Finding Perfect Polynomials modulo 2

Ugur Caner Cengiz

October 20, 2014

Contents

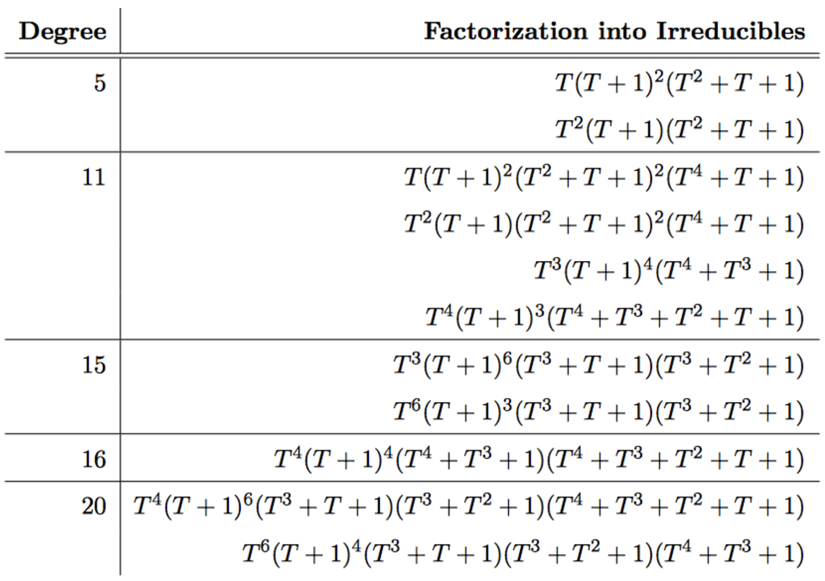
Chapter 1

Perfect Polynomials modulo 2

1.1 Introduction to the Problem

In 1941, E. F. Canaday published a part of his dissertation on perfect polynomials modulo 2. In this paper he regarded a perfect polynomial as a polynomial whose sum of divisors equals itself even though he called them "one-rings" [1]. If we let $\sigma(A) = \sum_{D|A} D$, then A is a perfect polynomial provided that $\sigma(A) = A$. Since the polynomials are mod 2, the coefficients were only able to be 1 or 0.

Canaday has identified an infinite class of perfect polynomials in addition to the 11 polynomials that do not fit in this class. The infinite class can be represented as $x^{2^n-1}(x+1)^{2^n-1}$ where n is a positive integer. The other perfect polynomials Canaday identified are shown in figure 1.1, which uses 'T' as a variable. Throughout the paper variable 'x' will be used instead.



Degree	Factorization into Irreducibles
5	$T(T+1)^2(T^2+T+1)$
	$T^2(T+1)(T^2+T+1)$
11	$T(T+1)^2(T^2+T+1)^2(T^4+T+1)$
	$T^2(T+1)(T^2+T+1)^2(T^4+T+1)$
	$T^3(T+1)^4(T^4+T^3+1)$
	$T^4(T+1)^3(T^4+T^3+T^2+T+1)$
15	$T^3(T+1)^6(T^3+T+1)(T^3+T^2+1)$
	$T^6(T+1)^3(T^3+T+1)(T^3+T^2+1)$
16	$T^4(T+1)^4(T^4+T^3+1)(T^4+T^3+T^2+T+1)$
20	$T^4(T+1)^6(T^3+T+1)(T^3+T^2+1)(T^4+T^3+T^2+T+1)$
	$T^6(T+1)^4(T^3+T+1)(T^3+T^2+1)(T^4+T^3+1)$

Figure 1.1: Canaday's list for perfects

Canaday noticed that all of the perfect polynomials he found were divisible by $x \times (x+1)$. He suggested that perfect polynomials modulo 2 exist in two forms: the ones that are divisible by $x \times (x+1)$ and the ones that are not. Since $x \times (x+1)$ looked similar to $n \times (n+1)$, which would always give an even integer, perfect polynomials divisible by $x \times (x+1)$ are called 'even' and the rest is called 'odd'. Even though Canaday did not use the term 'odd' for his one-rings, he proved that any odd perfect polynomial has to be a perfect square [1]. More recent work by Luis Gallardo and Olivier Rahavandrainy proves that an odd perfect polynomial would have at least 5 distinct prime factors [3].

In concurrence with his findings, Canaday conjectured that there are no odd perfect polynomials. He had found some infinite classes of perfects; however, and the highest degree of a perfect polynomial he found was 20. Higher degrees of polynomials are much harder to analyze. Nevertheless, by using computers it is possible to check higher degrees for perfects polynomials. In this report some of these programs will be discussed.

1.2 Some Useful Properties

An important property of σ function is that it is multiplicative over integers.

Lemma: If gcd(m,n) = 1, then σ(mn) = σ(m) $\times\sigma$(n).

Proof. Every positive integer can be uniquely factored using prime numbers.

Let $m = (\mathrm{p}_1^{\alpha_1}\mathrm{p}_2^{\alpha_2}\mathrm{p}_3^{\alpha_3}...\mathrm{p}_k^{\alpha_k})$ and similarly $n = (\mathrm{q}_1^{\beta_1}\mathrm{q}_2^{\beta_2}\mathrm{q}_3^{\beta_3}...\mathrm{q}_l^{\beta_l})$.

Therefore, all p_i and q_j are distinct since (m,n) = 1.

$mn = (\mathrm{p}_1^{\alpha_1}\mathrm{p}_2^{\alpha_2}\mathrm{p}_3^{\alpha_3}...\mathrm{p}_k^{\alpha_k})(\mathrm{q}_1^{\beta_1}\mathrm{q}_2^{\beta_2}\mathrm{q}_3^{\beta_3}...\mathrm{q}_l^{\beta_l})$

Let σ(m) = $(a_1 + a_2 + ... + a_t)$ and σ(n) = $(b_1 + b_2 + ... + b_z)$.

$$\sigma(\mathrm{m})\sigma(\mathrm{n}) = (\mathrm{a}_1(b_1+b_2+...+b_z)+a_2(b_1+b_2+...+b_z)+...+a_t(b_1+b_2+...+b_z)) \quad (1.1)$$

Because all p_i and q_j are distinct, every a_k and b_l are distinct except for 1. This implies that the sum (1.1) has every divisor of mn and no element is repeated. Therefore, the sum equals σ(mn). σ(mn) = σ(m)σ(n) □

Similar to positive integers every polynomial can be factored using irreducible polynomials of lesser degrees. Therefore, σ is also multiplicative over polynomials. This property is useful to prove a property in perfect polynomials modulo 2.

Theorem: A perfect polynomial A is divisible by x if and only if it is also divisible by $x+1$.

Proof. $\Rightarrow$

Let $x \mid A$ and σ(A) = A. For contradiction assume that $(x+1) \nmid A$.

Let $A = (x^{\alpha})\Pi(P_i^{\beta_i})$,

Then σ(A) = (x$^{\alpha} + x^{\alpha-1} + ... + x + 1)\Pi(P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1) = A$

(x+1)$\not|$ P_i and $x \not| \ P_i$ which imply $P_i(0) = P_i(1) = 1$. because of the Remainder Theorem in polynomials and we are using modulo 2. Notice that any complete polynomial with an odd degree is divisible by $(x + 1)$.

For example, let Q = $(x^{2n+1} + x^{2n} + ... + x + 1)$, then $Q = (x + 1)(x^{2n} + x^{2n-2} + x^{2n-4} ... + x^2 + 1)$

Therefore, α and all β_i have to be even.

We know $A = (x^{\alpha} + x^{\alpha-1} + ... + x + 1)\Pi(P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1)$

Clearly, $x \not| ($x$^{\alpha} + x^{\alpha-1} + ... + x + 1)$.

Since β_i is even $(P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1)(0) = (1 + 1 + 1 + ... + 1)$ where there are an odd number of 1's. Hence, $(P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1)(0) = 1$.

Thus, x does not divide either $(x^{\alpha}+x^{\alpha-1}+...+x+1)$ or $(P_i^{\beta_i}+P_i^{\beta_i-1}+...+P_i+1)$.

This implies $x \not| A$ which is an obvious contradiction to what we were given.

Therefore, if A is perfect and $x \mid A$, then $(x + 1) \mid A$.

$\Leftarrow$

Similarly, let $(x+1) \mid A$ and σ(A) = A. For contradiction assume that $x \not| A$.

Let $A = (x + 1)^{\alpha}\Pi(P_i^{\beta_i})$,

Then σ(A) = $((x+1)^{\alpha}+(x+1)^{\alpha-1}+...+(x+1)+1)\Pi(P_i^{\beta_i}+P_i^{\beta_i-1}+...+P_i+1) = A$. Also, $P_i(0) = P_i(1) = 1$.

α and all β_i have to be even because otherwise $((x + 1)^{\alpha} + (x + 1)^{\alpha-1} + ... + (x + 1) + 1)(0) = 0$ and $((P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1))(0) = 0$.

However, when α and all β_i are even both $((x+1)^{\alpha}+(x+1)^{\alpha-1}+...+(x+1)+1)(1)$ and $((P_i^{\beta_i} + P_i^{\beta_i-1} + ... + P_i + 1))(1)$ equal 1.

This implies $(x + 1) \not| A$ which is a similar contradiction.

Hence, if A is perfect and $(x + 1) \mid A$, then $x \mid A$.

Therefore, a perfect polynomial A is divisible by x if and only if it is also divisible by x+1. $\square$

This proves that perfect polynomials modulo 2 can be separated into even and odd perfect polynomials.

Chapter 2

The Programs

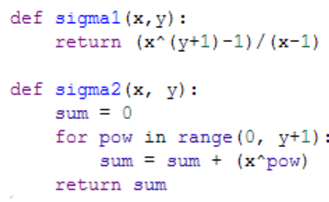
The main focus of this Richter 2014 project was to write a computer program capable of finding perfect polynomials modulo 2. System for Algebra and Geometry (Sage) was used to code the programs. Sage is free, mathematically inclined and uses Python, which is very easy to learn, to code. Sage offered a very easy solution. By running R.<x> = IntegerModRing(2)[] as the initial line of every workpage created on Sage, it was possible to use ‘x’ as a variable which created polynomials modulo 2 instead of recognizing ‘x’ as a symbol.

The initial aim for the project was to write a program to calculate the sum of the divisors of a polynomial. Then, it progressed into implementing a recursive algorithm for a more sophisticated way of searching. The last aim was to maximize the efficiency of the specialized programs for even and odd perfect polynomials.

2.1 Earlier Versions

2.1.1 A Program for Finding the Sum of Divisors

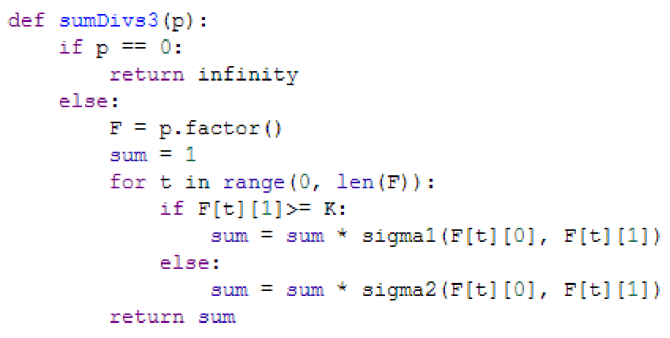
Using the multiplicative property of σ and checking the sum of divisors of each factor seemed to be much more effective than trying to find all the divisors of the initial polynomial and add them up. Therefore, two helper functions were developed named *sigma*1 and *sigma*2. *sigma*2 simply loops through every factor and adds them up whereas *sigma*1 uses a property that comes from calculus. If we let a factor of the main polynomial be P^α,
then $\sigma(P^\alpha) = (P^\alpha + P^{\alpha-1} + ... + P + 1) = \frac{P^{\alpha+1}-1}{P-1}$. As seen in Figure 2.1, input ‘x’ is the factor and ‘y’ is its respective power.



```
def sigma1(x,y):
    return (x^(y+1)-1)/(x-1)

def sigma2(x, y):
    sum = 0
    for pow in range(0, y+1):
        sum = sum + (x^pow)
    return sum
```

Figure 2.1: Functions *sigma*1 and *sigma*2

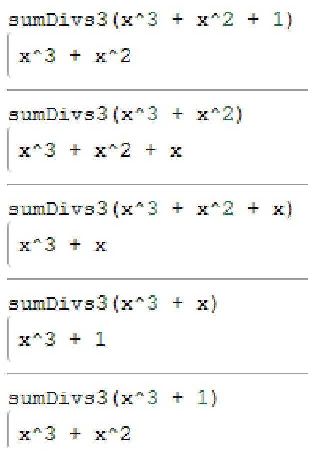
The results from calculating σ for each factor need to be multiplied to find the sum of divisors of the main polynomial. $sumDivs$(p) function uses $sigma1$ and $sigma2$ to do this.



```
def sumDivs3(p):
    if p == 0:
        return infinity
    else:
        F = p.factor()
        sum = 1
        for t in range(0, len(F)):
            if F[t][1]>= K:
                sum = sum * sigma1(F[t][0], F[t][1])
            else:
                sum = sum * sigma2(F[t][0], F[t][1])
        return sum
```

Figure 2.2: Function sumDivs3(p) uses $sigma1$ and $sigma2$ to calculate σ(p)

As we walk through the steps in the program, first the program checks if the polynomials is congruent to 0 or not. (This step was later omitted by the next generation of $sumDivs$() programs.) Then, it factors the polynomial and runs $sigma1$ or $sigma2$ for each factor according to 'K'. The 'K' is just a limiting point based on the power of a factor. The initial hypothesis was that after a certain point running $sigma1$ would be faster as the division in the formula would be easier than running $sigma2$. Then, all the results from the helper functions are multiplied to give the correct result for σ(p). Some examples are on Figure 2.3.

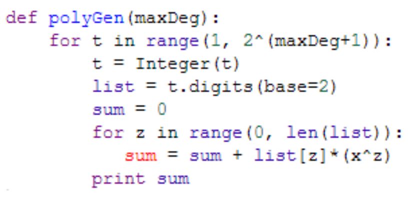


```
sumDivs3(x^3 + x^2 + 1)
 x^3 + x^2

sumDivs3(x^3 + x^2)
 x^3 + x^2 + x

sumDivs3(x^3 + x^2 + x)
 x^3 + x

sumDivs3(x^3 + x)
 x^3 + 1

sumDivs3(x^3 + 1)
 x^3 + x^2
```

Figure 2.3: Some Examples of $sumDivs3$(p) Calculations

2.1.2 A program for generating polynomials

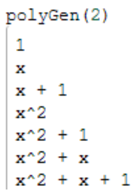
In the beginning of the project it was important to be able to generate polynomials mod 2 to run $sumDivs3()$ on them. The author's initial misconception was that we had to check every polynomial up to a certain degree with $sumDivs3()$ to check if it was perfect. Checking every polynomial up to a certain degree would be accurate but also inefficient. The author had already written a program to generate polynomials. A more efficient algorithm was used later.



```
def polyGen(maxDeg):
    for t in range(1, 2^(maxDeg+1)):
        t = Integer(t)
        list = t.digits(base=2)
        sum = 0
        for z in range(0, len(list)):
           sum = sum + list[z]*(x^z)
        print sum
```

Figure 2.4: $polyGen(MaxDeg)$ Creates Polynomials

In Figure 2.4, it shows that the polynomials are created by converting the base to 2 and regarding the number as a list if 1's and 0's which correspond to the coefficients of the polynomial. Therefore $2^{MaxDeg+1}$ - 1 would have MaxDeg + 1 number of 1's representing the coefficients from 1 to x^{MaxDeg}. In Python, every for loop starts from the initial number and ends right before it hits the final number entered in parentheses. Therefore, setting the loop from 1 to $2^{MaxDeg+1}$ makes sure that we produce every polynomials with maximum degree of $MaxDeg$. Changing the final statement from *print* to *return* or calculating $sumDivs()$ within the generator program ensures that the output created is not an expression but an operable polynomial with one variable.



```
polyGen(2)
1
x
x + 1
x^2
x^2 + 1
x^2 + x
x^2 + x + 1
```

Figure 2.5: Examples of Polynomials Created by $polyGen(MaxDeg)$

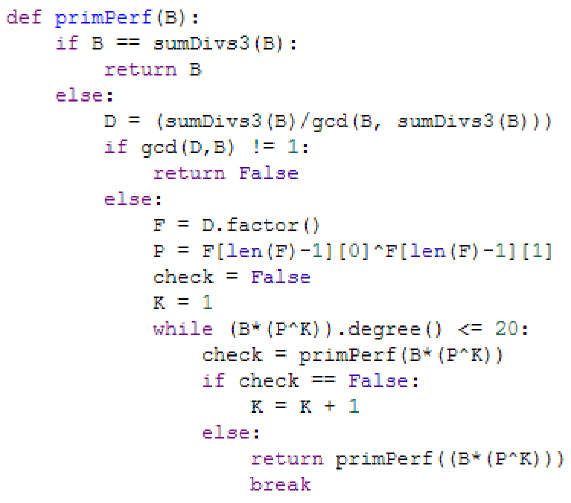
Figure 2.5 shows that the program works well and creates the polynomials.

2.1.3 Main Finding Algorithm

Even though creating polynomials up to a certain degree and checking each of them for the possibility of being perfect would be an accurate solution to the problem, it is very inefficient and it requires too many computations for higher degrees. Thankfully, there is a much more sophisticated algorithm created by Dr. Paul Pollack, who is an Assistant Professor in the Mathematics Department at the University of Georgia. The algorithm moves the focus from finding perfects to finding *primitive perfects* shortening the number of calculations.

Let A be *primitive perfect* if σ(A) $= A$ and A cannot be written as a product of co-prime polynomials BC where both B and C are also perfect. Let $A = B \times C$

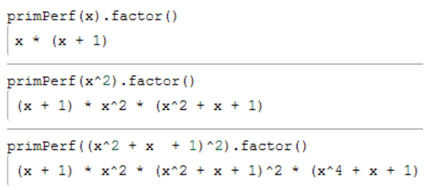
a non primitive perfect polynomial whereas B and C are primitive. Start the process with checking if σ(B) $= B$. If it is, then B is perfect and we can output it. If not, we calculate D where D = ((σ(B)) / (gcd(B, σB))). It's clear that $D \mid C$. Therefore, if $gcd(B, D) > 1$, then $gcd(B, C) > 1$ which means B and C are not co-prime, a contradiction to what we started with. Hence, we stop. If B and D have no common factor, then we let P be the greatest factor of D and restart the algorithm taking $BP, BP^2, BP^3, ..., BP^k$ where degree of $BP^k < K$. K is only a defined maximum degree. Therefore, this algorithm creates a recursive tree-like process looking for primitive perfects.



```
def primPerf(B):
    if B == sumDivs3(B):
        return B
    else:
        D = (sumDivs3(B)/gcd(B, sumDivs3(B)))
        if gcd(D,B) != 1:
            return False
        else:
            F = D.factor()
            P = F[len(F)-1][0]^F[len(F)-1][1]
            check = False
            K = 1
            while (B*(P^K)).degree() <= 20:
                check = primPerf(B*(P^K))
                if check == False:
                    K = K + 1
                else:
                    return primPerf((B*(P^K)))
                    break
```

Figure 2.6: primPerf(B) is the Initial Program Written for This Algorithm

$primPerf(B)$ basically does exactly what the algorithm says. It checks if B is perfect. If not, it calculates D and $gcd(B, D)$ and according to the result it finds P and restarts the algorithm up to degree 20. One of the biggest problems with this program was that after it found a perfect polynomial it did not stop. Therefore, the "false check" system was introduced. In the cases where the algorithm is supposed to stop and not return any output it returns *false*. Also, the *break* statement in the second part of the *while* loop stops the loop if it found a perfect. There were cases where $primPerf(B)$ returned nothing. Therefore, it was necessary to check the *type* of the output when a finder program was being run to print out all the results from $primPerf(B)$. Therefore, only the outputs who had the same type as x where printed and the program worked without a problem to find perfects.



```
primPerf(x).factor()
 x * (x + 1)

primPerf(x^2).factor()
 (x + 1) * x^2 * (x^2 + x + 1)

primPerf((x^2 + x  + 1)^2).factor()
 (x + 1) * x^2 * (x^2 + x + 1)^2 * (x^4 + x + 1)
```

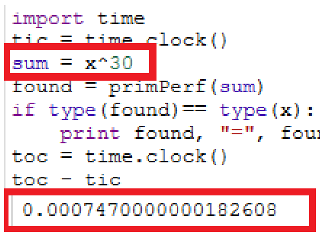
Figure 2.7: Some Results of $primPerf$(B)

As you can see from Figure 2.7, $primPerf$(B) finds a perfect polynomial and stops. The results presented here are already known perfects from Canaday's list. The results respectively are a member of the infinite class, 2^{th} item and 4^{th} item on Figure 1.1. Another curious point is that the input of the calculations was present as a factor of the polynomial. This is due to the calculations of P within the algorithm.

2.2 Improvements

2.2.1 Speed

When $primPerf(B)$ was first working on polynomials with higher degrees than 20 such as taking x^{30} as input and 200 as the limiting maximum degree, it was slow. It took 21 minutes to run $primPerf(x^{30})$ -and find no perfects of course-. Then, with Dr. Enrique Treviño's suggestion dynamic programming methods were implemented within the function. Dynamic programming is simply a trade off between memory and speed. Although it might be different and perhaps harder to do it with other software, Sage offered a very user friendly approach to dynamic programming. It has a built-in *cache* system which does exactly as needed. By adding the line "@CachedFunction" right under the definition line, the program starts saving a previously calculated input's result. In recursive functions this can immensely increase the speed. Figure 2.8 shows this.



```
import time
tic = time.clock()
sum = x^30
found = primPerf(sum)
if type(found)== type(x):
    print found, "=", found.factor()
toc = time.clock()
toc - tic
 0.000747000000182608
```

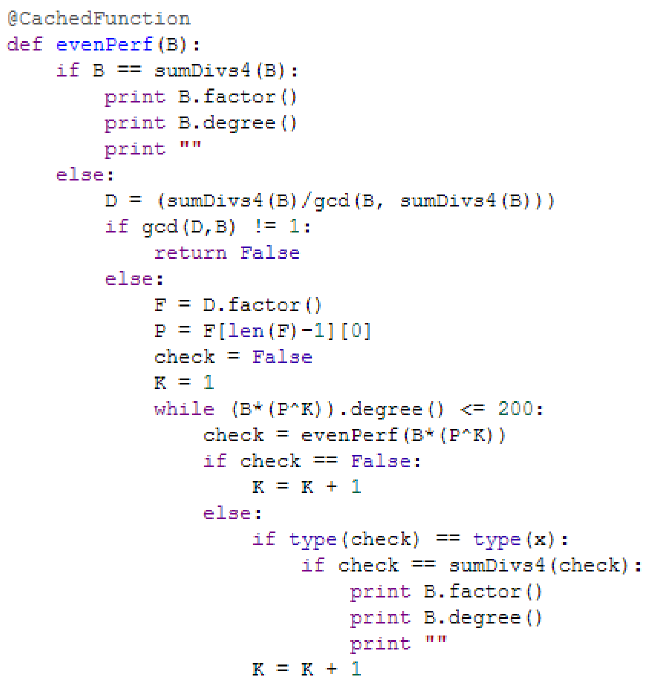
Figure 2.8: Increasing Speed by Saving Previous Calculations

It is mandatory to note here that this was the first time $primPerf(x^{30})$ was calculated after "@CachedFunction" was added because after the saving starts the results of the previously run inputs would be instantenous. Therefore, it is very important to clear the caches after changing the maximum degrees and before starting to run the program as otherwise the data is unreliable.

Another improvement on the speed was to increase the efficiency of $sumDivs3$(p). The original aim was to determine the cut off degree for power where $sigma2$ becomes faster than $sigma1$. However, the results of the tests showed that Sage is well equipped for doing high degree polynomial divisions and $sigma1$ was faster than $sigma2$ % 99 of the time when polynomials up to degree 100 were tested. The calculations which $sigma2$ completed faster showed no pattern or consistency. In each trial the polynomial that was calculated faster with $sigma2$ was different. Hence, in $sumDivs4$(p) only $sigma1$ was used. Currently these implementations prove sufficient for higher degree research; however, further improvements might be possible and required in the future.

2.2.2 More Specific Functions

Because of the structural differences between even and odd perfect polynomials, it was logical to further improve the original $primPerf(B)$ function into $evenPerf$(B) and $oddPerf$(B) functions.

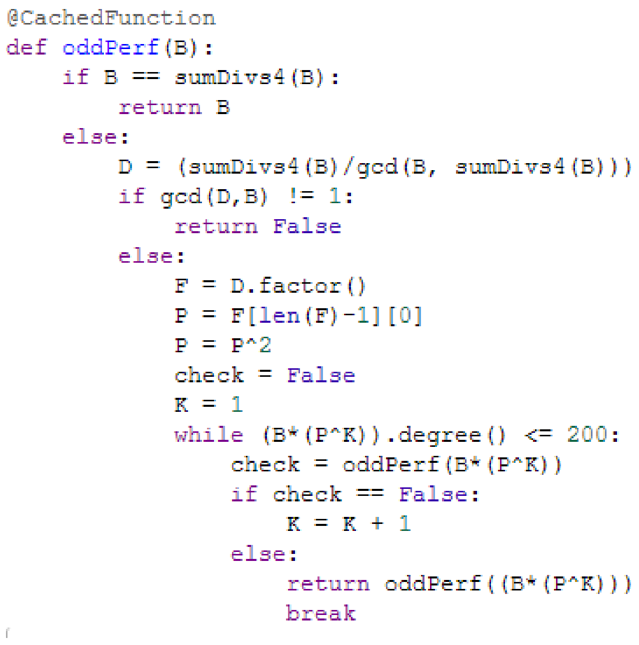


```
@CachedFunction
def evenPerf(B):
    if B == sumDivs4(B):
        print B.factor()
        print B.degree()
        print ""
    else:
        D = (sumDivs4(B)/gcd(B, sumDivs4(B)))
        if gcd(D,B) != 1:
            return False
        else:
            F = D.factor()
            P = F[len(F)-1][0]
            check = False
            K = 1
            while (B*(P^K)).degree() <= 200:
                check = evenPerf(B*(P^K))
                if check == False:
                    K = K + 1
                else:
                    if type(check) == type(x):
                        if check == sumDivs4(check):
                            print B.factor()
                            print B.degree()
                            print ""
                    K = K + 1
```

Figure 2.9: $evenPerf$(B) Function

The $evenPerf$(B) function specializes in finding even perfect polynomials. It takes seed polynomials in the form x^1, x^2, x^4, x^6 and x^{2n-1} and checks every resulting polynomial in the tree-like structure of the algorithm. Since the divisibility by x and $(x+1)$ are together the program also finds the polynomials divisible by $(x+1)$. Considering the infinite class of perfect polynomials the maximum degree has to be less than or equal to double the degree of the input. Another important point in this polynomial is that it does not stop once it finds perfect with the given seed polynomial. It goes up to checking the maximum degree with every seed. Therefore, it finds every perfect polynomial that the seed or P, the biggest factor of D which is calculated based on the seed polynomial, is associated with. This is because it is known since Canaday that some certain even perfect polynomials have the same factors. The inhibition of termination is done by increasing the power even if a perfect is found. In this system the perfects are printed instead of returned to the previous branch of the algorithm so that multiple outputs are produced.

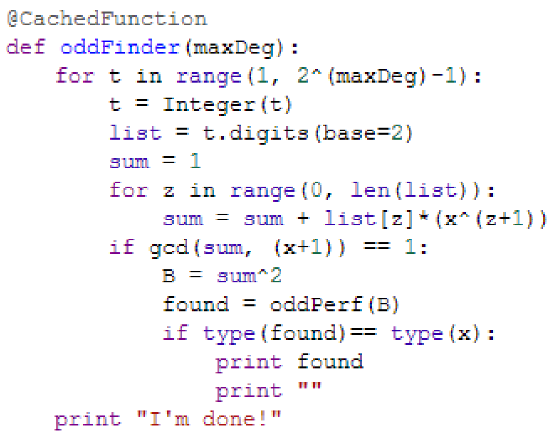
On the other hand, $oddPerf$(B) function is designed to stop once it finds anything. This is because it is believed that there are no odd perfect polynomials at all. If even one of them is found, that would mean that Canaday's conjecture is proven wrong with a counter example. In such a case the problem would be solved and perhaps the programs would be modified for further research.



```
@CachedFunction
def oddPerf(B):
    if B == sumDivs4(B):
        return B
    else:
        D = (sumDivs4(B)/gcd(B, sumDivs4(B)))
        if gcd(D,B) != 1:
            return False
        else:
            F = D.factor()
            P = F[len(F)-1][0]
            P = P^2
            check = False
            K = 1
            while (B*(P^K)).degree() <= 200:
                check = oddPerf(B*(P^K))
                if check == False:
                    K = K + 1
                else:
                    return oddPerf((B*(P^K)))
                    break
```

Figure 2.10: $oddPerf$(B) Function

As seen on Figure 2.10, the P value is squared. This is because it is known that an odd perfect polynomial has to be a perfect square. Even the initial finder program that creates polynomials and then calls $oddPerf$(B) squares the polynomials before running. Also, it is hard to miss that $oddPerf$(B) is designed to terminate once it finds anything very similar to its ancestor $primPerf$(B).



```
@CachedFunction
def oddFinder(maxDeg):
    for t in range(1, 2^(maxDeg)-1):
        t = Integer(t)
        list = t.digits(base=2)
        sum = 1
        for z in range(0, len(list)):
            sum = sum + list[z]*(x^(z+1))
        if gcd(sum, (x+1)) == 1:
            B = sum^2
            found = oddPerf(B)
            if type(found)== type(x):
                print found
                print ""
    print "I'm done!"
```

Figure 2.11: $oddFinder$() Function that Calls $oddPerf$(B)

This is the $oddFinder$() function that creates the polynomials to put in $oddPerf$(B). It is noticeable that first the polynomial's divisibility by $x \times (x+1)$

is checked and then it is squared before *oddPerf*(B) is used. Since it is also known that odd perfect polynomials have at least 5 distinct factors. Once the program uses *oddPerf*(B) on polynomials up to a certain degree k, it is capable of checking all the possible odd perfects up to degree $10k$. Therefore, if the maximum limiting degree within *oddPerf*(B) is 200, then only polynomials with degree up to 20 must be used as B.

2.3 Results

$$
\begin{array}{l}
x \times (x+1)^2 \times (x^2+x+1) \\
x \times (x+1)^2 \times (x^2+x+1)^2 \times (x^4+x+1) \\
(x+1) \times x^2 \times (x^2+x+1) \\
(x+1) \times x^2 \times (x^2+x+1)^2 \times (x^4+x+1) \\
x^3 \times (x+1)^4 \times (x^4+x^3+1) \\
x^3 \times (x+1)^6 \times (x^3+x+1) \times (x^3+x^2+1) \\
(x+1)^3 \times x^4 \times (x^4+x^3+x^2+x+1) \\
x^4 \times (x+1)^4 \times (x^4+x^3+1) \times (x^4+x^3+x^2+x+1) \\
x^4 \times (x+1)^6 \times (x^3+x+1) \times (x^3+x^2+1) \times (x^4+x^3+x^2+x+1) \\
(x+1)^3 \times x^6 \times (x^3+x+1) \times (x^3+x^2+1) \\
(x+1)^4 \times x^6 \times (x^3+x+1) \times (x^3+x^2+1) \times (x^4+x^3+1) \\
x \times (x+1) \\
x^3 \times (x+1)^3 \\
x^7 \times (x+1)^7 \\
x^{15} \times (x+1)^{15} \\
x^{31} \times (x+1)^{31} \\
x^{63} \times (x+1)^{63}
\end{array}
$$

Figure 2.12: Results up to Degree 200

Figure 2.12 shows the results of the programs up to degree 200. The initial data comes from *primPerf*(B) but *oddPerf*(B) and *evenPerf*(B) programs confirmed the results as well. The first 11 polynomials on Figure 2.12 are the exact ones from Figure 1.1. The last six are members of the infinite class of perfect polynomials modulo 2. Therefore, Canaday is absolutely correct, at least up to degree 200 polynomials. Not only there is no odd perfect polynomials but Canaday also found all the possible even perfect polynomials as well. This brings the question whether there are any other perfect polynomials at all. It is definitely a question for further research where higher degrees of polynomials should be checked.

Chapter 3

Reflections

One word to describe my summer for Richter 2014 would be amazing. Prior to this year and the Math 230 class with Dr. Treviño, I had never even thought about majoring Mathematics. I had only taken the class because understanding the basics of proofs sounded essential for any science. Now, I might even consider a post-graduate degree in Math because it was nothing but fun so far.

I had always enjoyed facing new problems and trying to solve them as long as they were not major life crises. This research project proved to be exactly what I needed to accelerate my learning experience. I have tested my limits and seen my strengths in necessary skills. For example, although coding was not something I had ever tried, I realized I can teach myself a lot to be able to code the programs I presented in this paper. Initially, Sage regarded x as an expression; therefore, the polynomial forms I entered into the worksheets were not mutable. Factoring was impossible. I searched for a way out and found out that Sage is even capable of creating the polynomials in a specific mod p which was 2 in our case. With factoring down, it was necessary to define the σ function on Sage. Coding a working *sumDivs*(p) program took me a week. After that, I thought I was done with most of the project since I was oblivious to Dr. Pollack's algorithm. Coding Dr. Pollack's algorithm was 100 times harder than working on *sumDivs*(p) since it had to be a recursive program which is an area that I did not learn much in my CS 100 level class where I was learning the very basics of Python. That also was manageable after long days of staring at a computer and using trial and error method. However, I have been very sceptical over my programs doubting that they work properly. My notebook is full of monologues, debating over issues that come up during the coding process and it has pages full of possible answers to the problems which are crossed out one by one as I try each and every one of them. Terminating $primPerf(\mathrm{B})$ was one of the hard problems I had to face, personally. However, in the end it worked (I think). Therefore, this summer definitely helped me to advance my research skills, coding confidence and absolutely my patience.

Also, working with Dr. Treviño on various proofs and understanding Canaday's paper was priceless. I don't think a college student gets a lot of opportunities to work one on one with a professor and ask any question. Being able to understand even only portions of Canaday's paper made me develop a keener curiosity for Mathematics. I definitely will be working on more Math problems in the future. I am grateful to everyone who made this summer possible.

List of Figures

Bibliography

[1] E.F. Canaday *The Sum of The Divisors of a Polynomial. Duke Mathematical Journal*, 8(4):721–737, 1941

[2] L. Gallardo. and O. Rahavandrainy. *Odd Perfect Polynomials over F_2 Journal de Théeorie des Nombres de Bordeaux*, 19(1):165–174, 2007.

[3] L. Gallardo. and O. Rahavandrainy. *There is no odd perfect polynomial over F_2 with four prime factors Portugaliae Mathematica*, 66(2):131–145, 2009.