Exploring the Euclidean Algorithm

Stephen Capps, Sarah Sahibzada, & Taylor Wilson

Texas A&M University

Supurvisor: Sara Pollock

July 6, 2015

Overview

The Euclidean Algorithm

The Euclidean Algorithm is used to find the Greatest Common Divisor between any pair of whole numbers p, q such that $\mathrm{p} > \mathrm{q}$. It follows that

$$p = n_1 * q + r_1$$

$$q = n_2 * r_1 + r_2$$

.

.

.

$$r_{k-1} = n_{k+1} * r_k$$

Where

$$r_k = \gcd(\mathrm{p}, \mathrm{q}).$$

For example, here is the gcd(42, 36):

$$\gcd(42, 36) = 6 :$$

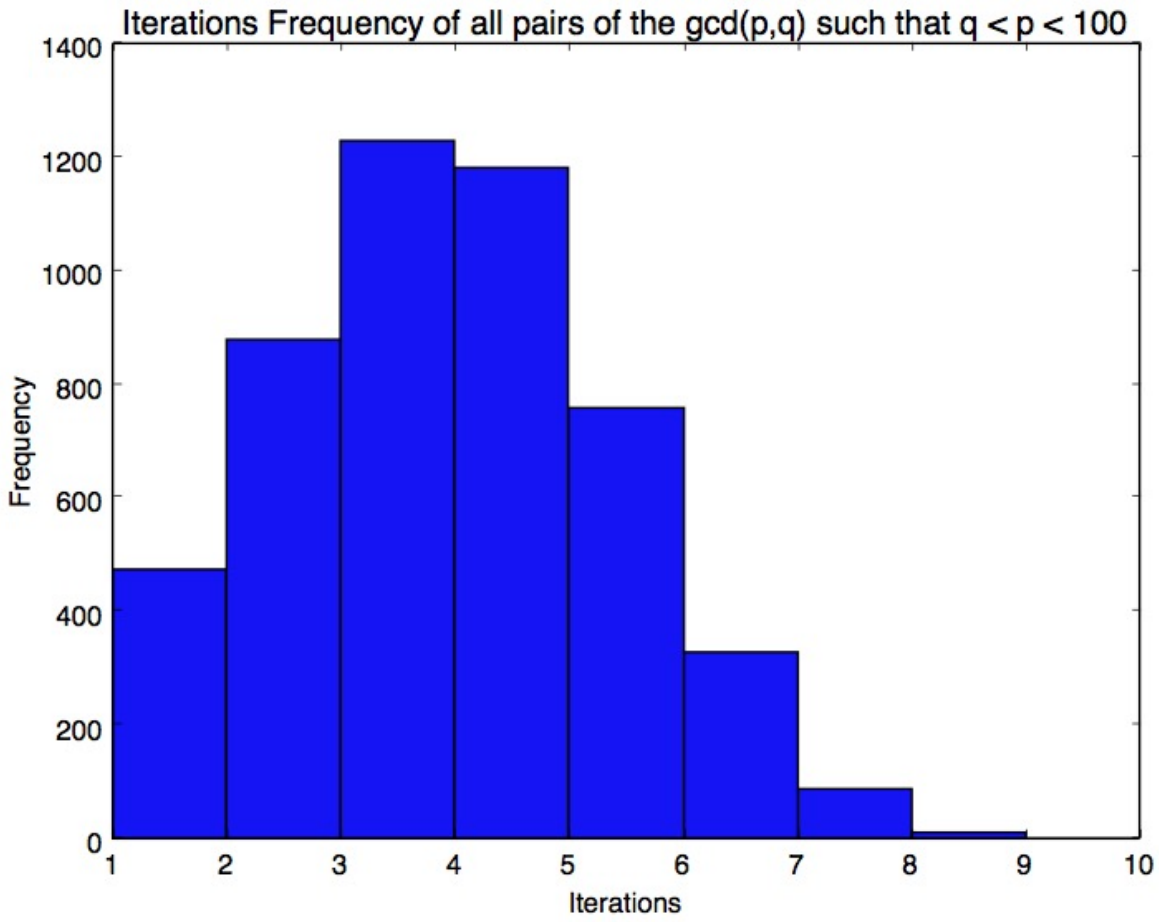
$$42 = 1 * 36 + 6 \tag{1}$$

$$36 = 6 * 6 + 0 \tag{2}$$

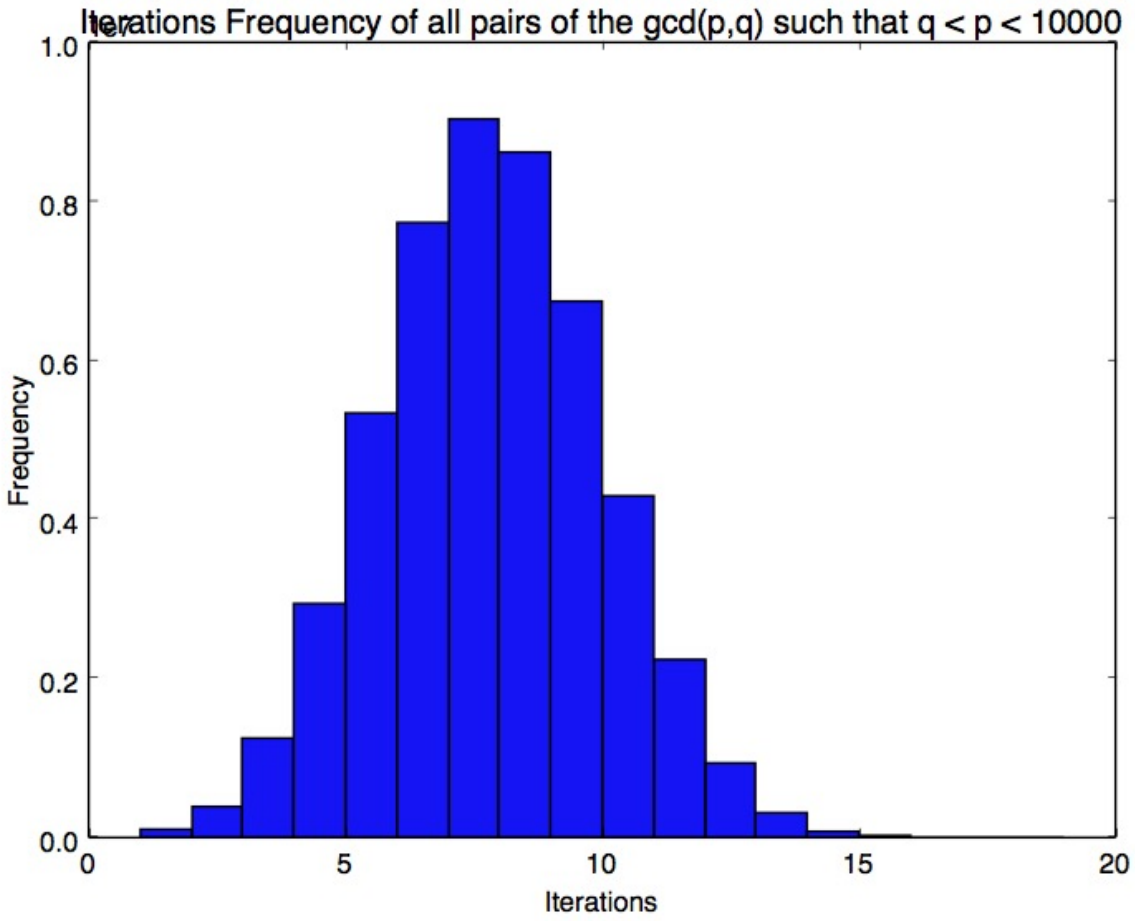
As you can see, it took 2 iterations to complete the algorithm. This is what we will explore. Here are some more gcds and their iterations:

$\gcd(p, q) = d$	Iterations
$\gcd(689, 456) = 1$	6
$\gcd(78, 45) = 3$	5
$\gcd(8394, 238) = 2$	7

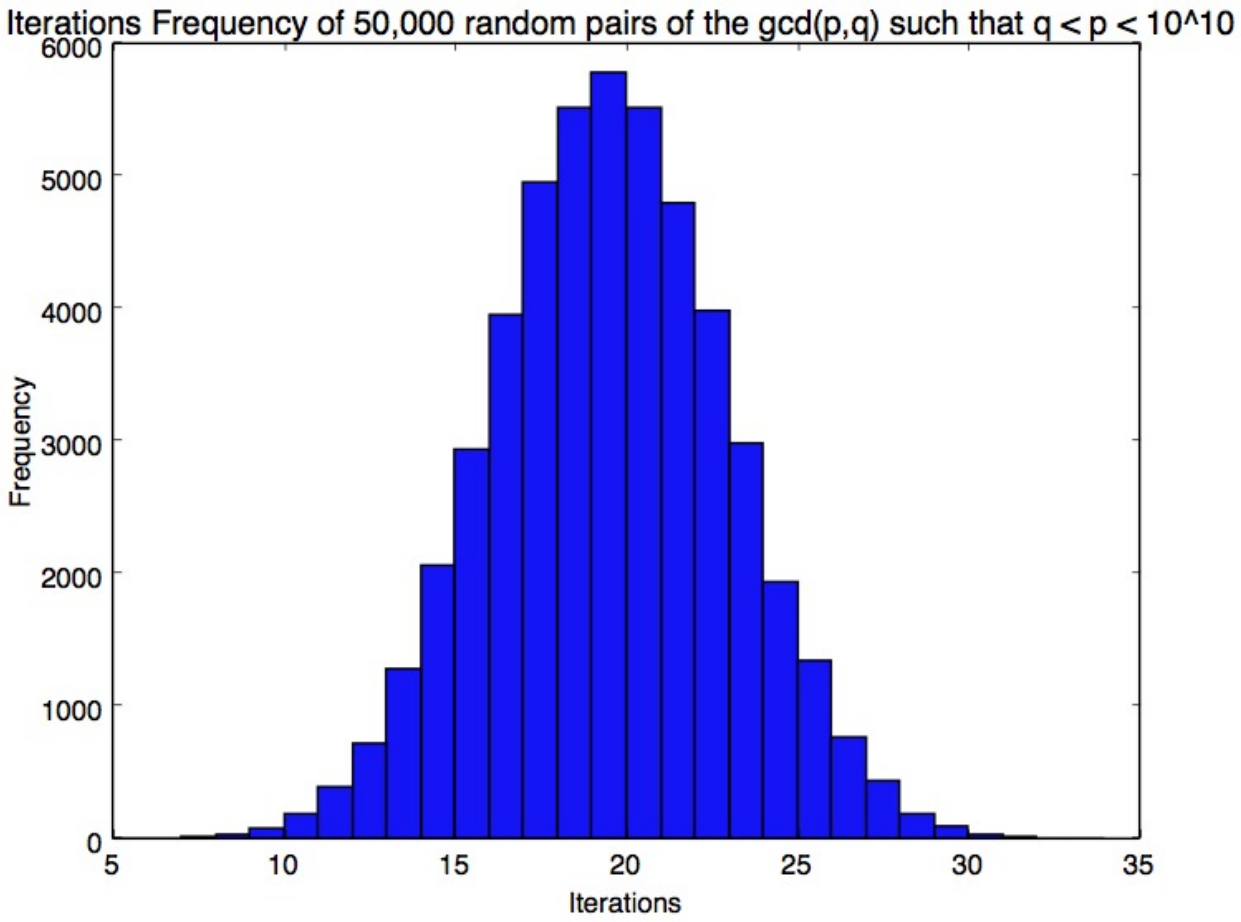
Next, we decided to explore the distributions of these iterations: Do most pairs take many iterations? What is the distribution?
The following graphs are the answer to these questions.


Iterations Frequency of all pairs of the gcd(p,q) such that q < p < 100
Frequency
Iterations
0
200
400
600
800
1000
1200
1400
1
2
3
4
5
6
7
8
9
10

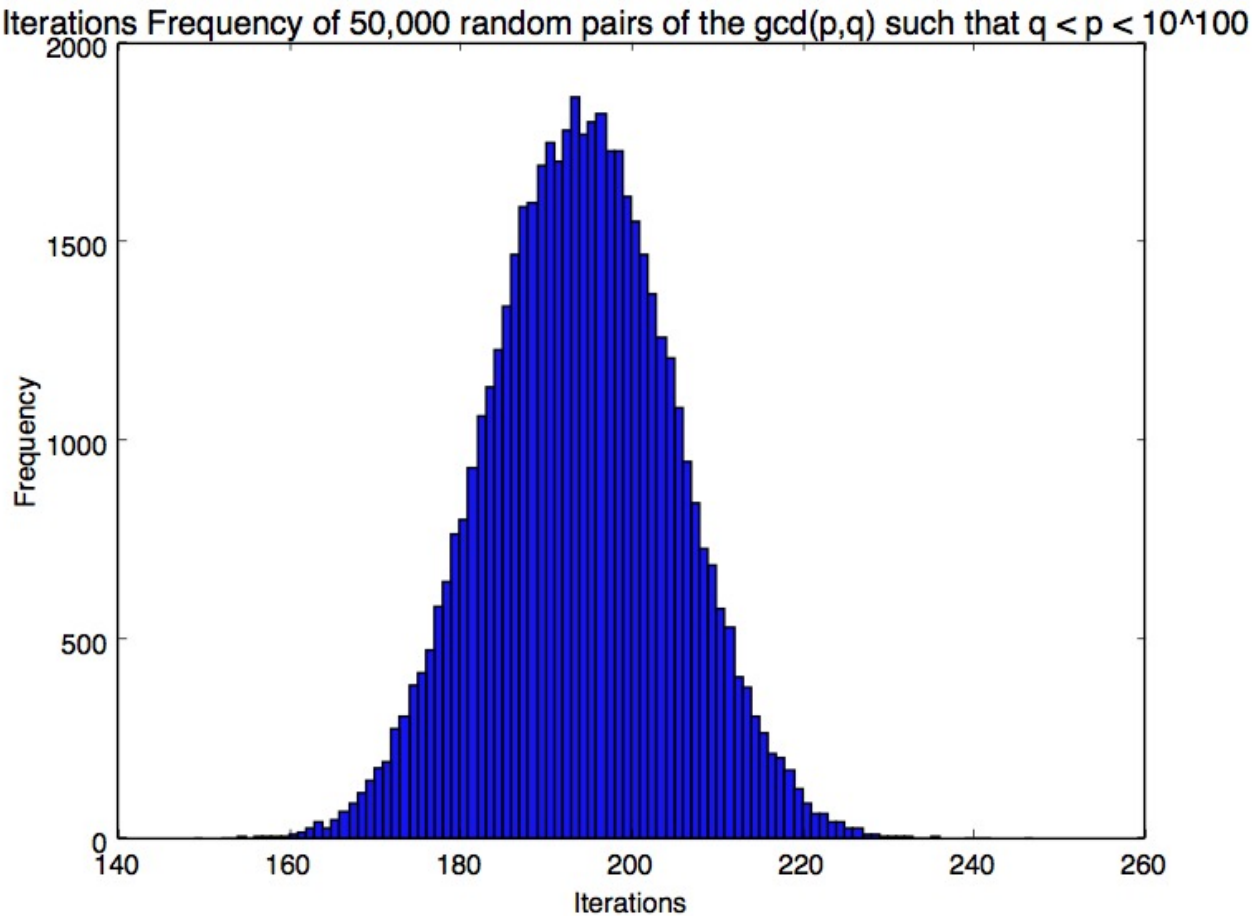
(Figure 1)


Iterations Frequency of all pairs of the gcd(p,q) such that q < p < 10000
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Iterations

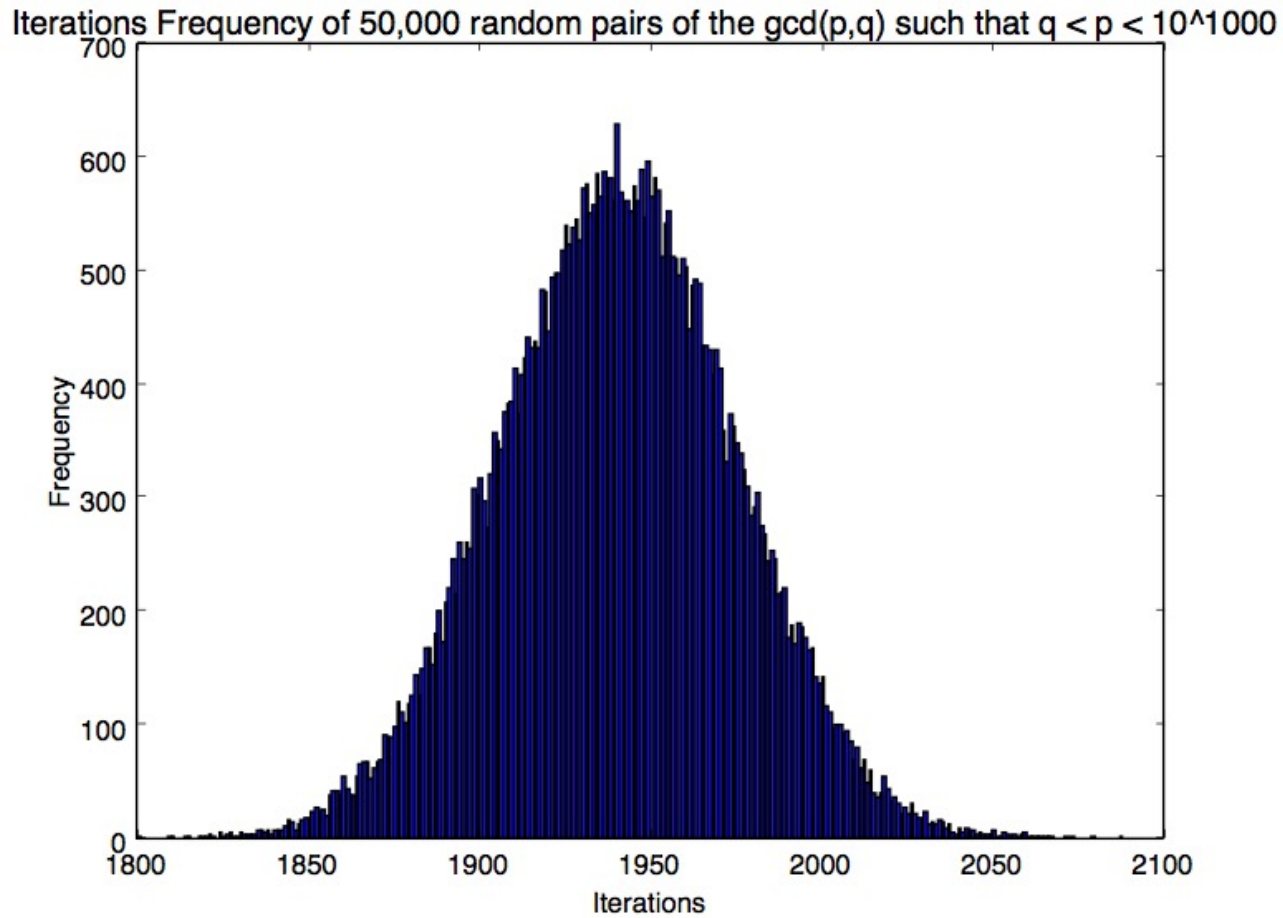
(Figure 2)


Iterations Frequency of 50,000 random pairs of the gcd(p,q) such that q < p < 10^10
Frequency
Iterations
0
1000
2000
3000
4000
5000
6000
5
10
15
20
25
30
35

(Figure 3)


Iterations Frequency of 50,000 random pairs of the gcd(p,q) such that q < p < 10^100
Frequency
Iterations
0
500
1000
1500
2000
140
160
180
200
220
240
260

(Figure 4)


Iterations Frequency of 50,000 random pairs of the gcd(p,q) such that q < p < 10^1000
Frequency
0
100
200
300
400
500
600
700
1800
1850
1900
1950
2000
2050
2100
Iterations

(Figure 5)

- in Figure 1, it must be noted that the lack of normality here is due to the lack of available bins. The size of this data set was no more than 4950, and spanned across merely 9 bins. As the number of bins needed increases, the more normal the graph becomes.
- in Figure 5, the inconsistencies in the normal distribution can be attributed to a too small a sample size. If given the computing power and time, one could compute all pairs less than 10^{1000}. Note the increasing mean iterations as we climb the upper bound.

Introduction to Complexity Theory

Insofar as this course focuses on algorithms, it is first necessary to define the term 'algorithm', as well as several other auxiliary terms used across this presentation. It is necessary to define some sort of machine-independent metric for algorithm running time, so it can be universally used. It is also necessary to define some auxiliary terms to do this:

- An algorithm is a well-defined computational procedure that takes a variable input and halts with an output. (Oorschot and Vanstone, A Handbook of Applied Cryptography)
- The running-time of an algorithm is a function of the number of single-bit operations it takes for an algorithm to complete.
- The size of an input is the number of bits in the input, or the number of inputs, to an algorithm.
- The execution time of an algorithm's implementation is taken here to mean the average amount of time in nanoseconds for a particular implementation of an algorithm to run.

A Crash Course in Algorithmic Analysis

Often, the run-time functions can grow to be complex and involve multiple factors. It is often enough to know that a particular algorithm "grows proportionally to" some value (Goodrich, Data Structures and Algorithms in C++). This is known as asymptotic notation.

- Let f(n) and g(n) be functions such that $f(n) : \mathbb{Z} - > \mathbb{R}$ and $g(n) : \mathbb{Z} - > \mathbb{R}$. Then we say that f(n) is O(g(n)) iff $\exists$ c >0 and n $\in \mathbb{Z}$ such that f(n) $\leq$ cg(n), for $\forall$ n' $\geq$ n
- This is known as Big-O notation. It allows us to ignore constant factors and lower-order terms, focusing only on the highest-order term–the one which governs its asymptotic behavior.
- In general, there are six functions fundamental to algorithmic analysis: log(n) <n <n log(n) $<n^2$ $<n^3$ $<2^n$ (Goodrich, Data Structures and Algorithms in C++)
- Hence, this is known as asymptotic notation.

Bit-Complexity of Integer Operations

In the ring of integers, we define the bit complexity of addition and multiplication, as well as their inverses, as below:

Operation	Bit-Complexity
+	O(log(a) + log(b)) = O(log(n))
-	O(log(a) + log(b)) = O(log(n))
*	O(log(a) * log(b)) = $O(log(n)^2)$
\	O(log(a) * log(b)) = $O(log(n)^2)$

These are fundamental values; they allow a baseline computation for running-time complexity. Things like recursion, which incur an extra execution time penalty for function calls, are one of many factors that can confound execution time.

Euclidean Algorithm Variations

Three iterative versions, two recursive versions, and two binary versions of this algorithm were implemented.

- Binary GCD algorithm will run with $O(log(b)^2)$ bit complexity, where b represents the maximum number of bits in the integer inputs.
- The iterative modular algorithm will run with O(log(u) * log(v)), where u and v are integer inputs. The execution time may be longer, due to an added penalty for function calls.
- The least-remainders Euclidean algorithm will run in, at the worst case, O(log(u) * log(v)). (Bach and Shallit, Algorithmic Number Theory).

Iterative, Subtractive

```
Input: integers U, V
Output: GCD(U,V)
if  U >V then
   swap(U,V)
end if
while U ≠ V do
   if U >V then
      U ¡- U - V
   else
      V = V - U
   end if
end while
return U
```

Iterative, Modular

Input: integers U, V
Output: GCD(U,V)
if V >U **then**
  swap(U,V)
end if
while U $\neq$ V **do**
  (U,V) <- (V, U MOD V)
end while
return U

(Bach and Shallit, Algorithmic Number Theory)

Recursive, Least Remainder

```
Input: integers U, V
Output: GCD(U,V)
if V == 0 then
   return V
else
   return LeastRemainderEuclidean(V, U MOD V)
end if
```

Auxiliary Functions for Least Remainder

```
Input: integers U,V
Output: "Round Mod" function
if V == 0 then
   return U
else
   Return U = (V * ROUND(U / V))
end if
```

```
Input: integers U to round
Output: Integer round function
Frac = U - 0.5
Return CEIL(Frac) as Integer
```

Binary GCD

```
g <- 1
while U, V are even do
   U <- U/2
   V <- V/2
   g = 2g
end while
while U ≠ 0 do
   if U is even then
      U <- U/2
   else if V is even then
      V <- V/2
   else
      t <- abs(U-V)/2
      if U <V then
         V <- t
      end if
   end if
end while
```

Binary GCD cont'd

```
else
U <- T
MAX(U,V) = abs(U,V)/2
return g*V
```

Implementation Detail: Java BigInteger Library

- In order to test these algorithms against large numbers, the Java BigInteger Library was used due to its support for arbitrary-precision arithmetic and numbers.
  - A BigInteger is an abstraction for an arbitrary-length bit string
  - In-built GCD function against which our algorithms were compared
  - No in-built random number generator; used Java's random number generator to generate individual bits in a string and wrap these into a BigInteger
- Hierarchical implementation: parent Euclidean Algorithm class with Iterative Algorithm and Recursive Algorithm derived classes; each implementation was a subclass of these
- In all cases, the corresponding BigInteger arithmetic operation was used; for the binary GCD algorithm, both the arithmetic operations and functions which performed direct bit shifting, were used.
- All data were collected on a Linux-based server hosted through the Texas A&M University Department of Computer Science. This process was automated with scripts in Bash.

Baseline Run-Times for BigInteger Library Functions

In order to effectively measure the running-time of each algorithm, some data on the execution-time of the relevant BigInteger functions were collected, as well as the execution time of our random number generation function:

Function	Average Time (ns)
Random Number Generation	1.9e5 ns
Addition	1.3e5 ns
Subtraction	1.0e5 ns
Multiplication	1.4e5 ns
Modulus	1.4e5 ns
Division	1.7e5 ns
Left Shift	3.9e2 ns
Right Shift	4.7e2 ns
Absolute Value	2.3e2 ns
Assignment	2.1e2 ns

All data measured was adjusted to account for the costs of absolute value, random number generation, et cetera.

Complexity Analysis Results

The resultant execution times for each algorithm are given below:

Algorithm	**Execution Time**
Extended	6.09e9 ns
Iterative, Mod	4.35e9 ns
Iterative, Subtractive	1.91e9 ns
Recursive, Mod	4.77e9 ns
Recursive, Least Positive Remaidner	11.8e9 ns
Steins Binary Algorithm, BigInteger Operations	6.04e9 ns
Steins Binary Algorithm, Bit Shifting Operations	4.68e9 ns
Java BigInteger GCD	1.30e9 ns

Neural Networks

Artificial Intelligence (AI) is a growing area in computer science with a myriad of applications across disciplines. AI is divided into subgroups of study with seven fundamental areas:

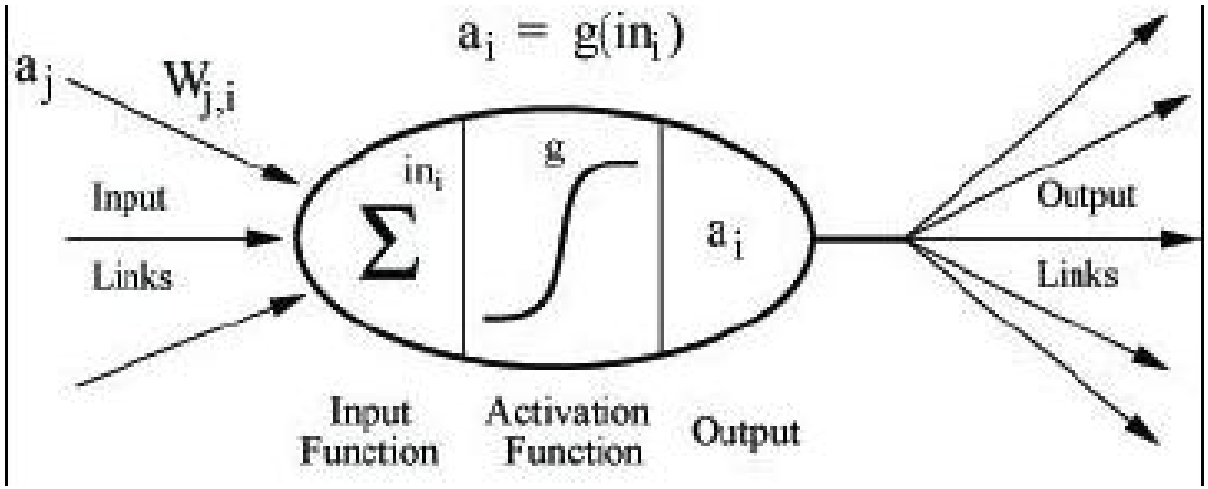
1. reasoning
2. knowledge
3. planning
4. learning
5. communication/natural language processing
6. perception
7. moving and manipulating

As well as an eighth subgroup:

8. the notion of distributed intelligence.

- Machine learning: refers to the ability of a computer to study data, form patterns, and make predictions
- Typical applications of AI involve pattern-recognition problems; however, applying AI and machine learning methods specifically to number theory problems has not been studied thoroughly
- Our aim was to evaluate the effectiveness of an Artificial Neural Network when applied to three key problems:
  1. Classification of prime and composite numbers
  2. Integer factorization
  3. Determining the greatest common divisor (GCD)

- At its core, an artificial neural network is an abstraction for the functionality of neurons and human cognition
- The process of learning is achieved through alterations of synaptic connections between neurons, making an ANN the ideal choice for pattern-recognition problems


$a_i = g(in_i)$
a_j
$W_{j,i}$
Input
Links
in_i
Σ
g
a_i
Output
Links
Input Function
Activation Function
Output

(Figure 6)

- Input is provided to an input function
- Each input has an associated weight
- Each neuron will then compute a weighted sum of its inputs
- Then it applies an activation function to act as a threshold to derive output

There are several options for Thresholding Functions:

- AND
- OR
- XOR
- Linear
- Sigmoid
- Hyperbolic Tangent

The complexity of the problem being attempted by the ANN determines the correct function to use.

- Neurons whose activation functions create hard thresholds are called perceptrons.
- Neurons whose activation functions are logistic functions are known as sigmoid perceptrons.
- The perceptron maps input directly to output.
- Historically, single perceptrons were used to solve problems, but were not nearly as successful.

There are two principal ANN architectures:

1. Feed-forward networks
   - Directed, acyclic graph
   - Flows in one direction only
   - It maintains no internal state nor represents more than the input passed to the network
   - May be split into multiple layers: Input, Hidden, Output
   - Lacks any kind of memory
2. Recurrent Networks
   - Utilizes the notion of recurrence
   - Cyclic
   - Maintains a small amount of short-term memory

There are two main learning methods in an ANN:

1. Supervised
   - Done through an external teacher
   - The network is told the desired response to certain input signals
   - Reinforcement learning is a form of supervised where a correct response is given a certain weight and the better the weight, the more correct the answer
2. Unsupervised
   - Also known as self-organizing
   - Based entirely upon local information
   - The network is given a large amount of data and allowed to recognize its own patterns

Implementation of any learning algorithm, supervised or unsupervised, is complicated by the presence of multiple outputs and hidden layers of the network: the fact that the hidden layer leaves its computations out of reach of the user means that any information obtained about errors at the hidden layer is all but useless to the user. Therefore, an algorithm for backpropagation must be used to back-propagate errors from the output layer to hidden layers.

function BACK-PROP-LEARNING(*examples, network*) returns a neural network
inputs: *examples,* a set of examples, each with input vector **x** and output vector y
network, a multilayer network with *L* layers, weights , activation function *g*
local variables: A, a vector of errors, indexed by network node

repeat
for each weight $w_{i,j}$ in *network* **do**
a small random number
for each example (x, y) in *examples* do
/ * *Propagate the inputs forward to compute the outputs* * /
for each node *i* in the input layer do
$a_i \quad x_i$
for $\ell = 2$ *to* L **do**
for each node j in layer **do**
in_j
a) $g(in_j)$
/ * *Propagate deltas backward from output layer to input layer* * /
for each node j in the output layer **do**
$\Delta[j] \quad g'(in_j) \times (y_j —$
fur f = L — 1 to 1 du
for each node *i* in layer **do**
$\Delta[i] \quad g'(in_i) \sum_j \quad j$
1 Update every weight in network using deltas *1*
for each weight $w_{i,j}$ in network do
es x a_t x $\Delta[j]$
until some stopping criterion is satisfied
return n*etwork*

Figure 18.24 The back-propagation algorithm for learning in multilayer networks

The ANN was implemented in Python, using the Pybrain machine learning library. The network was set up as a reinforced back-propagation system. For factorization, a class was defined containing prime factorizations of numbers 1 through 15. For primality testing, a list of prime numbers up to 10,000 was obtained. For the GCD problem, numbers were randomly generated from 0 to 100, and every possible combination of pairs tested.

The teaching dataset used was combinations of (x,y), with x ranging from 2-5 and y ranging from 1-20. The input was a pair of x and y combined with the desired output of the gcd. It was set up to randomly go through $10,000$ iterations of the training dataset and then attempt to guess the output for each of the teaching entries. For the majority of the runs, there were 2 hidden layers activated. When the learning rate was increased from 0.05 to 0.1, the results improved slightly, though not drastically. Momentum was determined to work best at the value 0.025. Surprisingly, the organization of the training data had some of the most interesting results.

Sample data: (2,1) [1.] (2,2) [2.] (2,3) [1.58476845]
(2,4) [1.58827758] (2,5) [1.58827945] (2,6) [1.58827945]
(2,7) [1.58827945] (2,8) [1.58827945] (2,9) [1.58827945]
(2,10) [1.58827945] (3,1) [0.99940695] (3,2) [1.00206377]
(3,3) [3.38776544] (3,4) [1.58842719] (3,5) [1.58827903]
(3,6) [1.58827945] (3,7) [1.58827945] (3,8) [1.58827945]
(3,9) [1.58827945] (3,10) [1.58827945] (4,1) [0.99940584]
(4,2) [0.99941081] (4,3) [1.01127187] (4,4) [4.41099692]
(4,5) [1.58941163] (4,6) [1.58827936] (4,7) [1.58827945]
(4,8) [1.58827945] (4,9) [1.58827945] (4,10) [1.58827945]
(5,1) [0.99940584] (5,2) [0.99940585] (5,3) [0.99942807]
(5,4) [1.05195013] (5,5) [4.77198339]

There are several explanations that could be applied to why the network was unable to produce the desired results.

1. The training dataset could have been too small.
2. Prime numbers do not follow a well-defined pattern.
3. Only feed-forward networks were used.
4. Not all potential functions for activation were tested.
5. While various parameters of the network were tested, very little was tested in the way of data representation beyond an elementary normalization of GCD inputs and outputs.
6. Finally, it could be that a neural network in the implementation used for our project is simply not well suited for this type of problem.

There have already been well established algorithms, as demonstrated in the previous project, which rely on more concrete building blocks than just recognizing patterns that may not even be there. It is possible that other methods of artificial intelligence, such as support vector machines, might have more success at one or all of the problems investigated.

References

P. W. Epasinghe (1983)

Euclid's Algorithm and the Fibonacci Numbers

The Fibonacci Quarterly

Bach, Eric and Jeffrey Shallit (1996)

Algorithmic Number Theory

Foundations of Computing Series

Menezes, A.J., van Oorschot, P. and Scott Vanstone (1997)

Handbook of Applied Cryptography

CRC Press

The End