Proof for Conjecture

Benjamin Hansen

April 1, 2020

Abstract

Today's encryption algorithms (the process of securing data) exploit the inability of modern computers to factorize large integers. If a classical algorithm can generate prime factors very quickly, the security of our systems would be undermined. As such, we present here thoughts on factoring and a theorem to prove relationships between factors.

Introduction

While delivering fast food for a restaurant, I was in my car a lot just thinking while I drove. Throughout 2019 I became more and more intrigued with factoring numbers. I had developed a computer model which factored numbers as fast as I knew how and over and over thought of ways to speed up that algorithm. It did well, but when the numbers were in the billions to trillions, it took an insane amount of time to factor. Usually with numbers that large, I never saw the algorithm complete.

This work is built off the wonderful ideas of discrete mathematics I learned in my Discrete Mathematics I & II classes at BYU-Idaho. Much thanks to Professor Lee Barney and Professor Rick Neff for teaching me mathematics in such a way that has left my mind pondering and hungering for more.

On my food deliveries, I thought of different operations that could be performed to compute the factors of a number.

Given a natural number n

The following algorithms list my attempts and the number of critical operations performed to compute the factors of a number. By critical I mean

operations like division, modulus, multiplication, and arithmetic with values greater than 1. Adding or subtracting one to a value is a very quick process and will not be reported. At the beginning, each attempt explains a brief overview of the algorithm.

NOTE these quantity amounts for how many operations performed are for factoring a number instead of finding the prime factorization of a number. Also, the operations to save the factors of the number when found are not included, yet may be discussed in the summary. The nitty-gritty operations of computing those factors is the main performance bottleneck.

To get the next factor

First attempt

Summary: Start with potential factor f at n. If $f \bmod n = 0$, meaning f divides n, (written as $f|n$), then f is a factor of n. Decrement f by one each time until $f = 1$ and that is the last factor of n.

Critical Operations:

- n 1 modulo operations
- n 1 condition checks
- n iterations

Second attempt

Summary: Start with potential factor f at $\lfloor \frac{n}{2} \rfloor$. If $f \bmod n = 0$, meaning f divides n, (written as $f|n$), then f is a factor of n. To get it's counterpart factor, the value where $f \cdot x$ for some integer x equals n, integer divide n by f. Since f is a factor in this iteration, $\frac{n}{f}$ will produce the same value as the integer division. Decrement f by one each time until $\frac{n}{f} f$, which means we have found all the factor pairs.

Critical Operations:

- 1 integer divide
- $\lfloor \frac{n}{2} \rfloor$ 1 modulo operations
- $\lfloor \frac{n}{2} \rfloor$ floating-point divisions
- $2\lfloor \frac{n}{2} \rfloor$ condition checks
- $\lfloor \frac{n}{2} \rfloor$ iterations

Third attempt

Summary: Start with potential factor f at $\lfloor\sqrt{n}\rfloor$. If $f \bmod n = 0$, meaning f divides n, (written as $f|n$), then f is a factor of n. To get it's counterpart factor, the value where $f \cdot x$ for some integer x equals n, integer divide n by f. Since f is a factor in this iteration, $\frac{n}{f}$ will produce the same value as the integer division. Decrement f by one each time until $f = 1$, which is the last factor.

Critical Operations:

- 1 $\sqrt{n}$ computation
- 1 truncation of a floating-point value
- $\lfloor\sqrt{n}\rfloor$ 1 modulo operations
- $\lfloor\sqrt{n}\rfloor$ 1 condition checks
- $\lfloor\sqrt{n}\rfloor$ iterations

Fourth attempt

Summary: Start with potential factor f at $\lfloor\sqrt{n}\rfloor$. If $\frac{n}{f}$ when truncated (meaning all decimals are chopped off making the decimal a whole number) has the same value as $\frac{n}{f}$, then $f|n$ and is a factor. The division is the counterpart factor. Decrement f by one each time until $f = 1$, which is the last factor.

Critical Operations:

- 1 $\sqrt{n}$ computation
- $\lfloor\sqrt{n}\rfloor$ 1 floating-point division
- $\lfloor\sqrt{n}\rfloor$ truncations of a floating-point value
- $\lfloor\sqrt{n}\rfloor$ 1 condition checks
- $\lfloor\sqrt{n}\rfloor$ iterations

Final attempt

Summary: This is the algorithm I have been and will be talking the most about. It goes like this. Everything will be retained as fractions using fractional arithmetic instead of floating-point arithmetic. Later I will give an explanation how I held the fractions. Start y_k at $\frac{n}{1}$. k will go from 1 to $\lfloor\sqrt{n}\rfloor$ and this algorithm will relate y_k to k, as counterpart factor pairs of n. To check if y_k is a factor with counterpart factor k, check if the numerator in

y_k is divisible by its denominator. If so, evaluate the fraction using integer division to get the integer factor of y_k. That means k is also a factor. The next potential factor, $y_{k+1} = y_k \ \frac{y_k}{k+1}$, which you will see proven later. Each $y_k = y_{k-1} \ \frac{y_{k-1}}{k}$. Continue checking if the next y_k computed is a factor. Go until y_k has a numerator that is less than the denominator. That will denote that $k\lfloor\sqrt{n}\rfloor$ and we are done factoring.

Critical Operations:

- $\lfloor\sqrt{n}\rfloor \times 5 + 1$ multiplications
- $2\lfloor\sqrt{n}\rfloor$ condition checks
- $\lfloor\sqrt{n}\rfloor$ 1 modulo operations
- $\lfloor\sqrt{n}\rfloor$ 1 integer division operations
- $\lfloor\sqrt{n}\rfloor$ iterations

We will summarize the results of this final attempt later and explain its limitations.

Theorem and Proof

Author: Benjamin Hansen

Theorem

Let $n \in \ltimes$ For $k = 1, \ldots, \lfloor\sqrt{n}\rfloor$, let a_k be the real number such that $a_k k = n$. The numbers $a_1, a_2, \ldots, a_{\lfloor\sqrt{n}\rfloor}$ satisfy the recurrence relation

$a_k = a_{k-1}\frac{a_{k-1}}{k}$

Proof

By definition, $a_k k = n$ and $a_{k-1}(k-1) = n$ and thus $a_k k = a_{k-1}(k-1)$. Dividing both sides by k gives $a_k = \frac{k-1}{k}a_{k-1} = (1 \ \frac{1}{k})a_{k-1} = a_{k-1} \ \frac{a_{k-1}}{k}$.□

Acknowledgements:

Thanks to Professor Curtis Nelson for his refinements and feedback on the statement and proof.

Motivation

As you can see, the final attempt algorithm takes considerable more operations. It is by far the slowest of them all. I will admit that. But, what we learn from this algorithm can enhance our understanding and uncover insights into the relationships of numbers that we have not before seen or considered.

Factoring is all about finding relationships between numbers. This algorithm I will propose, the slowest attempt, shows us the following relationships between numbers.

Given some natural number n, lets say we wanted to find the real number y needed such that $x \cdot y = n$ for counting values of x from 1 to the $\lfloor \sqrt{n} \rfloor$. Lets consider an example:

Example

Let $n = 36$. I want to find the value needed such that

$1 \cdot y_1 = n$ and

$2 \cdot y_2 = n$ and

$3 \cdot y_3 = n$ and

$4 \cdot y_4 = n$ and

$5 \cdot y_5 = n$ and

$6 \cdot y_6 = n$.

Well, I can just compute $\frac{n}{1}, \frac{n}{2}, ..., \frac{n}{6}$ to find the values of y_k where $k = 1, 2, ..., \lfloor \sqrt{n} \rfloor$.

But what if there was some interesting relationship between those values of y_k - that I could find the next value y_{k+1} without needing to even use n in my equation?

There is a way! Take the previous y_{k-1} and subtract the division of y_{k-1} by k, k being the integer counting value below the square root of n.

$y_k = y_{k-1} \ \ \frac{y_{k-1}}{k}$.

Where do we start for our value of y_k? $y_1 = \frac{n}{1} = n$ which should hopefully seem obvious.

That is all we need to get this recurrence relation going. A recurrence relation is a recursive relation where the next value can be computed based on the previous values. Unlike functions like $\sin x$ where they don't require knowledge of previous values computed, recurrences relations have the form

$a_k = c \cdot a_{k-1} + b$ or something with the previously value generating the next value.

Our recurrence relation is

$y_k = y_{k-1} \ \frac{y_{k-1}}{k}$.

Let's see it work for our example.

$y_2 = y_1 \ \frac{y_1}{2} = 36 \ \frac{36}{2} = 18.$

$y_3 = y_2 \ \frac{y_2}{3} = 18 \ \frac{18}{3} = 18 \ \ 6 = 12$, and sure enough $3 \cdot 12 = 36$

$y_4 = y_3 \ \frac{y_3}{4} = 12 \ \frac{12}{4} = 12 \ \ 3 = 9$, and sure enough $4 \cdot 9 = 36$

$y_5 = y_4 \ \frac{y_4}{5} = 9 \ \ \frac{9}{5} = 7\frac{1}{5}$, and sure enough $5 \cdot 7\frac{1}{5} = 36$

$y_6 = y_5 \ \frac{y_5}{6} = 7\frac{1}{5} \ \frac{7\frac{1}{5}}{6} = 7\frac{1}{5} \ \frac{36}{30} = 6$, and sure enough $6 \cdot 6 = 36$

Isn't that fascinating? It seemed like magic to me at first. But the more I thought about it and tried to prove it in my mind and in writing, the more basic the problem became. It became more and more obvious that this recurrence works. But at first, this algorithm seemed like a gold mine that hadn't been discovered before!

Relationships Uncovered - Usefulness

From this algorithm and theorem we uncover some interesting relationships:

- The next integer in the counting sequence up to $\lfloor\sqrt{(}n)\rfloor$ has a scalar value y_k where $y_k = y_{k-1} \ \frac{y_{k-1}}{k}$ where y_k and k were defined early.

  Remark: This is interesting because usually we think of possible factors or multiplication values ($k \cdot y_k$ needed to equal a certain value n as usually found by just dividing n by k. Now we have another relationship in our tool belt of mathematical facts that the next value has a scalar multiple of $y_{k-1} \ \frac{y_{k-1}}{k}$ with no need to know the value of n.

- If y_k is an integer (whole number), then we are guaranteed that y_k is a factor of n.

  Remark: This is what really intrigued me at first. Who would have thought that through this recurrence algorithm, you can find all the factors of a number. It was extremely weird and fascinating to me. The algorithm goes about computing numbers in the middle of the algorithm that don't even seem related to n. And indeed, if someone just saw values of y_k without knowing where they came from, they would have no clue like me how they relate to a common number. Yet, majestically from fractions arise whole number factors of n. Though we may hit a long run of non-factors of a number through this algorithm (like say $n = 124897238$), this algorithm will produce every whole number factor of 124897238 only using 124897238 for one iteration, the first iteration.

- A prime number p will have no whole number values for y_k except y_1.

  Remark: I am greatly interested in prime numbers and uncovering relationships about them that may unlock doors to finding quicker methods of doing computation with prime numbers or factorizing them. The RSA Cryptosystem is what keeps our technology world secure today. It exploits the inability of a modern computer to factor very large integers (thousands of digits long). Thus, if an algorithm can factor numbers by computation instead of iteration (looping through a bunch of values to find possible answers), then our security systems will vanish in safety. Many bright minds have put their heads into trying to find such quick algorithms for finding prime factorizations but no one has been able to find some quick formula that will generate the prime factor.

  Aside: Prime factorization is a number theory algorithm which finds the prime factors of a number. For example, 15 can be written as the product of two prime factors: 5 and 3. 36 can be written as $2 \cdot 2 \cdot 3 \cdot 3 = 2^2 \cdot 3^2$.

- The last y_k before $k\lfloor\sqrt{n}\rfloor$ will be a very rough yet close estimate of the $\lfloor\sqrt{n}\rfloor$.

  Remark: As you may have noticed, two of the other factoring attempts use the $\sqrt{n}$. If you look into the code though, the algorithms compute the square root, creating some decimal value, only to truncate the decimal digits and create a whole number. This algorithm doesn't use the slow operations of square root and division to compute the next y_k. It only uses multiplication and subtraction, which I think is an interesting combination. Remember, division is just repeated subtraction as multiplication is just repeated addition.

  The algorithm I found computes the integer square root when the final fraction y_k is evaluated using integer division. I tried this out for values 1 - 100 and the algorithm correctly computed the integer square root of that value. What's interesting is that it does so without using the square root or checking if the integer square root worked. This makes sense because the stopping condition, explained below, happens naturally when our k counter goes above the $\sqrt{n}$. Thus the last k is the integer square root of n and y_k is also the integer square root of n.

  If this algorithm can be enhanced, it will be faster than the floating-point square root algorithm. All factors of a number, both prime or

not, can be found by iterating over values of y_k without needing to compute $\lfloor\sqrt{n}\rfloor$.

- Easy stopping condition. To stop, all we need to check is if the fraction y_k is less than 1 (meaning the denominator is greater than the numerator.

  Remark: The computer model I used mainted absolute precision by representing the y_k values as fractions instead of a floating-point value. This means that instead of round off errors (because computers only hold a finite number of decimals) the arithmetic operations are exact because any fractions are not evaluated (turned into a decimal value). The arithmetic of $y_k = y_{k-1} - \frac{y_{k-1}}{k}$ is held precise, as fractions, until the last y_k when then and only then we evaluate the fraction. That final evaluation is a very rough estimate of the square root of n and when truncated, as explained previously using integer division, produces the integer square root of the number.

Computer Model

The following code are the methods and the attempts created to speed up factoring numbers. The programming language is Erlang, a popular highly-scalable, massively parallel programming system.

It calls functions in a fractions module (library) that is imported at the top. I created this and will include the functions after this code block.

```
-export([sum_of_factors/1,f/1,with_subtraction/1,fact_below_n/1,highest_factor/1,factor
-import(fractions, [sub/2,divide/2, eval/1, den_le_num/1, is_whole_then_divide/1]).

%%%%% Slow Algorithm. First Attempt
%% Just a list, let's put them in tuples above
factors_of(N) -> factors_of(N, N, []).

%% param: 1st is what is being divided, 2nd is what are we dividing by, 3rd is the list
factors_of(_, 1, Factors) ->
    [1|Factors];
factors_of(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
factors_of(Dividend, Divisor - 1, [Divisor|Factors]);
factors_of(Dividend, Divisor, Factors) ->
factors_of(Dividend, Divisor - 1, Factors).
```

```
%% Second Attempt
%% Call tail recursive fact function giving it N (to be factored),
%% N div 2 "or floor division 2", and the empty set
%% Why floor division two? Because every number N will have at least a
%% factor of N div 2 because 2 * the biggest number possible that divides N
%% will be the biggest factor possible
fact1(N) when N > 0 -> fact1(N, N div 2, [{1,N}|[]]).

%% when the Dividend divided by the Divisor has become greater
%% than the Divisor, we know we're done factoring, no more tuples to find
fact1(Dividend, Divisor, Factors) when Dividend / Divisor > Divisor ->
Factors;
%% when the Divisor evenly divides the dividend
fact1(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
fact1(Dividend, Divisor - 1,
     [{Dividend div Divisor, Divisor}|Factors]);
%% else keep going
fact1(Dividend, Divisor, Factors) ->
fact1(Dividend, Divisor - 1, Factors).

%% Third Attempt
%% Call tail recursive fact function giving it N (to be factored),
%% N div 2 "or floor division 2", and the empty set
%% Why floor division two? Because every number N will have at least a
%% factor of N div 2 because 2 * the biggest number possible that divides N
%% will be the biggest factor possible
%%
%% Actually!! An even quicker algorithm... any number if we consider each factor
%% with its pair that evenly multiplies to N, then any factor can be found by
%% testing divisors less than or equal to the square root of N
%% Truncate the square root to get rid of any floating point decimals.
fact(N) -> fact(N, trunc(math:sqrt(N)), [{1,N}|[]]).

%% When The Divisor has reached one, we're done. We already put 1 as a Factor
fact(_,1,Factors) -> lists:sort(Factors); %% Call sort which with tuples will sort by t

%% when the Divisor evenly divides the dividend
fact(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
fact(Dividend, Divisor - 1,
     [{Divisor, Dividend div Divisor}|Factors]);
```

```
%% else keep going
fact(Dividend, Divisor, Factors) ->
fact(Dividend, Divisor - 1, Factors).

%% Fourth Attempt
%% Use / and truncate instead of div and rem to generate factors.
%% Our hope is this is faster
%% All factors of a number can be found starting at its square root down to 1
f(N) -> f(N,trunc(math:sqrt(N)),[]).

f(N,1,List) -> lists:sort([{1,N}|List]);

f(N,PotentialFactor,List) ->
%% Get the division
Division = N / PotentialFactor,
Trunc = trunc(Division),
%% If the divisor is a potential factor of N then like 36 / 4 = 9.0 then
%% trunc(9.0) to 9 == 9.0
case Trunc == Division of
%% It is a factor
true -> f(N,PotentialFactor - 1,[{PotentialFactor,Trunc}|List]);
_    -> f(N,PotentialFactor - 1,List)
end.

%% Final attempt
%% A better way to do with_subtraction() that uses the fraction library we are creating
%% A recursive algorithm which recurses integer factors up to the square root of a numb
%% x_i*y_i = n
%% y_i = y_i-1 - y_i-1/x_i
%% When do we stop? When we encounter an invalid solution, namely x_i > n
%% But we compute that (because we're defining a alternate way of computing the square
%% as the numerator of y_i-1 / x_i is less than the denominator
with_sub(N) ->
    io:format("Beginning ~w ...~n", [N]),
    with_sub(2, {N, 1}). % x_0 = 2, y_0 = N

%% This is the previous Y not Y - 1.
%% Calculate Y and check if it is a valid solution
with_sub(X_i, Y_i_minus_1) ->
    io:format("X_i: ~w, Y_i-1: ~w~n", [X_i, Y_i_minus_1]),
```

```
    io:format("Evaluated Y_i-1: ~w~n", [eval(Y_i_minus_1)]),
    %io:fread("Keep Going ~w~n", [X_i]),
    %% Compute the initial step to see if it is a valid solution
    Y_divided = divide(Y_i_minus_1, X_i), % y_i-1 / x_i
    %% While the the denominator is less than or equal to the numerator,
    %% we still have plausible integer fractions below the square root of n
    %% So keep going
    %% Check the proposed division
    case den_le_num(Y_divided) of
        %% Continue recursing
        %% x_i+1 = x_i + 1
        %% y_i = y_i-1/x_i
        true -> with_sub(X_i + 1, sub(Y_i_minus_1, Y_divided)); % y_i-1 - y_i-1/x_i
        %% Else return the estimated square root!
        _    -> eval(Y_i_minus_1)
    end.

%% A better way to do with_subtraction() that uses the fraction library we are creating
%% A recursive algorithm which recurses integer factors up to the square root of a numb
%% x_i*y_i = n
%% y_i = y_i-1 - y_i-1/x_i
%% When do we stop? When we encounter an invalid solution, namely x_i > n
%% But we compute that (because we're defining a alternate way of computing the square
%% as the numerator of y_i-1 / x_i is less than the denominator
sub_factoring(N) ->
    io:format("Beginning ~w ...~n", [N]),
    %% Pass in the empty list to build our list of factors off of
    %% First obvious factor is 1 and itself
    %% Put them in pairs
    sub_factoring(2, {N, 1}, [{1,N}]). % x_0 = 2, y_0 = N

%% This is the previous Y not Y - 1.
%% Calculate Y and check if it is a valid solution
%% Receive a list of factors to add too
sub_factoring(X_i, Y_i_minus_1, Factors) ->
    io:format("X_i: ~w, Y_i-1: ~w~n", [X_i, Y_i_minus_1]),
    io:format("Evaluated Y_i-1: ~w~n", [eval(Y_i_minus_1)]),
    %io:fread("Keep Going ~w~n", [X_i]),
    %% Compute the initial step to see if it is a valid solution
    Y_divided = divide(Y_i_minus_1, X_i), % y_i-1 / x_i
```

```
    %% While the the denominator is less than or equal to the numerator,
    %% we still have plausible integer fractions below the square root of n
    %% So keep going
    %% Check the proposed division
    case den_le_num(Y_divided) of
        %% Continue recursing
        %% x_i+1 = x_i + 1
        true -> % if the case fails, Y_i will not be a factor of N
            %% y_i = y_i-1 - y_i-1/x_i = y_i-1 - y_divided
            Y_i = sub(Y_i_minus_1, Y_divided),
            case is_whole_then_divide(Y_i) of
                %% If zero is returned, the whole number condition check failed
                %% So don't add the factor
                0 -> sub_factoring(X_i + 1, Y_i, Factors); % y_i-1 - y_i-1/x_i
                %% Else a whole number is returned, the counterpart factor to x_i where
                %% Add the tuple pair to the factor list
                Y_i_whole -> sub_factoring(X_i + 1, Y_i, [{X_i, Y_i_whole}|Factors]) %
            end;
        %% Else return the Factors of N
        _    -> Factors
    end.
  -module(factors).
-export([sum_of_factors/1,f/1,with_subtraction/1,fact_below_n/1,highest_factor/1,factor
-import(fractions, [sub/2,divide/2, eval/1, den_le_num/1, is_whole_then_divide/1]).

%%%%% Slow Algorithm. First Attempt
%% Just a list, let's put them in tuples above
factors_of(N) -> factors_of(N, N, []).

%% param: 1st is what is being divided, 2nd is what are we dividing by, 3rd is the list
factors_of(_, 1, Factors) ->
[1|Factors];
factors_of(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
factors_of(Dividend, Divisor - 1, [Divisor|Factors]);
factors_of(Dividend, Divisor, Factors) ->
factors_of(Dividend, Divisor - 1, Factors).

%% Second Attempt
%% Call tail recursive fact function giving it N (to be factored),
%% N div 2 "or floor division 2", and the empty set
```

```
%% Why floor division two? Because every number N will have at least a
%% factor of N div 2 because 2 * the biggest number possible that divides N
%% will be the biggest factor possible
fact1(N) when N > 0 -> fact1(N, N div 2, [{1,N}|[]]).

%% when the Dividend divided by the Divisor has become greater
%% than the Divisor, we know we're done factoring, no more tuples to find
fact1(Dividend, Divisor, Factors) when Dividend / Divisor > Divisor ->
Factors;
%% when the Divisor evenly divides the dividend
fact1(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
fact1(Dividend, Divisor - 1,
     [{Dividend div Divisor, Divisor}|Factors]);
%% else keep going
fact1(Dividend, Divisor, Factors) ->
fact1(Dividend, Divisor - 1, Factors).

%% Third Attempt
%% Call tail recursive fact function giving it N (to be factored),
%% N div 2 "or floor division 2", and the empty set
%% Why floor division two? Because every number N will have at least a
%% factor of N div 2 because 2 * the biggest number possible that divides N
%% will be the biggest factor possible
%%
%% Actually!! An even quicker algorithm... any number if we consider each factor
%% with its pair that evenly multiplies to N, then any factor can be found by
%% testing divisors less than or equal to the square root of N
%% Truncate the square root to get rid of any floating point decimals.
fact(N) -> fact(N, trunc(math:sqrt(N)), [{1,N}|[]]).

%% When The Divisor has reached one, we're done. We already put 1 as a Factor
fact(_,1,Factors) -> lists:sort(Factors); %% Call sort which with tuples will sort by t

%% when the Divisor evenly divides the dividend
fact(Dividend, Divisor, Factors) when Dividend rem Divisor == 0 ->
fact(Dividend, Divisor - 1,
     [{Divisor, Dividend div Divisor}|Factors]);
%% else keep going
fact(Dividend, Divisor, Factors) ->
fact(Dividend, Divisor - 1, Factors).
```

```
%% Fourth Attempt
%% Use / and truncate instead of div and rem to generate factors.
%% Our hope is this is faster
%% All factors of a number can be found starting at its square root down to 1
f(N) -> f(N,trunc(math:sqrt(N)),[]).

f(N,1,List) -> lists:sort([{1,N}|List]);

f(N,PotentialFactor,List) ->
%% Get the division
Division = N / PotentialFactor,
Trunc = trunc(Division),
%% If the divisor is a potential factor of N then like 36 / 4 = 9.0 then
%% trunc(9.0) to 9 == 9.0
case Trunc == Division of
%% It is a factor
true -> f(N,PotentialFactor - 1,[{PotentialFactor,Trunc}|List]);
_    -> f(N,PotentialFactor - 1,List)
end.

%% Final attempt
%% A better way to do with_subtraction() that uses the fraction library we are creating
%% A recursive algorithm which recurses integer factors up to the square root of a numb
%% x_i*y_i = n
%% y_i = y_i-1 - y_i-1/x_i
%% When do we stop? When we encounter an invalid solution, namely x_i > n
%% But we compute that (because we're defining a alternate way of computing the square
%% as the numerator of y_i-1 / x_i is less than the denominator
with_sub(N) ->
    io:format("Beginning ~w ...~n", [N]),
    with_sub(2, {N, 1}). % x_0 = 2, y_0 = N

%% This is the previous Y not Y - 1.
%% Calculate Y and check if it is a valid solution
with_sub(X_i, Y_i_minus_1) ->
    io:format("X_i: ~w, Y_i-1: ~w~n", [X_i, Y_i_minus_1]),
    io:format("Evaluated Y_i-1: ~w~n", [eval(Y_i_minus_1)]),
    %io:fread("Keep Going ~w~n", [X_i]),
    %% Compute the initial step to see if it is a valid solution
```

```
    Y_divided = divide(Y_i_minus_1, X_i), % y_i-1 / x_i
    %% While the the denominator is less than or equal to the numerator,
    %% we still have plausible integer fractions below the square root of n
    %% So keep going
    %% Check the proposed division
    case den_le_num(Y_divided) of
        %% Continue recursing
        %% x_i+1 = x_i + 1
        %% y_i = y_i-1/x_i
        true -> with_sub(X_i + 1, sub(Y_i_minus_1, Y_divided)); % y_i-1 - y_i-1/x_i
        %% Else return the estimated square root!
        _    -> eval(Y_i_minus_1)
    end.

%% A better way to do with_subtraction() that uses the fraction library we are creating
%% A recursive algorithm which recurses integer factors up to the square root of a numb
%% x_i*y_i = n
%% y_i = y_i-1 - y_i-1/x_i
%% When do we stop? When we encounter an invalid solution, namely x_i > n
%% But we compute that (because we're defining a alternate way of computing the square
%% as the numerator of y_i-1 / x_i is less than the denominator
sub_factoring(N) ->
    io:format("Beginning ~w ...~n", [N]),
    %% Pass in the empty list to build our list of factors off of
    %% First obvious factor is 1 and itself
    %% Put them in pairs
    sub_factoring(2, {N, 1}, [{1,N}]). % x_0 = 2, y_0 = N

%% This is the previous Y not Y - 1.
%% Calculate Y and check if it is a valid solution
%% Receive a list of factors to add too
sub_factoring(X_i, Y_i_minus_1, Factors) ->
    io:format("X_i: ~w, Y_i-1: ~w~n", [X_i, Y_i_minus_1]),
    io:format("Evaluated Y_i-1: ~w~n", [eval(Y_i_minus_1)]),
    %io:fread("Keep Going ~w~n", [X_i]),
    %% Compute the initial step to see if it is a valid solution
    Y_divided = divide(Y_i_minus_1, X_i), % y_i-1 / x_i
    %% While the the denominator is less than or equal to the numerator,
    %% we still have plausible integer fractions below the square root of n
    %% So keep going
```

```
    %% Check the proposed division
    case den_le_num(Y_divided) of
        %% Continue recursing
        %% x_i+1 = x_i + 1
        true -> % if the case fails, Y_i will not be a factor of N
            %% y_i = y_i-1 - y_i-1/x_i = y_i-1 - y_divided
            Y_i = sub(Y_i_minus_1, Y_divided),
            case is_whole_then_divide(Y_i) of
                %% If zero is returned, the whole number condition check failed
                %% So don't add the factor
                0 -> sub_factoring(X_i + 1, Y_i, Factors); % y_i-1 - y_i-1/x_i
                %% Else a whole number is returned, the counterpart factor to x_i where
                %% Add the tuple pair to the factor list
                Y_i_whole -> sub_factoring(X_i + 1, Y_i, [{X_i, Y_i_whole}|Factors]) %
            end;
        %% Else return the Factors of N
        _    -> Factors
    end.
```

To maintain ultimate precision with no round off errors, it is necessary sometimes to use a fraction library. Currently the algorithm I use relies solely upon the fraction library to operate completely and exactly. In the future I will experiment and research a way to use floating-point arithmetic and division to speed up the process, but when I first tried, the round off errors were very great and difficult to know when to round and when not too. But with the fraction library, things work perfectly. A major limitation of using fractions for all the computations will be explained in the **Limitations** section.

```
-author("Benjamin Hansen").
-export([new/0,simp/1,add/2,add/3,sub/2,sub/3,divide/2,divide/3, mul/2, mul/3, eval/1,
         integer_divide_eval/1]).
-import(primes, [mod/2]). %% Import mod for modulo arithmetic

%% We will use a tuple of two to hold fractions
new() -> {}.

%% todo
simp(_) -> mod(0, 1).
```

```
%% Return whether the denominator is less than or equal the numerator
den_le_num({Numerator, Denominator}) -> Denominator =< Numerator.

%% Evaluates the fraction into a decimal
eval({Numerator, Denominator}) ->
    Numerator / Denominator.

%% Instead of using floating point division ('/'), use
%% integer division to evaluated this fraction ('div')
integer_divide_eval({Numerator, Denominator})->
    Numerator div Denominator.

%% What makes a fraction actually a whole number?
%% The denominator divides the numerator
is_whole({Numerator, Denominator})->
    %% Return if the Numerator mod the Denominator is zero
    Numerator rem Denominator == 0.

%% This function combines the integer_divide_eval() and is_whole() functions
%% to divide the fraction if it's a whole number or return zero to simulate error - not
is_whole_then_divide(Fraction)->
    case is_whole(Fraction) of
        true -> integer_divide_eval(Fraction);
        _    -> 0
    end.

%% Note: the basic, 2-argument arithmetic functions don't perform any simplification
%% or eager evaluation. They perform the arithmetic as fast (the least amount of operat
%% we know how.
%% Also note, because this is inhouse (for now, I'm just using it) I want it to be quic
%% no divide by zero error checking is made.
%% Also, if fastest performance is desired, the basic 2-argument functions will produce
%% with a negative in the denominator or a negative on both sides of the divide. Call
%% eager eval. methods to simplify the double negative.

%% Take the reciprocal of the fraction
reciprocal({Numerator, Denominator}) -> {Denominator, Numerator}.

%% Add two fractions together
```

```
add({Numerator1, Denominator1}, {Numerator2, Denominator2}) ->
    %% First multiply the other fraction by the denominators in the form of 1
    %% Example: 2/5 + 3/6 = 2/5(6/6) + 3/6(5/5) = 12/30 + 15/30 = 27/30 (which is equal
    {(Numerator1 * Denominator2) + (Numerator2 * Denominator1), (Denominator1 * Denomin
%% {Integer, Fraction} version
%% Adding integers makes it easy because the base stays the same
add(Whole, {Numerator, Denominator}) ->
    %% 2 + 3/6 = 2(6/6) + 3/6 = 15/6 = 5/2
    {(Whole * Denominator) + Numerator, Denominator};
%% {Fraction, Integer} version
%% Because addition is commutative, this is the same as the other way
%% Commutative
add({Numerator, Denominator}, Whole) ->
    %% 2 + 3/6 = 2(6/6) + 3/6 = 15/6 = 5/2
    {(Whole * Denominator) + Numerator, Denominator}.
add(_, _, eager) -> 0.

%% Subtract two fractions together
sub({Numerator1, Denominator1}, {Numerator2, Denominator2}) ->
    %% First multiply the other fraction by the denominators in the form of 1
    %% Example: 2/5 - 3/6 = 2/5(6/6) - 3/6(5/5) = 12/30 - 15/30 = -3/30 (which is equal
    {(Numerator1 * Denominator2) - (Numerator2 * Denominator1), (Denominator1 * Denomin
%% {Integer, Fraction} version
%% Subtracting integers makes it easy because the base stays the same
%% Associative
sub(Whole, {Numerator, Denominator}) ->
    %% 2 - 3/6 = 2(6/6) - 3/6 = 15/6 = 5/2
    {(Whole * Denominator) - Numerator, Denominator};
%% {Fraction, Integer} version
sub({Numerator, Denominator}, Whole) ->
    %% 9/6 - 1 = 3/6 - 1(6/6) = -3/6 = -1/2
    {Numerator - (Whole * Denominator), Denominator}.
sub(_, _, _) -> 0.

%% Divide two fractions together
divide({Numerator1, Denominator1}, {Numerator2, Denominator2}) ->
    %% Example: (2/5)/(3/6) = (2/5)*(6/3) = 18/15 (which is 6/5)
    %% Do the swap and multiply all at once
    {(Numerator1 * Denominator2), (Denominator1 * Numerator2)};
```

```
%% {Integer, Fraction} version
%% Dividing an integer with a fraction is weird but we can do it!
divide(Whole, {Numerator, Denominator}) ->
    %% Example: 2 / (4/3) = 2(3/4) = 6/4 (which is 3/2) we take the reciprocal and mult
    %% Example: 8 / (5/7) = 8(7/5) = 63/5.
    %% Do the reciprocal and multiply all at once
    {(Whole * Denominator), Numerator};

%% {Fraction, Integer} version
%% Associative
divide({Numerator, Denominator}, Whole) ->
    %% (3/6) / 2 = 3 / (6 * 2) = 3/12 = 1/4
    {Numerator, (Denominator * Whole)}. %% The denominator gets bigger

divide(_, _, _) -> 0.

%% Multiply two fractions together
mul({Numerator1, Denominator1}, {Numerator2, Denominator2}) ->
    %% Example: (2/5)*(3/6) = 6/30 (which is 1/5)
    {(Numerator1 * Numerator2), (Denominator1 * Denominator2)};

%% {Integer, Fraction} version
%% Multiply two fractions together
mul(Integer, {Numerator, Denominator}) ->
    %% Example: 2*(3/6) = 6/6 (which is 1/5)
    {Integer * Numerator, Denominator};

%% {Fraction, Integer} version
%% Multiply two fractions together
%% Commutative
mul({Numerator, Denominator}, Integer) ->
    %% Example: 2*(3/6) = 6/6 (which is 1/5)
    {Integer * Numerator, Denominator}.

mul(_, _, _) -> 0.
```

Limitations

As we have already discussed, this algorithm I have presented is the slowest algorithm out of the 5 algorithms I have experimented with to compute factors of numbers. Why? Because it not only has many more arithmetic operations to perform at each loop, if we were to check if the fraction y_k was a factor of the number, we would have check if the numerator is divisible by the denominator, which is a modulo operation (slow in modern computing). If the remainder was zero, meaning the denominator can divide the numerator, then a integer division operation is needed to get the actual integer value of the fraction (not as slow as modulo but still slow compared to multiply or add operations). Thus, with the combination of many arithmetic operations to compute the next y_k and many operations to check if that y_k is a factor, this algorithm will be the slowest.

Since I used a fraction library, I needed to exploit fractional arithmetic which you learn in Algebra classes that seems not too bad. But simplifying fractions takes many operations! Without realizing it, we were doing many operations in our head (or at least retrieving from our memory those operations). We gained an intuition into how to simplify arithmetic operations with fractions well, but the computer has to do all the steps your brain did subconsciously. So I tried to make my fraction library arithmetic operations as quick as possible - performing no optimizations or simplifications. As such, the denominators of each consecutive y_k got larger and larger, because instead of finding the greatest common divisor for a fraction subtraction, I just multiplied both fractions by the other fraction's denominator. This gave growth greater than factorial. Thus the denominators would increase like 1*2 = 2, 2*3 = 6, 6*2 = 12, 12*4 = 36, 36*4 = ... and the numbers got bigger and bigger. So, without performing any simplification to my fractions, my Erlang system couldn't factor a number above 109.

Conclusions

Though this theorem doesn't enhance the speedup of factoring algorithms, it gives us great insights into the relationships between factors, relationships between each integer below the square root of a number, and great insights into potential facts into the mystery into prime numbers. I believe this theorem when studied and thought about over a lifetime will give birth to quicker prime number arithmetic operations, a deeper mathematically understanding, and better algorithmic design. Just because an algorithm is

slow and inefficient doesn't mean that is of no use. Each algorithm enhances our understanding of numbers and how they relate to each other. The very facts, axioms, and foundations of mathematics rely on simple arithmetic and factoring. The more we understand about those operations as well as high-level mathematics, the better citizens we will be. We will be more logical and reasonable. We will have better ideas and more insight. We will be able to solve problems faster than those around us. We will make the world a better place.

Thank you for reading.

Benjamin Hansen, April 2020, Brigham Young University - Idaho.