Infinite Precision Arithmetic

LAKSHDEEP SINGH

ROLL NUMBER: CS23BTECH11031

, github id: lakshsidhu04

, India

ACM Reference Format:

Lakshdeep Singh, *Roll Number: CS23BTECH11031*, . 2024. **Infinite Precision Arithmetic**. 1, 1 (April 2024), 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Name → Lakshdeep Singh

Roll Number → *CS23BTECH11031*

github id → lakshsidhu04

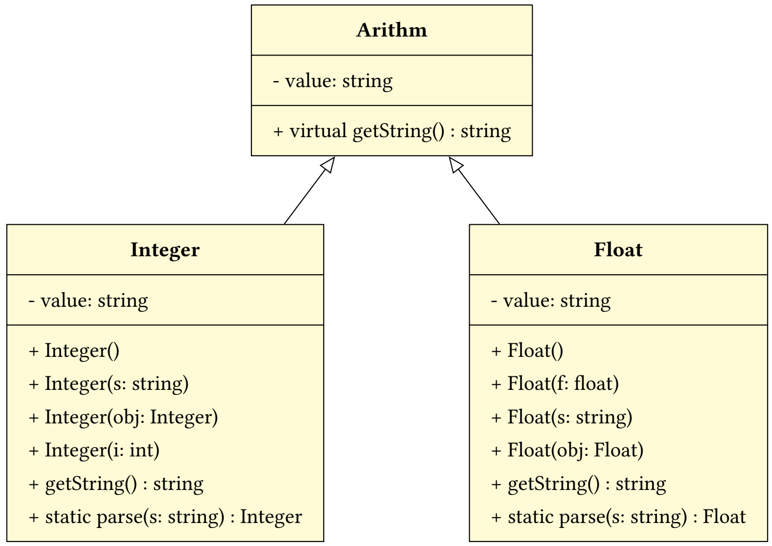
This report is a detailed explanation on the InfiniteArithmetic namespace. The namespace consists of 2 classes : *Integer* and *Float*. These classes provide interface to the user to perform various arithmetic operations with infinitely large numbers which are provided as *strings*. The report contains details about the usage as well as the inner implementation of all these functions. With the InfiniteArithmetic namespace, our software helps users handle huge numbers for tasks like complex calculations, cryptography, or financial analysis.

2 DESIGN

The basic idea is the both Integer and Float classes have similar operations of Addition, Subtraction, Multiplication and Division. Both the classes will support the corresponding Constructors, Destructors and parse function.

Author's address: Lakshdeep Singh
Roll Number: CS23BTECH11031
github id: lakshsidhu04
, India.

Manuscript submitted to ACM



3 README

This section contains to the users on how to use *InfiniteArithmetic* library. The Makefile has prewritten rules to create an executable as well as a linkable library.

(1) make all : creates an executable *my_inf_arith*
(2) make libmy_inf_arith : creates a static linkable .a library named *libmy_inf_arith.a*.

3.1 Using the Executable

The my_inf_arith is the executable generated after using the rule *make all*. The executable requires four parameters to work with:

(1) <u>Data Type</u>: *int*, *float* for Integer and Float class respectively
(2) <u>Operation</u> : *add* (for Addition) , *sub* (for Subtraction), *mul* (for mulitplication) and *div* (for division).
(3) <u>First Number</u>
(4) <u>Second number</u>

An example might look something like this:

```
./my_inf_arith int div 55674 434
```

Always keep in mind, **type of data(chosen by user) has more priority than the actual numbers entered**.

3.1.1 Float boundary cases to be careful of.

(1) A very important information about the library is that if the programmer has chosen float as his data type but entered both integers in the arguments, the hidden implementation will convert these integers into float and then return the answer as a float.
(2) If the user enters a float as .45 or as 54., the code will treat it as a proper float and internally convert them into 0.45 and 54.0 .

(3) If the user has chosen int as the type but entered one or both the numbers as float, the program will internally calculate a temporary answer with these as floats and then trim the decimal part from the final answer. For example:

```
./my_inf_arith int mul 4.5 2.4
```

The float variant would result in 10.8 but int case will return 10 as the answer.

3.2 Using the linkable .a library

The library can be linked with any executable or other libraries.

- If you have an object file *obj.o*, you want to create an executable *exec* by linking it with *libmy_inf_arith* ,we can use the command:

```
g++ -o exec obj.o libmy_inf_arith.a
```

- If you have another library *other_lib.a* , you want to create an executable *exec* by linking the object file, *obj.o* by linking it with *libmy_inf_arith* , we can use command:

```
g++ -o exec obj.o libmy_inf_arith.a other_lib.a
```

Now, the created executable can be used as explain in 3.1 .

4 FILE STRUCTURE AND DISTRIBUTION

The implementation mainly consist of 3 files:

- main.cpp → This file is the starting point of the execution. It uses the interface of **InfiniteArithmetic** namespace to handle user inputs and perform operations accordingly.
- interface.h → This file provides the interface for the library user and the structure(declaration) of **InfiniteArithmetic** Namespace.
- interface.cpp → This file provides implementation of all the addition, subtraction, multiplication and division operations for both *Integer* and *Float* class.

The other files include *main.o* and *interface.o* which are the corresponding object files. After using makefile, an executable *my_inf_arith* and static library *libmy_inf_arith.a*

5 ARITHM CLASS

The Arithm class is the *Abstract base* class for *Integer* and *Float*. It defines the purely *virtual* functions which must be overloaded in the *Derived* classes

6 INTEGER CLASS

The most important of the class is the string *value* where the number is stored. It is kept private from the user. The Integer class supports all the basic constructors and destructors. The constructors include:

- Integer() : The default constructor which initializes the string with zero.
- Integer(std::string s) : This initialises string with the argument string s.

- Integer(Integer& obj) : Copy Constructor
- Integer(int i) : To initialize with a normal integer.

Other functions include

- getString() : returns the value string of the respective object through which it is called.
- static parse(string s) : Parses the string and returns an instance of Integer class.

6.1 Addition and Subtraction

The Additions and Subtractions follow very similar approach. The operations are first performed digit by digits from the ones position(last index). The Algorithms for Addition and Subtraction are very correlated. Addition can be used in place of Subtraction with tweaking of some signs and vice versa.

6.1.1 ***Addition***.

(1) The simplest implementation of addition is when both the numbers are positive. The basic idea is to align the numbers towards right and start adding digits from the units place. With respect to strings, this means that addition starts from last index. The carry overs are taken from ith index to $i-1$ index. The tricky case is if the lengths of the strings are different. For this we align the numbers towards right and if the digits of any one strings run out, the digit is taken as zero. Refer Algorithm 1 . For example:

	5	4	3	2	4
+	0	0	4	3	3
=	5	4	7	5	7

Algorithm 1: Integer Addition

Function $s1, s2$
  Input: $s1$, $s2$: Strings to be added
  Output: ans: Resultant string after addition
  $l1 \leftarrow \text{length}(s1)$;
  $l2 \leftarrow \text{length}(s2)$;
  $l \leftarrow \max(l1, l2)$; `// The sum will go on till the longest string ends`
  $carry \leftarrow 0$;
  for $i \leftarrow 0$ **to** $l - 1$ **do**
    $first\_digit \leftarrow \text{if } i < l1 \text{ then } s1[l1 - i - 1] - \text{ord}('0') \text{ else } 0$; `// If s1 runs out of digits`
    $second\_digit \leftarrow \text{if } i < l2 \text{ then } s2[l2 - i - 1] - \text{ord}('0') \text{ else } 0$; `// if s2 runs out of digits`
    $currSum \leftarrow first\_digit + second\_digit + carry$;
    if $currSum > 9$ **then**
      $carry \leftarrow 1$; `// in case of a carry over`
      $currSum \leftarrow currSum - 10$;
    end
    else
      $carry \leftarrow 0$;
    end
    $ans \leftarrow toString(currSum) + ans$; `// converting int to char and appending to ans`
  end
  `// Add the carry if there is a carry after strings run out`
  if $carry > 0$ **then**
    $ans \leftarrow "1" + ans$;
  end
  return ans;

(2) The second case is when both the numbers are negative. This follows a similar approach as above, the only change being that the final answer will be the negative of the answer using above approach. The change we make here is that first we remove the '−' sign from both the strings and add them using the Algorithm 1. After it is calculated, we again insert a '−' sign at the beginning of the answer. The flag *neg* keeps a track if the answer be negative or not. Refer Algorithm 2

Algorithm 2: Addition when both strings have negative sign

Input : $s1$, $s2$: Input strings
Output: neg: Boolean indicating if both strings have negative sign at the beginning

```
neg ← false;
if s1[0] = '-' and s2[0] = '-' then
    neg ← true;
    s1 ← substr(s1, 1);
    s2 ← substr(s2, 1);
end
Use Algorithm 1
ans = '-' + ans;
```

(3) The next case comes if only one of the numbers is negative. We can simplify it as following:

$$s1 + s2$$

$$(s1 + (-s2)) \mid\mid ((-s1) + s2)$$

It simplifies to:

$$(s1 - s2) \mid\mid (s2 - s1)$$

Therefore, we will use the subtraction algorithm here to assist with the addition. The final answer will be negative if the magnitude of the negative number is more than the magnitude of the positive number. Refer Algorithm 3 The other case where s1 is non-negative and s2 is negative can be handled similarly as below.

Algorithm 3: Addition where only one string has a negative sign at the beginning

Data: $s1$, $s2$: Input strings
Result: *ans*: Resultant difference

```
if s1[0] ==' −' and s2[0] ≠' −' then
    // If s1 is negative and s2 is non-negative
    s1 ← substr(s1, 1);
    if checkMax(s1, s2) == s1 then
        // If s1 is maximum
        neg ← true;
        ans ← s1 - s2;
    end
    else
        // If s2 is maximum
        ans ← s2 - s1;
    end
end
```

6.1.2 *Subtraction*.

(1) The simplest subtraction is when both the numbers are positive. The approach is similar to addition Algorithm 1. The process starts from the units digit and subtraction is done digit by digit. If the digit difference is negative,

a borrow is taken from the next higher order digit and process is repeated till last index. In the next iteration, if the borrow is not 0, the current digit is decreased by one. Refer *Algorithm 4*.

Algorithm 4: Integer Subtraction

Input : $s1$, $s2$: Input strings
Output: ans: Resultant string after subtraction

```
max ← checkMax(s1, s2) ;                          // the larger number will be checked using this
if max = s2 then
    neg ← true ;                                  // flag for sign of the final answer
end
min ← (max = s1) ? s2 : s1;
ans ← "";
l1 ← length(max);
l2 ← length(min);
borrow ← 0;
for i ← 0 to l1 − 1 do
    first_digit ← if i < l1 then max [l1 - i - 1] - '0' else 0 ;     // in case of overflow of max
    second_digit ← if i < l2 then min [l2 - i - 1] - '0' else 0 ;    // in case of overflow of min
    currDiff ← first_digit − second_digit − borrow ;                 // borrow reduced at each iteration
    if currDiff < 0 then
        // in case of a borrow
        borrow ← 1 ;                              // borrow to be reduced in next iteration
        currDiff ← currDiff + 10;
    end
    else
        borrow ← 0;
    end
    ans ← std::to_string(currDiff) + ans;
end
return ans;
```

---

(2) The second case is if the first number is positive and second number is negative. This case can be seen as:

$$s1 - (-s2)$$

$$s1 + s2$$

Now this is nothing but an addition problem. The answer will always be positive. We use addition Algorithm 1 to assist us. Refer Algorithm 5

(3) Now, the left cases are when:

(a) First number is negative and the second number is positive. It can be seen as:

$$-s1 + (-s2)$$

$$-(s1 + s2)$$

This can be seen as adding the magnitude of the numbers and adding a − sign. This is very similar to Algorithm 2.

Algorithm 5: Algorithm for handling the case where only second string s2(subtrahend), has a negative sign at the beginning

Input : $s1$, $s2$: Input strings
Output : ans: Resultant sum

if $s2[0]$ = '-' ***and*** $s1[0] \neq$ '-' **then**
  $s2 \leftarrow$ substr($s2$, 1);
  neg $\leftarrow$ false;
  ans $\leftarrow$ s1 + s2 `// used the addition algorithm here`
end

(b) Both the numbers are negative. This can be seen as:

$$-s1 - (-s2)$$

$$s2 - s1$$

This can lead to two cases

(i) s2 > s1
The answer will be positive.

(ii) s1 > s2
The answer will be negative.

Algorithm 6: Integer Subtraction when both strings are negative

if $s1[0]$ *is* '-' *AND* $s2[0]$ *is* '-' **then**
  $s1 \leftarrow$ s1.substr(1);
  $s2 \leftarrow$ s2.substr(1);
  Create temporary objects $tmp1$, $tmp2$ of class InfiniteArithmetic::Integer initialized with $s1$ and $s2$ respectively;
  Create an empty object *ans*;
  if *checkMax(*$s1$*,* $s2$*) equals* $s1$ **then**
    $ans \leftarrow tmp1 - tmp2$;
    $neg \leftarrow$ true;
  else
    $ans \leftarrow tmp2 - tmp1$;
    $neg \leftarrow$ false;
  $ansStr \leftarrow$ ans.getString();
  if *neg AND ansStr is not "0"* **then**
    `// adding negative sign to zero has no meaning`
    $ansStr \leftarrow$ "-" + $ansStr$;
  Create object obj of class Integer initialized with $ansStr$;
  return obj;

6.2 Multiplication and Division

6.2.1 ***Multiplication.*** The Multiplication of Integers uses basics of single digit multiplication and addition. The Multiplication involves adjusting the carry overs over multiple iterations. Let's say the $l1$ and $l2$ are lengths of the input strings, the length of the answer cannot be more than $l1 + l2$ in any case. So, we initialise answer string with $l1 + l2$ zeroes. The algorithm does not require repeated creation of strings, it stores carry overs in the same *ans* string and adds it accordingly at each iteration. For example, the value at any index i will keep updating according to the carry overs without any need of repitive strings. The flag *neg* will track if the final answer is negative or not.

Algorithm 7: Integer Multiplication

$l1 \leftarrow$ length of $s1$;
$l2 \leftarrow$ length of $s2$;
$ans \leftarrow$ "0" repeated $l1 + l2$ times;
for $i \leftarrow l1 - 1$ **to** 0 **do**
    for $j \leftarrow l2 - 1$ **to** 0 **do**
        $mul \leftarrow (s1[i] -' 0') \times (s2[j] -' 0')$; `// digit by digit product`
        $sum \leftarrow mul + (ans[i + j + 1] -' 0')$; `// adding the previous number at the corresponding place`
        $ans[i + j + 1] \leftarrow sum \mod 10$; `// storing the units digit`
        $ans[i + j] \mathrel{+}= sum \div 10$; `// carry over at the next place`
while $ans[0]$ *equals '0'* **do**
    `// if the first digit is zero, because we took l1+l2 zeroes`
    $ans \leftarrow ans.substr(1)$
if *neg==true* **then**
    `// If only any one of the numbers is negative, the answer will be negative`
    $ans \leftarrow ' -' + ans$
return *ans*;

The algorithm uses 2 nested for loops to perform the product. The outer loop takes a digit of a number and multiplies it with all the digits of the second number one by one, storing it in the answer. At each iteration using i and j, the loop will store the current product and index $i + j + 1$. The carry over will be placed at $(i + j)$th position. At each iteration, the product will be computed and the carry over, which is already present at that place will be added.At each iteration, the product will be computed and the carry over, which is already present at that place will be added. This can be seen as:

index	0	1	2	3	4
place	-	(i+j-1)	(i+j)	(i+j+1)	-

Table 1. Table with Arrow from 4th to 3rd column

6.2.2 ***Division.*** The division algorithm follows the steps of **Long Division method**. The first iteration involves taking the substring of dividend equal to length of divisor. If the substring is less than the divisor, a digit from dividend is appended to the substring, the division is performed and the quotient is added to the answer. If an extra digit is required for division, a zero is added to the answer. The division will be stop once the remaining string becomes less than the divisor. Refer Algorithm 8

Algorithm 8: Integer Division

```
tmp ← s1;
count ← 0;
while (tmp.length() > 0 and tmp.length() ≥ s2.length() and tmp ≠ "" and tmp ≠ "0") do
    count ← count + 1;
    curr ← substring of tmp from 0 to tail;
    curr2 ← removeZeroes(curr);
    while checkMax(curr2, s2) == s2 and curr2 ≠ s2 do
        if (tail < s1.length()) then
            tail ← tail + 1 ;                 // loop to find the appropriate substring for division
        end
        else
            ans ← ans + "0";
            break;
        end
        if (count > 1) then
            // case when extra carry over is required in long division
            ans ← ans + "0";
        end
        curr ← substring of tmp from 0 to tail;
        curr2 ← removeZeroes(curr);
        if checkMax(curr2, s2) == curr2 or curr2 == s2 then
            break;
        end
    end
    dividend ← Integer(curr2);
    divisor ← Integer(s2);
    for i ← 1 to 10 do
        tempQuo ← Integer(i) tempProd ← String(divisor × tempQuo) ;   // loop to find remainder and
         quotient at that iteration
        if checkMax(tempProd, curr2) == tempProd and tempProd ≠ curr2 then
            ans ← ans+ String(i − 1);
            quotient ← Integer(i − 1);
            temp ← quotient × divisor;
            rem ← (dividend − temp);
            break;
        end
    end
    tmp ← rem+ tail.substr(tail);
    tail ← tail + 1;
    check if tmp is zero
    if zero then
        add the corresponding number of zeroes;
    end
    if checkMax(tmp2, s2) == s2 then
        break ;                             // stop if remaining s1 is less than divisor s2
    end
end
```

7 FLOAT CLASS

Float class, similar to Integer class, supports the similar constructors and destructor.

- Float() : The default constructor which initializes the string with zero.
- Float(std::string s) : This initialises the value string with the argument string s.
- Float(Float& obj) : Copy Constructor

Other functions include *getString()* and *parse* similar to Integer Class

7.1 Addition and Subtraction

Addition and Subtraction of *Float* class uses *Integer* addition after removing the decimal point. The magnitude and the sign of the answer will be adjusted by the *Integer* addition while the decimal is taken care by the *Float* implementation.

7.1.1 ***Addition***. Addition of Floats is basically aligning both numbers with respect to the decimal point and adding them as integers. The mismatching number of digits in the trailing decimals can be replaced with zero. For example if we want to add 234.234 and 34.34 , we can write 34.34 as 34.340 to add them. Aligning might look something like this:

$$\begin{vmatrix} 234.234 \\ 34.340 \end{vmatrix}$$

Now we remove the decimal and proceed them to add as Integers. After this, we put the decimal back in the resultant sum. The sign will automatically be taken care by Integer Addition. Refer Algorithm 9.

Algorithm 9: Float Addition

```
Procedure AddStrings
    s1, s2
s1 ← RemoveDecimal(s1);
s2 ← RemoveDecimal(s2);
Add the required trailing zeroes;
Make the decimal digits equal sum ← s1 + s2 ;          // use of integer addition
dec ← (dec1 > dec2) ? dec1 : dec2;
if sumStr == "-" then
    sumStr ← "0";
if sumStr != "0" then
    sumStr ← sumStr.substr(0, sumStr.length() - dec) + "." + sumStr.substr(sumStr.length() - dec);
return ans;
```

7.1.2 ***Subtraction***. Similar to *Float* Addition, *Float* Subtraction also uses *Integer* Subtraction. The process involves removing the decimal number, subtracting and putting the decimal back. The amazing thing here is that the sign of the resultant will already be taken care by the Integer subtraction. Refer Algorithm 10

Algorithm 10: Float Subtraction

```
Procedure Subtract Strings
    s1, s2
s1 ← RemoveDecimal(s1);
s2 ← RemoveDecimal(s2);
Add the required trailing zeroes;
Make number of decimal digits equal for easy addition
diff ← s1 − s2;
ans ← Put the decimal back;
return ans;
```

7.2 Mulitplication and Division

Both *Float* Multiplication and Division use *Integer* Multiplication and Division at their core.

7.2.1 ***Multiplication***. The *Float* Multiplication can be seen as multiplying the numbers without the decimal and adding the decimal back to the result. It is known that the decimal will be placed in $l1 + l2$ th position from behind $l1$ and $l2$ being the lengths of respective numbers without the sign. After the magnitude calculation, sign will be put in the front.

Algorithm 11: Float Multiplication

```
dec1 ← 0, dec2 ← 0;
for i ← 0 to length(s1) do
    // find decimal in s1
    if s1[i] ==' .' then
        dec1 ← length(s1) − i − 1;
        s1 ← substring(s1, 0, i) + substring(s1, i + 1);
        break;
    end
end
;                                                   // similar for s2
tempProd ← tmp1 × tmp2;
ans ← tempProd.getString();
if s3 == "0" then
    return InfiniteArithmetic::Float("0");
end
dec ← dec1 + dec2 ;          // decimal in final answer will be sum of both numbers
Put the decimal back
```

7.2.2 ***Division***. The *Float* Division can be seen as recurring use of *Integer* Division. The first iteration will give the integer part of the quotient and further ones will combine to give decimal part of the quotient. At each iteration, the dividend is the remainder of the previous iteration with an added zero which is allowed in decimal division. The implementation also involves making the digits after decimal point same which makes the division answer. The first iteration computes the integer part of the quotient. We keep track of the length of integer part which will help us to put decimal back in the end. Also, the 1000th digit is rounded off according to 1001th digit.

Algorithm 12: Float Division

```
while remainder ≠ "0" do
    count++ ;                                   // keeps track of number of iterations
    tmp1 = rem ;                                // remainder of previous iteration is the current dividend
    tempQuo = tmp1 / tmp2 ;                     // the current quotient
    if count == 1 then
        // first iteration means the tempQuo is the integer part of the final quotient
        int_quotient_length = tempQuo.getString().length() ; // the length of the integer quotient part
    end
    ans = ans + tempQuo.getString() ;           // appending to the quotient
    InfiniteArithmetic::Integer tempProduct;
    tempProduct = tmp2 × tempQuo;
    tempStr2 = tempProduct.getString();
    InfiniteArithmetic::Integer tempProd(tempStr2);
    rem = tmp1 - tempProd ;                     // calculate remainder
    tempStr = rem.getString();
    if tempStr == "0" then
        break ;                                 // stop once the remainder becomes zero
    end
    tempStr = tempStr + "0" ;                   // Decimal division allows an extra zero
    if checkMax(tempStr,s2) == s2 then
        tempStr = tempStr + "0" ;               // if added zero does not make the dividend more than divisor
        ans = ans + "0";
    end
    rem = InfiniteArithmetic::Integer(tempStr);
    if count == 1001 then
        break ;                                 // stop once decimal precision reaches 1000 digits
    end
end
```

8 POINTS TO KEEP IN MIND

(1) The type chosen by user is of higher priority then the type of numbers entered.

(2) The boundary cases as discussed in 3.1.1.
(3) Keep sure not to enter a string which is not a integer or a decimal because doing so will result in an error.
(4) The user can enter the number as .653 instead of 0.653.
(5) Their is no restriction on number of trailing zeroes or leading zeroes entered. For example, the user can enter 0.953000 or 0.9530 or 0000.953, all will be treated same.

9 LIMITATIONS

- The precision upto 1000 digits can result in very wrong answers in some cases. For example, $\frac{1}{10^{2000}}$ will result in 0 but it is not zero.
- If the user has chosen the type as int, but entered one or more float numbers, the final answer will be an int in every case. So, the type has a greater priority over the actual numbers entered by the user. For example, if the user has chosen type as *int* and the operation as *add* and user enters 334.22 and 543.99, the actual answer should be 878.21, but the library will return only 878 as the answer.

10 VERIFICATION PROCESS

The verification of test cases was done using :

- scripting test cases $\longrightarrow$ The python and shell scripts can be used to generate random cases and the executable can be run using automation.
- manual generation of test cases $\longrightarrow$ I have manually typed some test cases which can result in possible errors or wrong answers. All the tricky test cases cannot be automatically generated, so I have typed the possible tricky ones manually and verified the answers from the Infinite Precision Calculators on the web.

11 LEARNINGS FROM THE PROJECT

Here are some of my learnings from the project:

- Concepts of OOPs $\longrightarrow$ I learnt how the concepts of OOPs can help to create real-life Software Design.
- Utilisation of makefile $\longrightarrow$ Using makefile makes it a lot easier to compile and run files rather than writing repititive commands for the same. For example, without makefile, I will have to write 3 separate commands but using makefile all I have to do is define a rule for it in the makefile and run:

```
make <rulename>
```

- The importance of gdb $\longrightarrow$ gdb debugging helped me a lot while to solve code bugs.
- The Modularity of code $\longrightarrow$ The division of repititive code into functions can really improve code efficiency and programmability.
- Real Life usage of Scripting $\longrightarrow$ The shell scripts and python scripts can be used to efficiently generate random test cases and verify the answers.

12 COMMITS

Here are the screenshots which display the commit history and version.

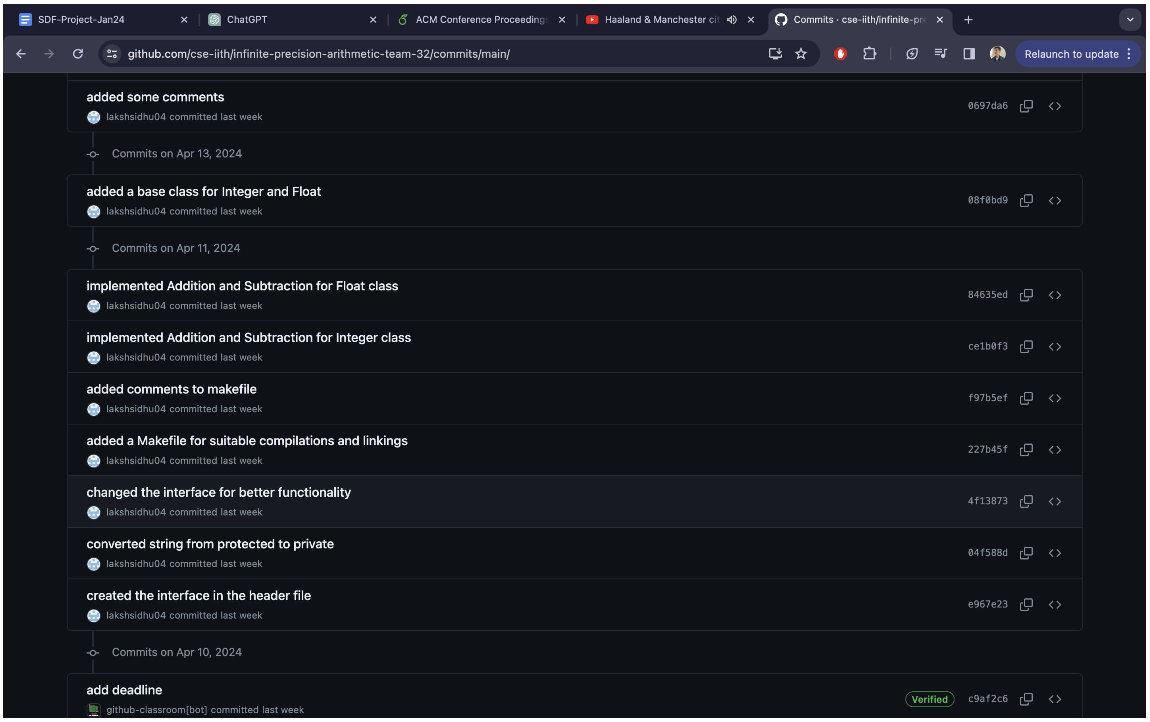


Fig. 1. Screenshot 1

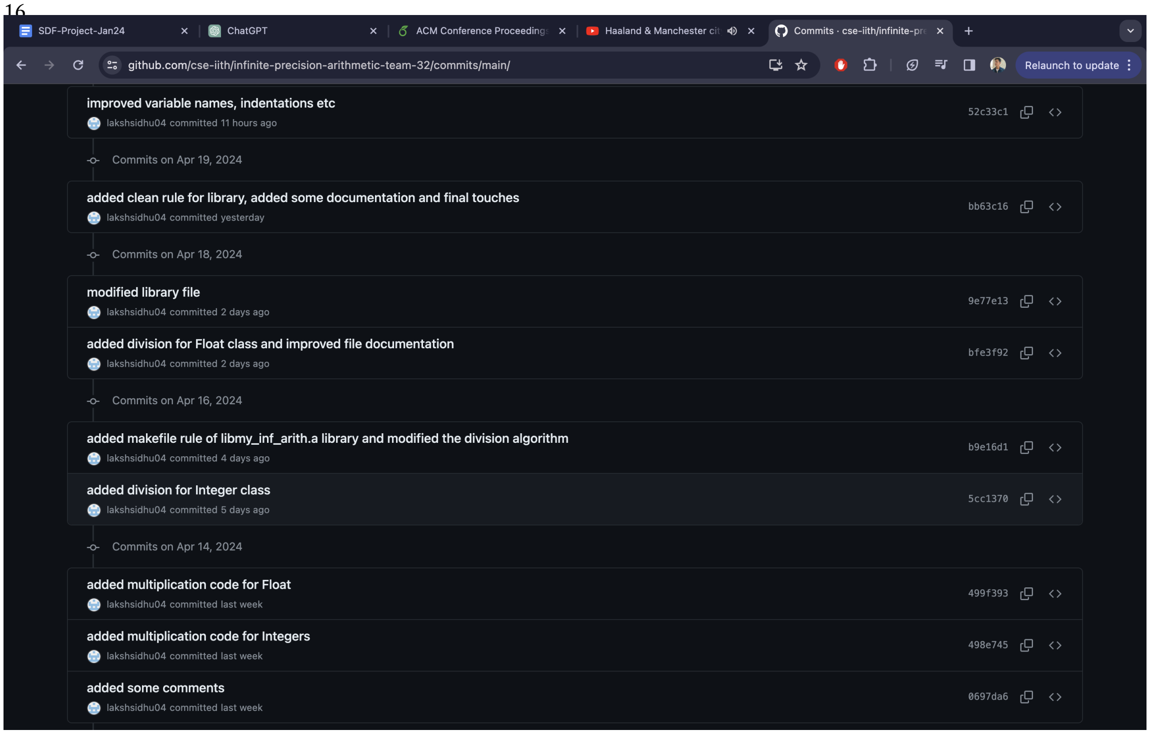


Fig. 2. Screenshot 2

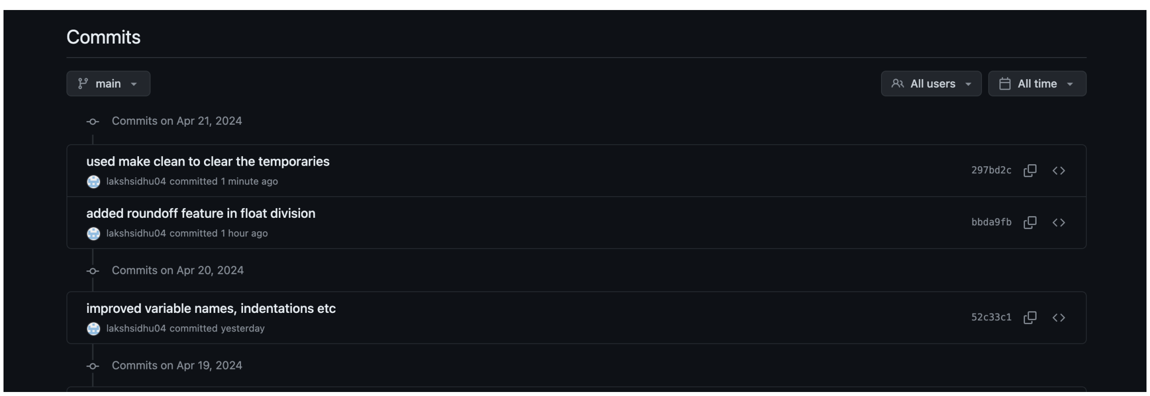


Fig. 3. Screenshot 3