

Laboratory 1

Tiutiu Natan-Gabriel

Introduction

I will present 3 algorithms for computing GCD: Dijkstra's Algorithm, Euclidean Algorithm and Binary Algorithm. Because we work in Haskell, each of these algorithms will be able to compute the gcd for very large numbers.

However, supposing that we wouldn't work in a language that supports computations with large numbers I also implemented Dijkstra's Algorithm for such a case. I treated each input number as a string, then split this string in a list of characters and then transforming each character into the corresponding digit. For Dijkstra's algorithm I need to compare numbers and subtract numbers. So I implemented 2 functions which can compare and subtract lists-our representation for numbers (sometimes I work with lists in reversed order-for example, doing so, it's easier to subtract 2 lists). I named this algorithm `dijkstraImprovedAlgorithm`.

You call binary algorithm using "binary number1 number2", euclidean algorithm using "euclidean number1 number2" and dijkstra algorithm using "dijkstra number1 number2". For `dijkstraImprovedAlgorithm` just use "dijkstraImproved".

1. GCD Dijkstra's Algorithm for 2 natural numbers

Description:

As long as $a \neq b$, we call recursively `dijkstraImprovedAlgorithm(a-b, b)`, if $a > b$ or `dijkstraImprovedAlgorithm(a, b-a)`, else.

Here is the mathematical model:

$$dijkstra(a, b) = \begin{cases} a, & \text{if } a = b \\ dijkstra(a - b, b), & \text{if } a > b \\ dijkstra(a, b - a), & \text{else} \end{cases}$$

The proof of correctness is based on the following property:

$gcd(a,b)=gcd(a-b,b)$, if $a > b$ (4) $gcd(a,b)=gcd(a,b-a)$, if $b > a$ (5)

Proof of this property:

We will prove just (4), as (5) can be proven if we let $b=a$ and $a=b$ in (4). Let $d=gcd(a,b) \Rightarrow d|a$ and $d|b$ (6), so $d|a-b$ (7). Suppose that $gcd(a-b,b)=e \Rightarrow$ because d is a common divisor of b and $a-b$ we will have $e|d$ (11). We also have that $e|a-b$ and $e|b$ (8), so $e|a$ (9). Using (8) and (9) we have that e is a common divisor of a and $b \Rightarrow d|e$ (10)

Using (10) and (11) we have that $d=e$, that is $gcd(a,b)=gcd(a-b,b)$.

```
<<dijkstra>>=
--Dijkstra algorithm
dijkstra :: Integer->Integer->Integer
dijkstra a b =
    if (a==b)
    then a
    else if (a>b)
    then (dijkstra (a-b) b)
    else (dijkstra a (b-a) )
@
```

2.GCD Euclidean Algorithm for 2 natural numbers

Description:

As long as b , which is the second element is not equal to 0, we will call recursively $euclidean(b,a \% b)$. We can observe that always second element becomes the first element in the new recursive call. If we would not inverse the position of the elements, we would get stuck (in some cases).

Ex.: $a = 1000, b = 100$ $euclidean(1000,100) = euclidean(1000,100) = \dots = euclidean(1000,100) = \dots$

Here is the mathematical model:

$$euclidean(a,b) = \begin{cases} euclidean(b, a \% b), & \text{if } b > 0 \\ a, & \text{else} \end{cases}$$

The proof of correctness is based on the following lemma:

If $a = c \pmod{b}$ (1), then $(a,b) = (c,b)$ (3)

Proof of this lemma:

From (1) we have $b|a-c$, so there is a y such that $by=a-c$. If there is a d such that d divides a and b , then it will also divide $c=a-by$. \Rightarrow any divisor of a and b is a divisor of c and b . (2) Suppose $(a,b)=x$ and $(c,b)=y$. Using (2) we have that $x|y$ and $y|x$, so we have that $(a,b)=(c,b)$.

For our problem we use this:

We have $a=a \% b \pmod b \Rightarrow$ using lemma (3) we have that $(a,b)=(a \% b,b)$

```
<<euclidean>>=
euclidean a b =
  if (b>0)
    then (euclidean b (mod a b) )
    else a
@
```

3.GCD Binary algorithm (Stein's algorithm) for 2 natural numbers

Description:

This is an improved version of Dijkstra's algorithm. As long as at least one of the elements is even we divide one (or two) elements by 2. If both are even then we use that $\gcd(2u, 2v) = 2 \cdot \gcd(u, v)$. If just one is even then we use $\gcd(2u, v) = \gcd(u, v)$. After both operands become odd (if it is the case) we will have an algorithm identical to Dijkstra's algorithm.

Attention! As long as there are even numbers, we will not divide them by 2, but instead we will shift the elements (in binary representation) one position to the right, which is a very cheap operation, compared to division. This is why I called this algorithm an improved version of Dijkstra's algorithm.

Here is the mathematical model:

$$\text{binary}(a, b) = \begin{cases} a, & \text{if } a = b \\ b, & \text{if } a = 0 \\ a, & \text{if } b = 0 \\ 2 * \text{binary}(a/2, b/2), & \text{if } a \text{ and } b \text{ even} \\ \text{binary}(a/2, b), & \text{if just } a \text{ is even} \\ \text{binary}(a, b/2), & \text{if just } b \text{ is even} \\ \text{dijkstra}(a-b, b), & \text{if } a > b \\ \text{binary}(a-b, b), & \text{if } a > b \\ \text{binary}(b-a, a), & \text{else} \end{cases}$$

```

<<binary>>=
--Binary algorithm (Stein's algorithm)
binary a b =
    if (a==b)
        then a

    else
        if (a== 0)
            then b

        else
            if ( (andBitwise (complement a) 1)==1 ) then
                (if ( (andBitwise b 1)==1 )
                    then (binary (shiftR a 1) b)
                    else
                        ( shiftL (binary (shiftR a 1) (shiftR b 1)) 1 ) )

            else
                if ( (andBitwise (complement b) 1)==1 )
                    then (binary a (shiftR b 1))
@

```

So,from now on we will have 2 odd numbers,if we will reach this point of the program(last 2 cases).As long as $a \neq b$,we call recursively $\text{binary}(a-b,b)$, if $a > b$ or $\text{binary}(a,b-a)$,else.

Here I wil add the proof of corectness for last 2 cases of binary algorithm (same proof as for Dijkstra's algorithm), which is based on the following property:

$\text{gcd}(a,b)=\text{gcd}(a-b,b)$, if $a > b$ (4) $\text{gcd}(a,b)=\text{gcd}(a,b-a)$, if $b > a$ (5)

Proof of this property:

We will prove just (4),as (5) can be proven if we let $b=a$ and $a=b$ in (4) . Let $d=\text{gcd}(a,b) \Rightarrow d|a$ and $d|b$ (6) ,so $d|a-b$ (7) .Suppose that $\text{gcd}(a-b,b)=e \Rightarrow$ because d is a common divisor of b and $a-b$ we will have $e|d$ (11).We also have that $e|a-b$ and $e|b$ (8),so $e|a$ (9).Using (8) and (9) we have that e is a common divisor of a and $b \Rightarrow d|e$ (10)

Using (10) and (11) we have that $d=e$,that is $\text{gcd}(a,b)=\text{gcd}(a-b,b)$.

```

<<binary>>=
    else
        if (a>b)
            then (binary (a-b) b )
        else
            (binary (b-a) a)
@

```

We also define the “and” function for 2 integers

```
<<andBitwise>>=
andBitwise :: Int -> Int -> Int
andBitwise a b = a .&. b
@
```

4.GCD Dijkstra's Algorithm for 2 natural numbers supposing that Haskell would not support operations with large numbers

This is the wrapper function for the main function `dijkstraImprovedAlgorithm`, which computes the ged of 2 natural numbers.(this function will ask you to input this 2 numbers).

```
<<dijkstraImproved>>=
dijkstraImproved=do
    print "x="
    x <- getLine
    print "y="
    y <- getLine
    print "GCD of x and y is:"
@
```

At the end this function will show you the result and the total execution time.

```
<<dijkstraImproved>>=
    start <- getCurrentTime
    print (intListToString (reverseL (dijkstraImprovedAlgorithm (reverseL (inputToIntList x)
    stop <- getCurrentTime
    print $ diffUTCTime stop start
@
```

Function `intListToString` receives a list of integers and returns a string formed from concatenating the integers. `(show x)` transforms an integer in a string.

```
<<intListToString>>=
intListToString (x:rest) = (show x) ++ (intListToString rest)
intListToString rest = []
@
```

Function `stringToStringList` receives a string and returns a list of strings(characters) containing each character from initial string.

```
<<stringToStringList>>=
stringToStringList (x:rest) = [x]:(stringToStringList rest)
stringToStringList rest = []
@
```

Function `stringToInt` receives a string and returns that string transformed into an integer.

```
<<stringToInt>>=
stringToInt a=read a::Int
@
```

Function `stringListToIntList` receives a list of strings and returns a list formed of each string from given list transformed into an integer.

```
<<stringListToIntList>>=
stringListToIntList (x:rest) = (stringToInt x):(stringListToIntList rest)
stringListToIntList rest = []
@
```

Function `inputToIntList` is a wrapper which transforms the string read from keyboard into a list of strings(characters) and afterwards transform this list of strings into a list of integers.

```
<<inputToIntList>>=
inputToIntList x= stringListToIntList (stringToStringList (x))
@
```

GCD Dijkstra's Algorithm for 2 large natural numbers

Description:

As long as $a \neq b$, we call recursively `dijkstraImprovedAlgorithm(a-b,b)`, if $a > b$ or `dijkstraImprovedAlgorithm(a,b-a)`, else.

$$dijkstraImprovedAlgorithm(a,b) = \begin{cases} a, & \text{if } a = b \\ dijkstraImprovedAlgorithm(a-b,b), & \text{if } a > b \\ dijkstraImprovedAlgorithm(a,b-a), & \text{else} \end{cases}$$

The proof of corectness is based on the following property:

$$gcd(a,b)=gcd(a-b,b), \text{ if } a > b \quad (4) \quad gcd(a,b)=gcd(a,b-a), \text{ if } b > a \quad (5)$$

Proof of this property:

We will prove just (4), as (5) can be proven if we let $b=a$ and $a=b$ in (4) . Let $d=gcd(a,b) \Rightarrow d|a$ and $d|b$ (6) ,so $d|a-b$ (7) .Suppose that $gcd(a-b,b)=e \Rightarrow$ because d is a common divisor of b and $a-b$ we will have $e|d$ (11).We also have that $e|a-b$ and $e|b$ (8),so $e|a$ (9).Using (8) and (9) we have that e is a common divisor of a and $b \Rightarrow d|e$ (10)

Using (10) and (11) we have that $d=e$,that is $gcd(a,b)=gcd(a-b,b)$.

```
<<dijkstraImprovedAlgorithm>>=
```

```

-- Dijkstra algorithm for large numbers
dijkstraImprovedAlgorithm a b =
    if (a==b)
        then a
    else if ((compareLists (reverseL a) (reverseL b))== (reverseL a))
        then (dijkstraImprovedAlgorithm (subtractListsAndEliminate0 a b 0) b)
    else (dijkstraImprovedAlgorithm a (subtractListsAndEliminate0 b a 0) )

```

@

Function subtractListsAndEliminate0 calls function subtractLists, then reverse the list, in order to be easier to eliminate the 0's in front of the number and after all these steps, it reverse the list again (initial form of the list)

Attention! We represent numbers as lists of digits, in reversed order (doing so, the subtraction is performed easier)

```

<<subtractListsAndEliminate0>>=
subtractListsAndEliminate0 x y r =reverseL (eliminate0InFrontOfNumber (reverseL (subtractL

```

Function eliminate0InFrontOfNumber eliminates all 0's in front of a given number (represented as a list). So, as long as the first digit is 0, we call this function recursively for the tail. When the first digit is != 0 we simply return the list.

```

<<eliminate0InFrontOfNumber>>=
eliminate0InFrontOfNumber (x:resx)=
    if ( x==0)
        then (eliminate0InFrontOfNumber resx)
    else (x:resx)

```

@

Function subtractLists subtracts 2 lists of digits, which represent 2 reversed integers. We will use x for 1st number, y for the 2nd one and r for the carry. First number will be always larger than the second number because before calling this function in dijkstraImprovedAlgorithm we compare this 2 numbers, as set the larger number on the first position.

We have some cases:

1. numbers have length 1 and first digit would be 0 => then we return [], in order to avoid 0's in the front of the number

2. numbers have length 1 and first digit would't be 0 => then the 1st digit will be x-y-r

3. y has length 1 and x-y-r >= 0 => we will return (x-y-r) U tail of x

4. y has length 1 and x-y-r < 0 => we will return (x-y-r+10) U subtractLists(tail of x, [0], 1). Because we cannot use for y an empty list, we use for y [0], which will have the same effect.

5. if $x-y-r \geq 0$ (and both x and y have more than 1 elements) \Rightarrow the carry will be 0 and we will return $(x-y-r) \cup \text{subtractLists}(\text{tail of } x, \text{tail of } y, 0)$
6. if $x-y-r < 0$ (and both x and y have more than 1 elements) \Rightarrow the carry will be 1 and we will return $(x-y-r+10) \cup \text{subtractLists}(\text{tail of } x, \text{tail of } y, 1)$

Mathematical model:

$$\text{subtractLists}(x_1, x_2 \dots x_n; y_1, y_2, \dots, y_m) = \begin{cases} [], \text{ if } m = n = 1 \text{ and } x - y - r = 0 \\ [x - y - r], \text{ if } m = n = 1 \\ [x - y - r] \cup [x_2 \dots x_n], \text{ if } x - y - r \geq 0 \text{ and } m = 1 \\ [x - y - r + 10] \cup \text{subtractLists}(x_2 \dots x_n, [0], 1), \text{ if } x - y - r < 0 \\ [x - y - r] \cup \text{subtractLists}(x_2 \dots x_n, y_2 \dots y_n, 0), \text{ if } x - y - r \geq 0 \\ [x - y - r + 10] \cup \text{subtractLists}(x_2 \dots x_n, y_2 \dots y_n, 1), \text{ if } x - y - r < 0 \end{cases}$$

```
<<subtractLists>>=
subtractLists :: [Int] -> [Int] -> Int -> [Int]
subtractLists (x:resx) (y:resy) r =
    if ( (length resx)==0 && (length resy)==0 && x-y-r==0)
    then []
    else if ( (length resx)==0 && (length resy)==0)
    then [x-y-r]
    else if ( x-y-r>=0 && (length resy)==0)
    then (x-y-r):resx
    else if ( x-y-r<0 && (length resy)==0)
    then (x-y-r+10):(subtractLists resx [0] 1)
    else if ( x-y-r>=0)
    then (x-y-r):(subtractLists resx resy 0)
    else if ( x-y-r<0)
    then (x-y-r+10):(subtractLists resx resy 1)

    else [-1]
@
```

Function compareEqualLists compares two lists of equal size. The lists represents 2 natural numbers represented in reversed order. The function will return the list representing the larger number.

As long as the head of x = head of y we return x \cup compareEqualLists(tail of x, tail of y). Whenever we have that head of x < head of y or head of x > head of y, we return y or x.

Mathematical model:

$$compareEqualLists(x1, x2...xn; y1, y2, ..., yn) = \begin{cases} x1...xn, & \text{if } x1 > y1 \\ y1...yn, & \text{if } x1 < y1 \\ x1 \cup compareEqualLists(x2...xn; y2, ..., yn), & \text{else} \end{cases}$$

```
<<compareEqualLists>>=
compareEqualLists (x:resx) (y:resy)=
  if ( x> y)
    then (x:resx)
  else if ( x < y)
    then (y:resy)
  else [x] ++ compareEqualLists resx resy
@
```

Function compareLists compares 2 lists. The mathematical model is:

$$compareLists(x, y) = \begin{cases} x, & \text{if } length(x) > length(y) \\ y, & \text{if } length(y) > length(x) \\ compareEqualLists(x, y), & \text{else} \end{cases}$$

```
<<compareLists>>=
compareLists x y=
  if (( length x)>(length y))
    then x
  else if (( length x)<(length y))
    then y
  else (compareEqualLists x y)
@
```

Function reverseL reverses a given list. The mathematical model is:

$$reverseL(x1, x2, ...xn) = \begin{cases} [], & \text{if } n = 0 \\ reverseL(x2, ...xn) \cup x1, & \text{else} \end{cases}$$

```
<<reverseL>>=
reverseL [] = []
reverseL (x:xs) = reverseL xs ++ [x]
@
```

This functions will be used to compute execution time:

```
<<dijkstraTime>>=
dijkstraTime a b=do
  start <- getCurrentTime
  print (dijkstra a b)
  stop <- getCurrentTime
```

```

    print $ diffUTCTime stop start
@

<<euclideanTime>>=
euclideanTime a b=do
    start <- getCurrentTime
    print (euclidean a b)
    stop <- getCurrentTime
    print $ diffUTCTime stop start
@

<<binaryTime>>=
binaryTime a b=do
    start <- getCurrentTime
    print (binary a b)
    stop <- getCurrentTime
    print $ diffUTCTime stop start
@

<<*>>=
import Data.Bits
import Data.Time

<<dijkstra>>
<<euclidean>>
<<binary>>
<<andBitwise>>

<<dijkstraImproved>>
<<intListToString>>
<<stringToStringList>>
<<stringToInt>>
<<stringListToIntList>>
<<inputToIntList>>
<<dijkstraImprovedAlgorithm>>
<<subtractListsAndEliminate0>>
<<eliminate0InFrontOfNumber>>
<<subtractLists>>
<<compareEqualLists>>
<<compareLists>>
<<reverseL>>

<<dijkstraTime>>
<<euclideanTime>>
<<binaryTime>>
@

```

Performance table

Binary algorithm cannot compute gcd for very large numbers because it uses the shifting of integers.

Numbers	Dijkstra	Euclidean	Binary	Result(should be the same)
120,85	0.0002596s	0.001373s	0.0002359s	5
1025,2300	0.0006341s	0.0006089s	0.0003399s	25
28766,95398	0.0002381s	0.0002177s	0.0005736s	2
1438777,2501952	0.0002925s	0.0002282s	0.0004395s	1
387738773877 250125012501	0.0011381s	0.0011578s	0.002062s	100010001
1234544323454 1234323454323	0.0051449s	0.0002681s	0.0002877s	1
45443234541235 43234543231236	0.0002215s	0.0004839s	0.0007217s	1
840058424028554 238058202380559	0.0037016s	0.0002123ss	0.0005522s	1
4920194863458060 2345118523597022	0.00024s	0.0005206s	0.0004915s	2
53689085406123764 57947095123457468	0.000631s	0.0002302s	0.0003754s	4
2^31 2^32	0.0014088s	0.001273s	0.0015668s	36028797018963968