to see the method we've used in this implementation, we'll take an example and understand the way we work on it.

Let's say, the given fraction is:

$$\frac{3}{2} + \frac{5}{3} - \frac{7}{6}$$

We need to equalize all the denominators so as to be able to add and subtract the numerators easily. The nearest value the denominators can be scaled upto is the LCM of all the denominators. Thus, we need to find the LCM of all the denominators and then multiply all the denominators with appropriate integer factors to make them equal to the LCM. But, in order to keep the individual fraction values unchanged, we need to multiply the individual numerators also with the same factors.

In order to find the LCM, we can go as follows. We use the method  $lcm(a, b, c) = lcm(lcm(a, b), c)$ . Thus, if we can compute the lcm of two denominators, we can keep on repeating the process iteratively over the denominators to get the overall lcm. To find the lcm of two numbers *a* and *b*, we use  $lcm(a, b) = (a * b) / gcd(a, b)$ . For the above example, the *lcm* turns out to be 6.

Thus, we scale up the denominators to 6 as follows:

$$\frac{3 * 3}{2 * 3} + \frac{5 * 2}{3 * 2} - \frac{7}{6}$$


Thus, we can observe that, the scaling factor for a fraction  $\frac{num}{den}$  is given by:  $num * x / den * x$ , where *x* is the corresponding scaling factor. Note that,  $den * x = lcm$ . Thus,  $x = lcm / den$ . Thus, we find out the corresponding scaling factor *x<sub>i</sub>* for each fraction.

After this, we can directly add or subtract the new scaled numerators.

In the current example, we obtain  $\frac{12}{6}$  as the result. Now, we need to convert this into an irreducible fraction. Thus, if we obtain  $\frac{num_i}{den_i}$  as the final result, we need to find a largest factor *y*, which divides both *num<sub>i</sub>* and *den<sub>i</sub>*. Such a number, as we know, is the gcd of *num<sub>i</sub>* and *den<sub>i</sub>*.

Thus, to convert the result  $\frac{num_i}{den_i}$ , we divide both the numerator and denominator by the gcd of the two numbers *y* to obtain the final irreducible  $\frac{num_i/y}{den_i/y}$ .

Note: A problem with this approach is that we find the lcm of all the denominators in a single go and then reduce the overall fraction at the end. Thus, the lcm value could become very large and could lead to an overflow. But, this solution suffices for the current range of numbers.

Java 

```
1 class Solution {
2     public String fractionAddition(String expression) {
3         List<Character> sign = new ArrayList<>();
4         for (int i = 1; i < expression.length(); i++) {
5             if (expression.charAt(i) == '+' || expression.charAt(i) == '-')
6                 sign.add(expression.charAt(i));
7         }
8         List<Integer> num = new ArrayList<>();
9         List<Integer> den = new ArrayList<>();
10        for (String sub : expression.split("\\s+")) {
11            for (String subsub : sub.split("/")) {
12                if (subsub.length() > 0) {
13                    String[] fraction = subsub.split("/");
14                    num.add(Integer.parseInt(fraction[0]));
15                    den.add(Integer.parseInt(fraction[1]));
16                }
17            }
18        }
19        if (expression.charAt(0) == '-') num.set(0, -num.get(0));
20        int lcm = 1;
21        for (int x : den) {
22            lcm = lcm * x;
23        }
24
25        int res = lcm / den.get(0) * num.get(0);
26        for (int i = 1; i < num.size(); i++) {
27            if (sign.get(i - 1) == '+') res += lcm / den.get(i) * num.get(i);
```

- Complexity Analysis
- Time complexity:  $O(n \log x)$ . Euclidean GCD algorithm takes  $O(\log(ab))$  time for finding gcd of two numbers *a* and *b*. Here *n* refers to the number of fractions in the input string and *x* is the maximum possible value of denominator.
  - Space complexity:  $O(n)$ . Size of *num*, *den* and *sign* list grows upto *n*.

### Approach 2: Using GCD

#### Algorithm

We know that we can continue the process of evaluating the given fractions by considering pairs of fractions at a time and continue the process considering the result obtained and the new fraction to be evaluated this time. We make use of this observation, and thus, instead of segregating the signs, numerators and denominators first, we directly start scanning the given strings and operate on the fractions obtained till now whenever a new sign is encountered.


We operate on the pairs of fractions, and keep on reducing the result obtained to irreducible fractions on the way. By doing this, we can reduce the chances of the problem of potential overflow possible in case the denominators lead to a large value of lcm.

We also observed from the last approach, that we need to equalize the denominators of a pair of fractions say:

$$\frac{a}{b} + \frac{c}{d}$$

We used a scaling factor of *x* for the first fraction (both numerator and denominator). Here,  $x = \frac{lcm(b, d)}{b}$ . For the second fraction, the scaling factor *y* is given by  $y = \frac{lcm(b, d)}{d}$ . Here,  $lcm(b, d) = \frac{b * d}{gcd(b, d)}$ . Thus, instead of finding the lcm and then again determining the scaling factor, we can directly use:  $x = \frac{b * d}{gcd(b, d) * b} = \frac{d}{gcd(b, d)}$ , and  $y = \frac{b * d}{gcd(b, d) * d} = \frac{b}{gcd(b, d)}$ . Thus, we need to scale the numerators appropriately and add/subtract them in terms of pairs. The denominators are scaled in the same manner to the lcm of the two denominators involved.

After evaluating every pair of fractions, we again reduce them to irreducible fractions by diving both the numerator and denominator of the resultant fraction by the gcd of the two.


Java 

```
1 class Solution {
2     public String fractionAddition(String expression) {
3         List<Character> sign = new ArrayList<>();
4         if (expression.charAt(0) != '-') sign.add('+');
5         for (int i = 0; i < expression.length(); i++) {
6             if (expression.charAt(i) == '+' || expression.charAt(i) == '-')
7                 sign.add(expression.charAt(i));
8         }
9         int prev_num = 0, prev_den = 1, i = 0;
10        for (String sub : expression.split("\\s+")) {
11            if (sub.length() > 0) {
12                String[] fraction = sub.split("/");
13                int num = Integer.parseInt(fraction[0]);
14                int den = Integer.parseInt(fraction[1]);
15                int g = Math.abs(gcd(den, prev_den));
16                if (sign.get(i++) == '+') prev_num = prev_num * den / g + num * prev_den / g;
17                else prev_num = prev_num * den / g - num * prev_den / g;
18                prev_den = den * prev_den / g;
19                g = Math.abs(gcd(prev_den, prev_num));
20                prev_num /= g;
21                prev_den /= g;
22            }
23        }
24        return prev_num + "/" + prev_den;
25    }
26
27    public int gcd(int a, int b) {
```



- Complexity Analysis
- Time complexity:  $O(n \log x)$ . Euclidean GCD algorithm takes  $O(\log(ab))$  time for finding gcd of two numbers *a* and *b*. Here *n* refers to the number of fractions in the input string and *x* is the maximum possible value of denominator.
  - Space complexity:  $O(n)$ . Size of *sign* list grows upto *n*.



Rate this article: ★★★★★

Comments: 9 Sort By ▾









Type comment here... (Markdown is supported)

 Preview 





vinod23★461 May 26, 2017 10:30 AM



@woshifumingyan You are right, but remember cost of remove is O(n).

0    





woshifumingyuan★68 May 24, 2017 3:22 AM



In the first approach should be  
...  
if (expression.charAt(0) == '-')  
num.set(0, -num.get(0));

0     [Read More](#)





tedtqj★3 May 23, 2020 5:52 AM



Actually don't need to run gcd for two times in the second approach...

0    





Losty★135 April 14, 2020 7:53 AM



The problem can be solved in O(n + log x). You don't need to run GCD for each pair. Instead, bring all pairs to common Denominator (O(1)) to check if common denominator already dividable by current denominator, And run GCD at the end only 1 time to reduce the final fraction.

0     [Read More](#)





achigin★0 February 3, 2020 9:44 PM



Who writes this solutions? Is it author? I was not going to solve this thing, since I got the idea. But the description of the task is awful and unclear. I checked the first simple test case came to my mind: "-1/2--1/2"  
Guess what? Both solutions gave me the wrong answer. I see no value in this parsing type question at all. Dislike.

0     [Read More](#)





mingrui★110 February 3, 2020 1:17 PM



It's simpler to qualize the denominators to the product of all denominators, then use gcd only once in the last phase.

0    

xchen218★60 October 24, 2019 4:36 AM





Well I guess the company really wanted you to know the parsing skills...



0    

AngelicosPhosphoros★6 September 24, 2019 2:03 AM

This problem can be solved with O(1) space complexity.

You just need to create slices generator that provide fractions of string and map it into fractions and reduce them.

0     [Read More](#)

casd82★135 August 14, 2019 7:28 AM

noice

0 