



Faculty of Engineering & Technology  
Electrical & Computer Engineering Department

**COMPUTER ORGANIZATION AND ASSEMBLY  
LANGUAGE - ENCS336**

**Final Project Report**

**GCD and LCM and using them in adding fractions**

---

**Prepared by:**

Obada Tahayna

1191319

**Instructor:** Dr. Abdallatif Abuissa

**Section:** 1

**Date:** 02/01/2021

# Table of Contents

Table of Contents.....	II
Table of Figures .....	III
Program Overview .....	4
Declaring.....	4
Macros.....	4
printStr.....	4
printNum .....	4
scan.....	5
printThreeDigit.....	5
printFiveDigit.....	6
Code Area .....	7
Part1: Calculate GCD and LCM of Two Numbers .....	7
Part2: Find the Result of Adding Two Fraction Using GCD .....	8
Procedures.....	11
getGCD .....	11
scanThreeDigit.....	13
Outputs Samples.....	14
Part1: Calculate GCD and LCM of Two Numbers .....	14
Part2: Find the Result of Adding Two Fraction Using GCD .....	15
Full Output.....	16

# Table of Figures

Figure 1: Declaring .....	4
Figure 2: printStr Macro .....	4
Figure 3: printNum Macro .....	4
Figure 4: scan Macro .....	5
Figure 5: printThreeDigit Macro .....	5
Figure 6: printFiveDigit Macro.....	6
Figure 7: The Code of Reading 2 Numbers .....	7
Figure 8: Code Area - Calculate GCD and LCM .....	7
Figure 9: Code Area - Print GCD and LCM of the Two Numbers.....	8
Figure 10: Code for Reading Fractions.....	8
Figure 11: Code for Making the Denominators the Same .....	9
Figure 12: Code for Adding Numerators and Check error .....	9
Figure 13: Code for Simplifying the Result and Print it.....	10
Figure 14: Example of finding GCD using Euclid's Algorithm .....	11
Figure 15: getGCD Procedure Code .....	12
Figure 16: scanThreeDigit Procedure Code.....	13
Figure 17: Sample of Part1 Outputs (With Error Messages).....	14
Figure 18: Sample of Part1 Outputs (Without Invalid Inputs) .....	14
Figure 19: Sample of Part2 Outputs (With Invalid Input Errors) .....	15
Figure 20: Sample of Part2 Outputs (With Big Number Error).....	15
Figure 21: Full Output of The Program .....	16

# Program Overview

## Declaring

First of all, I declared the capacity of the model (small), and the stack size (100), and then the variables declared.

```
.model small
.stack 100
.data
msgWelcome db "Hello!", 10, 10, 13, 's'
msgEnter db 10, 13, "Please Enter Number s"
msgColon db ":", 10, 13, 's'
msgNewLine db "\n", 10, 13, 's'
msgErrorIn db "Error in Input, This is INVALID Number, Please enter just numbers", 10, 10, 13, 's'
msgErrorBet db "Error in Input, This is INVALID Number, Please enter number between 0 and 255", 10, 10, 13, 's'
msgEq db "The result is equal to:", 10, 13, 's'
msgLcm db "LCM = s"
msgExample db "Example of fractions addition, Enter two fractions", 10, 13, 's'
msgSlash db " / ", 10, 13, 's'
msgEqual db "=", 10, 13, 's'
msgInvalidNumberError db "Error, the result of adding first numerator with the second one after reduction of fractions is greater than 65536 and It cannot be processed", 10, 13, 's'
msgBye db 10, 10, 13, "Bye", 10, 13, "Bada Tahayna 1191319", 10, 13, 's'

number dw 00h
number1 dw 00h
number2 dw 00h
errIn db 0
temp db 0
counter dw 0
ten dw 0
rem1 dw 0
rem2 dw 0
LCDvalue dw 0
LCDvalue1 dw 0
LCDvalue2 dw 0
numerator1 dw 0
denominator1 dw 0
numerator2 dw 0
denominator2 dw 0
digit1 db 0
digit2 db 0
digit3 db 0
digit4 db 0
digit5 db 0
```

Figure 1: Declaring

## Macros

After the variables grabbed to `ds` and the value of `ax` made to be zero then the macros started.

```
051 mov ax, @data
052 mov ds, ax
053 mov ax, 0
```

### printStr

```
057 printStr macro message
058     mov ah, 09
059     lea dx, message
060     int 21h
061 endm
```

Figure 2: printStr Macro

This macro is to print a string message.

### printNum

```
066 printNum macro num
067     mov ah, 02
068     add num, 30h
069     mov dl, num
070     int 21h
071     sub num, 30h
072 endm
```

Figure 3: printNum Macro

Macro to print a number of one digit.

## scan

```
077
078
079
080

scan macro
    mov ah, 01
    int 21h
endm
```

Figure 4: scan Macro

Scan macro to get a one-digit number from user using `int 21h`.

## printThreeDigit

```
085
086
087
088
089
090
091
092
093
094
095
096
097
098
099
100
101
102
103
104

printThreeDigit macro num
    mov dx, 0
    mov ax, num
    div tenW
    mov digit1, dl

    mov dx, 0
    div tenW
    mov digit2, dl

    mov dx, 0
    div tenW
    mov digit3, dl

    printNum digit3
    printNum digit2
    printNum digit1
endm
```

Figure 5: printThreeDigit Macro

Macro that prints any 3-digit number, this macro built on printNum macro, by split the number to its digits and print each digit using printNum macro.

## printFiveDigit

```
112      printFiveDigit macro num
113
114          mov dx, 0
115          mov ax, num
116          div tenW
117          mov digit1, dl
118
119          mov dx, 0
120          div tenW
121          mov digit2, dl
122
123          mov dx, 0
124          div tenW
125          mov digit3, dl
126
127          mov dx, 0
128          div tenW
129          mov digit4, dl
130
131          mov dx, 0
132          div tenW
133          mov digit5, dl
134
135          printNum digit5
136          printNum digit4
137          printNum digit3
138          printNum digit2
139          printNum digit1
140      endm
```

Figure 6: printFiveDigit Macro

Like printThreeDigit macro, printFiveDigit Macro uses printNum macro to print the 5 digits of the 5-digit number.

## Code Area

### Part1: Calculate GCD and LCM of Two Numbers

#### Read Two Numbers

Loop to take 2 numbers from 3 digits and save them in the memory at first, then the values of 2 numbers taken from memory to variables (`number1` and `number2`) after exiting the loop.

In the procedure `scanThreeDigits`, if any digit isn't a number, the value of the `errIn` flag variable becomes to be 1 ([see how](#)), and here in the code when we get the two numbers, if any number has `errIn` equals 1 then that means there is an error with this number, so the program asks the user to enter it again.

If all digits are numbers (`errIn` is 0), then the program checks if the number entered is greater than 250, if so, an error message will show on the screen and the program will ask the user to enter the number again.

```
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194

mov cx, 2
getTwoNumbers:
    mov temp, cx

    rete:
        add counter, 1

        printStr msgEnter
        printNum counter
        printStr msgColon
        call scanThreeDigit

        cmp errIn, 1
        je errorInvalid1

        mov ax, 250
        cmp number, ax
        ja errorBetween1

        mov ax, 1
        cmp number, ax
        jb errorBetween1

        mov bx, number

        mov [si], bx
        add si, 2

        mov cx, temp
        loop getTwoNumbers

mov si, 0
mov ax, [si]
mov number1, ax

add si, 2
mov ax, [si]
mov number2, ax
```

Figure 7: The Code of Reading 2 Numbers

#### Calculate GCD and LCM

```
203
204
205
206
207
208
209
210
211
212

;calculate GCD and LCM
call getGCD

mov ax, number1
mul number2
div GCDValue

mov LCMValue, ax
;;
```

Figure 8: Code Area - Calculate GCD and LCM

The program calls [getGCD procedure](#) that calculates the GCD of `number1` and `number2`, then the result of LCM calculated as follow:  $LCM = (number1 * number2) / GCD$ .

## Print the GCD and LCM of the Two Numbers

The values of GCD and LCM printed after calculate them as the code mentioned.

```

217     printStr msgNewLine
218     printStr msgNewLine
219     printStr msgNewLine
220
221
222     printStr msgGCD
223     printFiveDigit GCDValue
224
225
226
227     printStr msgLCM
228     printFiveDigit LCMValue
229

```

Figure 9: Code Area - Print GCD and LCM of the Two Numbers

## Part2: Find the Result of Adding Two Fraction Using GCD

### Reading Fractions

```

255     mov counter, 1
256     rete2:
257         reteP1:
258             cmp counter, 1
259             je one
260
261         reteP2:
262             cmp counter, 2
263             je two
264
265         reteP3:
266             cmp counter, 3
267             je three
268
269         reteP4:
270             cmp counter, 4
271             je four
272
273         reteP5:
274             cmp counter, 5
275             je five
276
277
278     rete22:
279
280     cmp errIn, 1
281     je errorInvalid2
282
283
284     mov ax, 250
285     cmp number, ax
286     ja errorBetween2
287
288     mov ax, 1
289     cmp number, ax
290     jb errorBetween2
291
292
293     cmp counter, 1
294     je S1
295
296     cmp counter, 2
297     je S2
298
299     cmp counter, 3
300     je S3
301
302     cmp counter, 4
303     je S4
304
305     reteS:
306     add counter, 1h
307     jmp rete2
308
309
310
311     one:
312     call scanThreeDigit
313     jmp rete22
314
315
316     two:
317     printStr msgSlash
318     call scanThreeDigit
319     jmp rete22
320
321
322     three:
323     printStr msgplus
324     call scanThreeDigit
325     jmp rete22
326
327
328     four:
329     printStr msgSlash
330     call scanThreeDigit
331     jmp rete22
332
333
334     five:
335     printStr msgEqual
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

Figure 10: Code for Reading Fractions

The code that reads fractions looks very complex, the complexity of the code is because the way that the code handles with input errors (such as entering characters or symblos or entering numbers greater than 250).

In this part, the program will read 4 integers, first numerator, first denominator, second numerator and the second denominator. If the user entered invalid input in any of the four numbers, the porgram will show an error message and will ask the user to enter the number again after displaying the previous entered numebrs. ([see the outputs](#))



## Calculations

### Making the Denominators the Same

First, the program will calculate the GCD of the first denominator and the second one, Then it calculates the deviation of each denominators by the GCD calculated to find two values and store them on `rem1` and `rem2`, After that the program multiplies `numerator1` with `rem2` and `numerator2` with `rem1` and `denominator1` with `rem2`.

```
368 ; GCD of denominator1
369 ; and denominator2
370 mov ax, denominator1
371 mov number1, ax
372 mov ax, denominator2
373 mov number2, ax
374
375 call getGCD
376
377
378 ; denominator1 / GCD > rem1
379 mov ax, denominator1
380 div GCDValue
381 mov rem1, ax
382
383
384 ; denominator2 / GCD > rem2
385 mov ax, denominator2
386 div GCDValue
387 mov rem2, ax
388
389
390 ; numerator1 *= rem2
391 mov ax, rem2
392 mul numerator1
393 mov numerator1, ax
394
395
396 ; numerator2 *= rem1
397 mov ax, rem1
398 mul numerator2
399 mov numerator2, ax
400
401
402 ; denominator1 *= rem2
403 mov ax, rem2
404 mul denominator1
405 mov denominator1, ax
406
```

Figure 11: Code for Making the Denominators the Same

### Adding Numerators After Making the Denominators the Same

After the denominators of the two fractions became same, the program will add the first numerator and the second one as the numerator of result fraction, if the result of adding the numerators is greater than 65536 (more the 16 bit), then the program cannot process that number because we used 16-bit registers, so if the result is greater than 16 bits, the program will display an error message.

If we add 2 numbers in 16-bit register, and the result of adding is more than 16 bits, then the result will be less than the lowest of the two numbers. ([Check the outputs](#))

```
411 ; numerator1 += numerator2
412 mov ax, numerator1
413 mov number1, ax
414 mov ax, numerator2
415 add number1, ax
416
417
418 ;; if the result of adding numerator1 and
419 ;; numerator2 is less than the small number
420 ;; between them, that means that the result
421 ;; is greatest than 65536, that mean we cannot
422 ;; process it with our program so the program
423 ;; will print an error message
424
425
426 ;put smaller in number
427 mov ax, numerator1
428 cmp numerator2, ax
429 ja twoIsBigger
430
431 mov ax, numerator2
432 mov number, ax
433 jmp continue1
434
435
436 twoIsBigger:
437 mov number, ax
438 continue1:
439 ;;
440
441
442 ; check if the result is less than number
443 mov ax, number1
444 cmp number, ax
445 ja belowError
446
447
448 ; return value of number1 to numerator1
449 mov ax, number1
450 mov numerator1, ax
```

Figure 12: Code for Adding Numerators and Check error

## Simplifying the Result

If there are no errors, the program will store the fraction result as numerator and denominator in `numerator1` and `denominator1`. Then to simplify the result, the program finds the GCD of numerator and denominator, then we can find the simplest form by dividing each numerator and denominator by the DCG of them.

```
454 ; find GCD of numerator1 and denominator1
455 mov ax, denominator1
456 mov number2, ax
457
458 call getGCD
459
460
461 ; numerator1 /= GCD
462 mov ax, numerator1
463 div GCDValue
464 mov numerator1, ax
465
466 ; denominator1 /= GCD
467 mov ax, denominator1
468 div GCDValue
469 mov denominator1, ax
470
471
472
473
474
475 printFiveDigit numerator1
476 printStr msgSlash
477 printFiveDigit denominator1
```

Figure 13: Code for Simplifying the Result and Print it

# Procedures

## getGCD

This procedure calculates the GCD of the two values stored on `number1` and `number2` variables and stores the result in `GCDValue`.

I used The Euclidean Algorithm to find the GCD of the two numbers, The Euclidean Algorithm is a is an efficient method for computing the greatest common divisor (GCD) of two integers (numbers), the largest number that divides them both without a remainder.<sup>1</sup>

To use Euclid's algorithm, divide the smaller number by the larger number. If there is a remainder, then continue by dividing the smaller number by the remainder.

$$A \div B = Q1 \text{ remainder } R1$$

$$B \div R1 = Q2 \text{ remainder } R2$$

$$R1 \div R2 = Q3 \text{ remainder } R3$$

Continue this process until the remainder is 0 then stop. The divisor in the final step will be the greatest common factor.

For example, find the greatest common factor of 78 and 66 using Euclid's algorithm.

$$78 \div 66 = 1 \text{ remainder } 12$$

$$66 \div 12 = 5 \text{ remainder } 6$$

$$12 \div 6 = 2 \text{ remainder } 0$$

Thus, the greatest common factor is 6, since that was the divisor in the equation that yielded a remainder of 0.<sup>2</sup>

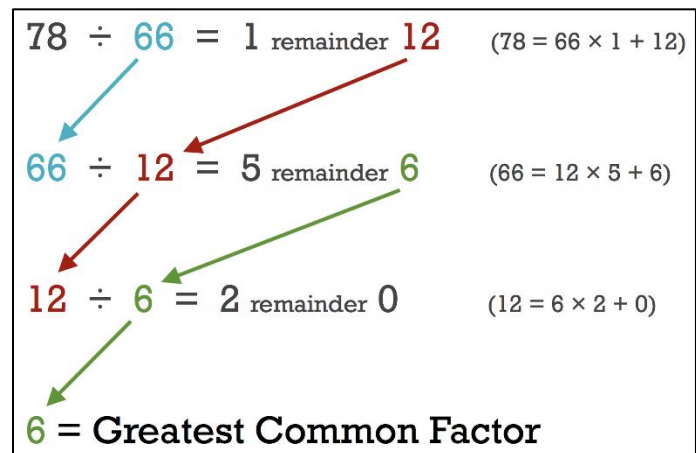


Figure 14: Example of finding GCD using Euclid's Algorithm

For more information about Euclid's algorithm, you can watch the video and visit the link below

Video: <https://www.youtube.com/watch?v=fwuj4yzoX1o>

Link: <https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm>

<sup>1</sup> [wikipedia.org](https://www.wikipedia.org)

<sup>2</sup> [inchcalculator.com](https://www.inchcalculator.com)

First of all, the program checks which number is greater than the other one, and moves the greater number to `number1` and the other to `number2`.

Then the value of `rem1` set as `number1`, and the value of `rem2` set as `number2`.

Our program will find the value of `rem1` mod `rem2`, and then save it in `rem1`, then it will find `rem2` mod the result of (`rem1` mod `rem2`) that stored in `rem1`, so it will swap `rem1` with `rem2` (after store the last remainder in `rem1`).

This process will keep repeating until the result of mod is zero, when it is, the value of GCD will be the last value of `rem1`.

```

637 getGCD proc
638     ;let number1 is the
639     ;greatest between
640     ;two numbers and
641     ;number2 is the
642     ;smallest
643     cmp number1, ax
644     ja noChange
645
646
647     ;swap
648     mov bx, number1
649     mov number1, ax
650     mov number2, bx
651
652     noChange:
653     ;end swapping
654
655
656
657     mov ax, number1
658     mov rem1, ax
659
660     mov ax, number2
661     mov rem2, ax
662
663
664     ; rem1%rem2>rem1
665     ; swap
666
667
668     GCD:
669     mov dx, 0
670     mov ax, rem1
671     div rem2
672     mov rem1, dx
673
674
675     ;swap
676     mov bx, rem1
677     mov ax, rem2
678     mov rem1, ax
679     mov rem2, bx
680     ;
681
682     cmp rem2, 0h
683     je exit
684
685     jmp GCD
686
687
688     exit:
689
690     mov ax, rem1
691     mov GCDValue, ax
692     ret
693 getGCD endp

```

Figure 15: getGCD Procedure Code

## scanThreeDigit

This procedure reads 3-digit numbers from user and save it in `number` variable.

```
701 scanThreeDigit proc
702     mov bx, 0
703     mov errIn, 0
704
705     mov cx, 3           ;to read 3 digit number
706
707     scanNum:
708
709         mov ah, 01h
710         int 21h
711
712
713
714         mov ah, 0
715         sub ax, 48      ; ASCII to DECIMAL
716
717         cmp ax, 0
718         jnb errorInFlag
719
720         cmp ax, 9
721         ja errorInFlag
722
723         rr:
724
725         mov dx, ax
726         mov ax, bx      ; Store the previous value in AL
727
728         mul ten          ; multiply the previous value with 10
729
730         add ax, dx       ; previous value + new value < after previous value is multiplied with 10 >
731         mov bx, ax
732
733         mov number, bx
734     loop scanNum
735     ret
736 scanThreeDigit endp
```

Figure 16: scanThreeDigit Procedure Code

First the value stored in `cx` set as the number of loops (equals the number of digits), in our case we need to read 3 digits so the value of `cx` set to 3.

The loop reads one digit each time. After read the digit, it will convert from ASCII to decimal, then the program checks if this digit isn't number (its value is more than 9 or less than 0), if so, the `errIn` flag variable will set as 1.

Then to make a number from multi digits, the program saves the last value, then multiply it with 10 and add the new entry to it, this process repeated until `cx` become zero (all loops finished).

# Outputs Samples

## Part1: Calculate GCD and LCM of Two Numbers

As you see, if the user enters a special characters or letters, a message displays that it's invalid number will appear, and asks the user to enter just numbers.

Also, if the user enters a number above 250, the program will display a message that tell the user that it's invalid number because it's above 250, and will ask user to enter number again, this will repeat until the user enters a valid number.

```
Hello!

Please Enter Number 1: 16+
This is INVALID Number, Please enter just numbers

Please Enter Number 1: 5d1
This is INVALID Number, Please enter just numbers

Please Enter Number 1: gdc
This is INVALID Number, Please enter just numbers

Please Enter Number 1: 965
This is INVALID Number, Please enter number between 0 and 250

Please Enter Number 1: 150
Please Enter Number 2: d66
This is INVALID Number, Please enter just numbers

Please Enter Number 2: 000
This is INVALID Number, Please enter number between 0 and 250

Please Enter Number 2: 100

GCD = 00050
LCM = 00300
```

Figure 17: Sample of Part1 Outputs (With Error Messages)

```
Hello!

Please Enter Number 1: 240
Please Enter Number 2: 190

GCD = 00010
LCM = 04560
```

Figure 18: Sample of Part1 Outputs (Without Invalid Inputs)

## Part2: Find the Result of Adding Two Fraction Using GCD

```
Example of fractions addition, Enter two fractions
12+
This is INVALID Number, Please enter just numbers

120/g69
This is INVALID Number, Please enter just numbers

120/135 + 452
This is INVALID Number, Please enter number between 0 and 250

120/135 + 215/000
This is INVALID Number, Please enter number between 0 and 250

120/135 + 215/090 = 00059/00018
```

Figure 19: Sample of Part2 Outputs (With Invalid Input Errors)

The program asks user to enter fractions, user will enter numerator then denominator for the first then the second number respectively, if the user enter an invalid number, the program will display the previous numbers entered then it will wait user to enter the number again (Try the code for a better understanding).

If the user enters numbers like  $211/223 + 199/227$ , the two denominators are prime numbers, that means when we try to make the denominators same, it will be  $47897/50621 + 50621/50621$ , and when we try to add them, the result will be  $98518/50621$ , but the problem is the numerator cannot store in 16-bit register because it's too big, and that will cause an incorrect result, so the program will display an error message as follow.

```
Example of fractions addition, Enter two fractions
211/223 + 199/227 =

Error, the result of adding first numerator with the second one after reduction
of fractions is greater than 65536 and It cannot be processed
```

Figure 20: Sample of Part2 Outputs (With Big Number Error)

## Full Output

```
Hello!

Please Enter Number 1: 100
Please Enter Number 2: 150

GCD = 00050
LCM = 00300

Example of fractions addition, Enter two fractions
128/036 + 125/115 = 00961/00207

Bye!
Obada Tahayna 1191319
```

*Figure 21: Full Output of The Program*