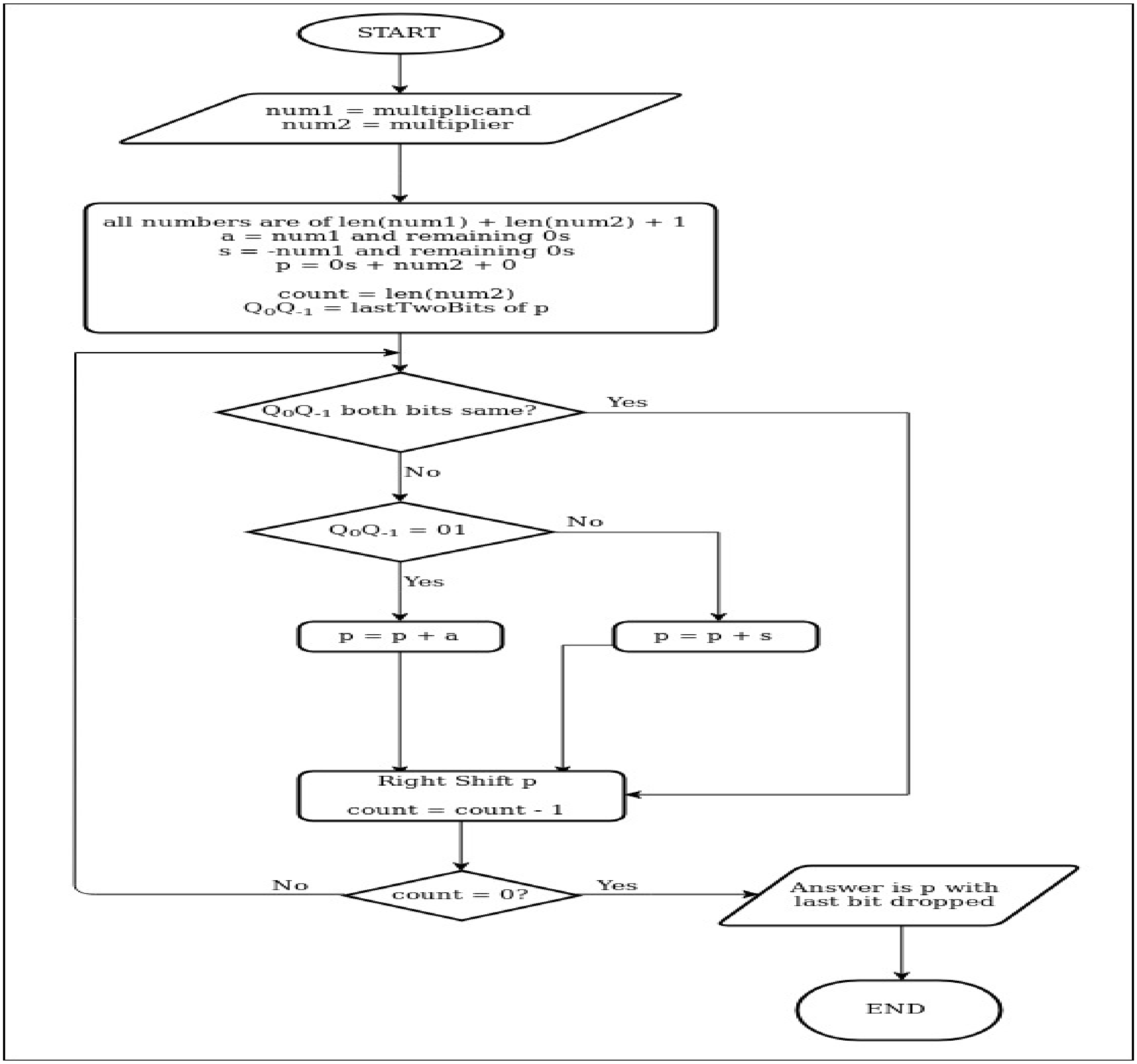
Computer Organization
Project - 2
Implementation of Booth's Algorithm

By: Vibhu Agrawal
2018116

Multiplication and division are two of the four basic operations often performed in computations. Booth's algorithm provides implementation for these two operations so that the time complexity and hardware complexity can be reduced.

Booth's Multiplication:


START
num1 = multiplicand
num2 = multiplier
all numbers are of len(num1) + len(num2) + 1
a = num1 and remaining 0s
s = -num1 and remaining 0s
p = 0s + num2 + 0
count = len(num2)
Q_0Q_{-1} = lastTwoBits of p
Q_0Q_{-1} both bits same?
Yes
No
Q_0Q_{-1} = 01
No
Yes
p = p + a
p = p + s
Right Shift p
count = count - 1
No
count = 0?
Yes
Answer is p with last bit dropped
END

Algorithm Used for Multiplication

1. Three numbers A, S and P (all of number of bits in multiplier + number of bits in multiplicand + 1) are initialized as follows:

A: multiplicand + all 0s
S: negative of multiplicand + all 0s
P: number of bits in multiplicand times 0 + multiplier + 0

2. The last two bits of P are compared. The following cases occur:

- If they are 00 or 11: move on to next step.
- If they are 01: P = P + A
- If they are 10: P = P + S

3. P is right shifted.

4. Steps 2 and 3 are carried on the same number of times as the number of bits in multiplier.

5. In the end, last bit of P is dropped, and P is the result of multiplication.

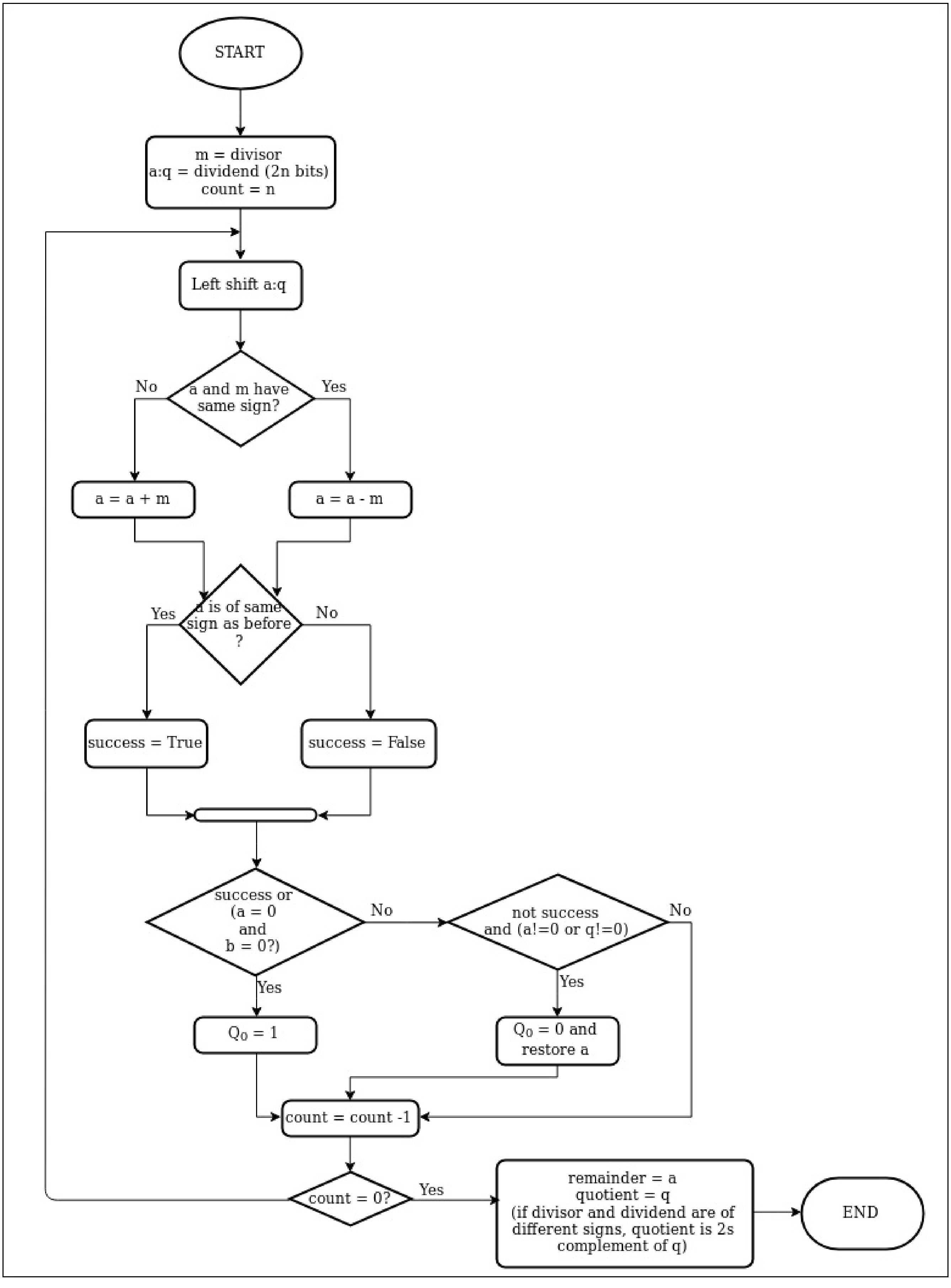
This algorithm uses the fact that if two 00 or two 11 occur simultaneously in the multiplier, the value of the product is just multiplied by a factor of 2.

Complexity Analysis:

If n is the number of bits in the binary of multiplier, the time complexity turns out to be O(n)*Complexity of (Addition + bit shift). O(n) factor comes from the fact that the steps 2 and 3 listed above in algorithm are carried out n number of times.
Complexity of Addition and bit shift is also generally n.

Hence the net run time complexity approaches $O(n^2)$.

Division Algorithm:



Algorithm Used For Division:

1. The following registers are initialized.
   M = divisor
   A:Q = dividend with sign extension (total 2n bits)

2. A:Q is left shifted.
3. If A and M are of same sign, A = A - M, else A = A + M
4. If sign of A is same as before or A=0 and Q=0, then make last bit of Q as 1. If sign of A is different and A≠0 or Q≠0, then restore the previous value of A, and make least significant bit of Q as 0.
5. Repeat steps 2,3,4 for n times(the number of bits in dividend.)
6. A is the remainder. Q has the quotient. If signs of divisor and dividend were initially different, then the twos complement of Q is the quotient.

Complexity Analysis:

If n is the number of bits in dividend, the time complexity is O(n)*Complexity of (left shift + addition + subtraction). The complexity of second term approximates O(n).

So the total complexity is roughly $O(n^2)$.

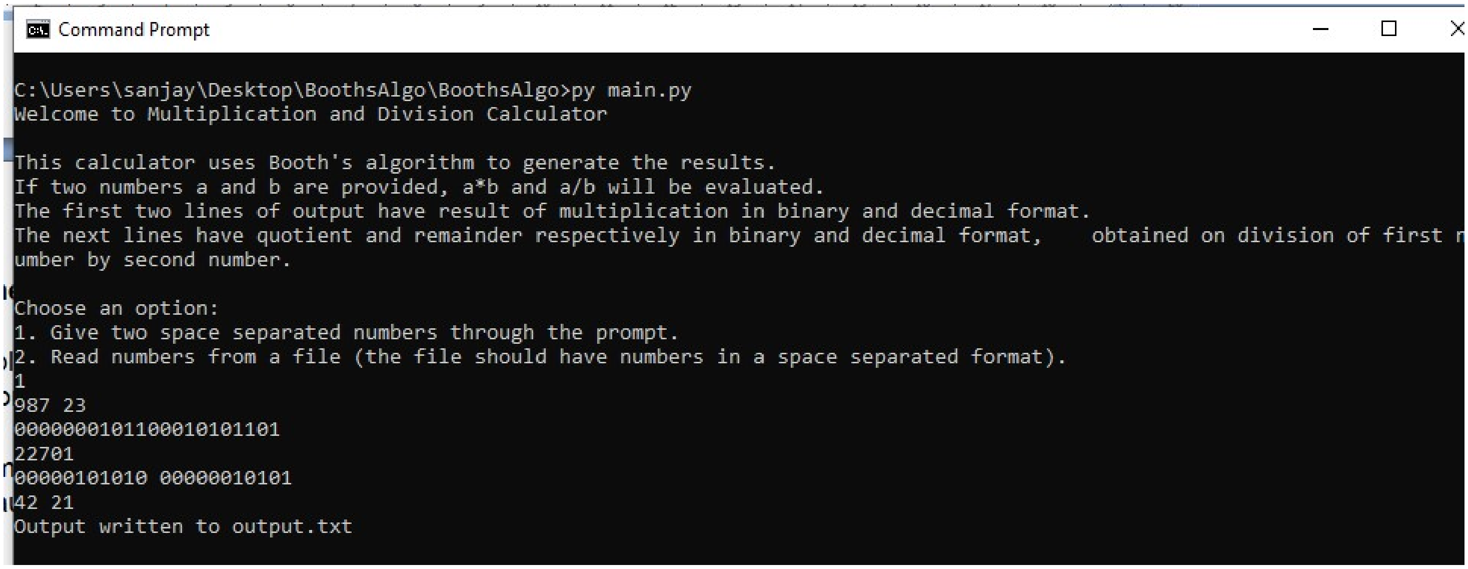
Implementation Details:

The implementation has been done in Python3.
Code consists of two files:

1. main.py
2. auxFunctions.py

When main.py is invoked, it prompts user for the input of two numbers, a and b, which can be provided through terminal or some file. The result evaluated (a*b and a/b) is displayed on terminal and stored in output.txt in same folder.

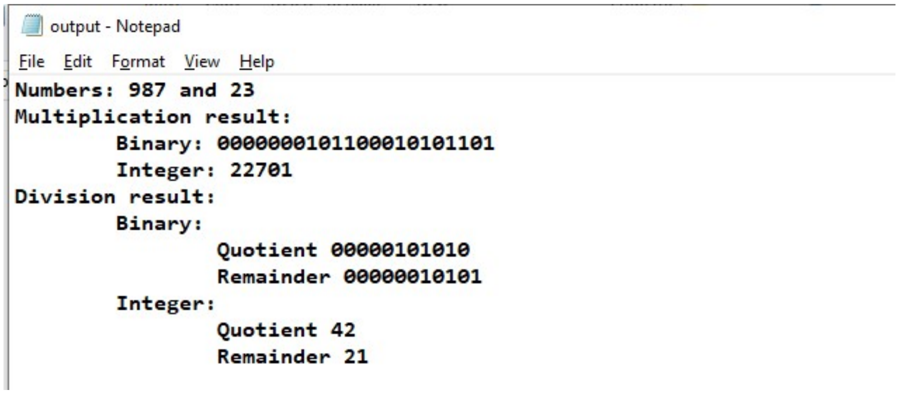
Test Cases:



```
C:\Users\sanjay\Desktop\BoothsAlgo\BoothsAlgo>py main.py
Welcome to Multiplication and Division Calculator

This calculator uses Booth's algorithm to generate the results.
If two numbers a and b are provided, a*b and a/b will be evaluated.
The first two lines of output have result of multiplication in binary and decimal format.
The next lines have quotient and remainder respectively in binary and decimal format,    obtained on division of first n
umber by second number.

Choose an option:
1. Give two space separated numbers through the prompt.
2. Read numbers from a file (the file should have numbers in a space separated format).
1
987 23
0000000101100010101101
22701
00000101010 00000010101
42 21
Output written to output.txt
```



```
Numbers: 987 and 23
Multiplication result:
        Binary: 0000000101100010101101
        Integer: 22701
Division result:
        Binary:
                Quotient 00000101010
                Remainder 00000010101
        Integer:
                Quotient 42
                Remainder 21
```

Other test cases:

1. 97 0
2. 0 97

3. -50 9
4. -50 -9
5. 1000 1000