Factorization in Numerical Semigroup Algebras

Sviatoslav Zinevich

June 2018

Contents

1 Introduction

For any field F, it is a fundamental property that any polynomial $p \in F[x]$ can be factored uniquely as a product of irreducible polynomials. This property does not hold, however, for semigroup algebras where some powers of x are forbidden; one instance being the semigroup algebra $F_2[x^2, x^3]$ of polynomials that do not contain x^1. For instance, x^6 has two factorizations into irreducibles: $x^2x^2x^2$ and x^3x^3. The emergence of such non-unique factorization leads to the question of effective factorization in $F[S]$, the semigroup algebra of polynomials whose terms all have exponents in the semigroup $S \subseteq \mathbb{N}$.

The first portion of this paper is dedicated to the issue of efficient computation of polynomial decomposition in arbitrary semigroup algebras. A python-based recursive solution is provided:

https://github.com/coneill-math/numsgpsalg

Recent research in the field of non-unique factorization in semigroup algebras has prompted a further discussion into the reducibility patterns of polynomials under such conditions. Specifically, we address the following question: what proportion of polynomials are reducible as the degree increases. In order to get viable results while checking reducibility of many polynomials, an efficient reduciblity check that did not require finding the full factorization was needed. The next portion of the paper discusses two such implementations and how they accomplish this goal.

The next part of the paper looks into computational evidence about reducibility patterns of polynomials in various rings generated by semigroups. A few different finite fields are used, along with some different semigroups, and an extensive computational overview of the proportion of irreducibles to reducibles is presented. An explanation about how the results were generated is also provided.

Finally, the results of the computational evidence are related to further analytical questions in the topic. Suggestions for future research are provided, and a brief overview of a proposed order in which they should be tackled is given.

2 Background Information

Definition 2.1. A *numerical semigroup* $S \subseteq \mathbb{N}$ is closed under addition and contains the zero element.

Example 2.2. Numerical semigroups can be generated by a finite list of generators. Given a set $\{g_1, g_2, ..., g_m\}$ of generators, the numerical semigroup generated by the set is given by

$$\langle g_1, g_2, ..., g_m \rangle = \{\textstyle\sum_{i=1}^{n} a_i g_i \mid a_i \subseteq \mathbb{N}\}.$$

That is, k is in the semigroup if there exists a combination of generators that sum up to k. The first few elements in the semigroup generated by $\langle 3, 5 \rangle$ are

$\{0,3,5,6,8,9,\ 10,12,15,...\}$. 0 is the identity element in every numerical semigroup, and $3,5$ are elementary, 6 is in the semigroup because $6 = 3 \cdot 2 + 5 \cdot 0$ and so on. Notice that 11 isn't in the numerical semigroup because there is no way to sum up 3 and 5 to yield 11.

Definition 2.3. A *field* F is a ring where the both operations $(+, \cdot)$ satisfy commutativity, associativity, the existence of additive and multiplicative identities $0, 1 \in F$, and the existence of inverses. In particular, if $e \in F$ then there exists $e_1 \in F$ such that $e + e_1 = 0$,and if e is nonzero, there exists e_2 such that $e \cdot e_2 = 1$.

Example 2.4. One field that is often encountered is $\mathbb{Z}_2 = \{0, 1\}$ that consists of two element, where addition and multiplication are both modulo 2. Another field is $\mathbb{Z}_5$, where addition and multiplication are modulo 5.

Definition 2.5. Fix a field F and semigroup S. The *semigroup algebra* $F[S]$ is the set of polynomials of the form $\sum_{i=0}^{k} a_i x^i$ where each $a_i \in F$, and $a_i = 0$ whenever $i \notin S$.

Example 2.6. $F_2[x^2, x^3]$ is the semigroup algebra over the field F_2 generated by $\langle 2, 3\rangle$, and can be written as $\{a_0 + \sum_{i=2}^{n} a_i x^i \mid a_i \in F_2, n \in \mathbb{N}\ \}$.

Definition 2.7. Fix a field F and a semigroup S. An element $f(x) \in F[S]$ is *irreducible* if its only divisors are the trivial divisors, the multiplicative identity element and itself.

Example 2.8. The semigroup algebra $F_2[x]$ is the semigroup algebra of all polynomials with coefficients in the field F_2. The elements $x + 1$ and $x^3 + x + 1$ are irreducible, while elements such as $x^2+1 = (x+1)^2$ and $x^3+x^2 = (x+1)\cdot x^2$ are reducible.

Definition 2.9. Fix field F and semigroup S. A *factorization* of an element E in a field $F[S]$ is the set of expressions of E as a product of irreducible elements, i.e. $\{\{e_1 e_2 ... e_n\} \subseteq F[S] \mid \Pi_{i=1}^{n} e_i = E\}$.

Example 2.10. In the semigroup algebra $F_2[x^2, x^3]$, the only factorization of $x^4 + x^2$ is $(x^2) \cdot (x^2 + 1)$. x^2 and $x^2 + 1$ are both irreducible. x^2 can only be reduced to $x \cdot x$ but x is not in the semigroup algebra and so x^2 is irreducible in $F_2[x^2, x^3]$. Similarly, $x^2 + 1$ can be decomposed to $(x + 1)^2$, which is not in the semigroup algebra and thus not a valid factorization of $x^2 + 1$. x^6 on the other hand has two factorizations $x^2 \cdot x^2 \cdot x^2$ and $x^3 \cdot x^3$. x^2 has been shown to be irreducible, and x^3 can be factored into $x \cdot x \cdot x$, which contains x which is not in the semigroup algebra, and $x^2 \cdot x$, which also contains x not in the semigroup algebra. Thus x^6 has two factorizations in $F_2[x^2, x^3]$.

3 Using the NumericalSemigroupAlg Package

This section will cover the class `NumericalSemigroupAlg` written in Python using the Sage open source package.

3.1 Basic Usage

The class itself contains two public methods, as well as three private methods and an initialization method.

When creating a new class instance, it is required to enter the semigroup generators in the form of a list, as the Sage package requires the list to generate the semigroup, as well as the size of the coefficient field. During initialization, the semigroup is created using the NumericalSemigroup package and is assigned to the `semigroup` class member. A ring is also generated using the Sage `PolynomialRing()` function. A class member `vars` is also created in order for the user to generate an object variable.

```
alg = NumericalSemigroupAlg([ 2 , 3 ] , 2)
alg
>> Numerical Semigroup Algebra object generated by the semigroup
>> [2, 3] in the semigroup algebra with 2 elements
x = alg.vars[0]
x^19 + x^14 + x^6 + x^4 + 1 in group
>> True
x^13 + x^7 + x^5 + 1 in group
>> False
```

In addition, a dictionary is created to store the factorization of every single polynomial generated. Thus, once a factorization of a polynomial has been computed, it won't have to be computed again later, saving resources. Using dynamic programming proves to be very useful, as polynomial decomposition is an inherently recursive process, where every reducible polynomial consists of the multiplication of smaller-degree polynomials. Having a dictionary changes the requirements of the algorithm, as it now has no need to shy away from computing multiple factorizations, but actually encourages them, since a comprehensive factorization dictionary enables $O(1)$ look-up. The `factorization()` function takes in a sage polynomial. This is the public method that generates all the factorizations of the polynomial, here named `element`, in the semigroup algebra. The inner workings of this function are detailed in Section 3.2.

```
group.factorization(x^9 + x^2 + 1)
>> [[x^4 + x^3 + 1, x^5 + x^4 + x^3 + x^2 + 1]]
group.factorization(x^11 + x^4)
>> [[x^2, x^2, x^7 + 1],
>> [x^2, x^2, x^3 + x^2 + 1, x^4 + x^3 + x^2 + 1],
>> [x^5 + x^3 + x^2, x^6 + x^4 + x^3 + x^2],
>> [x^3 + x^2, x^3 + x^2 + 1, x^5 + x^3 + x^2]]
```

The next function, `isIrreducible()`, checks if a given polynomial is irreducible. The optional second argument determines which implementation is used.

The default, `isIrreducible(p, True)` takes in a sage polynomial as input, and checks reducibility using a recursive algorithm. The recursive function `isIrreducible()` is a stripped down version of `factorization()`. If the polynomial is irreducible, it returns `True`, otherwise it returns `False`. It does so with an algorithm very similar to that of `factor()`. However, at the point where the factorization algorithm delves to find the factorization of the devising polynomials, `isIrreducible()` stops. It does not need to factor the divisors because their existence implies the original polynomial is reducible.

The second implementation of `isIrreducbile(p, False)`, also finds whether a polynomial is irreducible in the semigroup algebra provided in `__init__`. However, unlike the recursive `isIrreducible()` version, it does so without using recursion. This is accomplished by observing that the recursive algorithm simply looks for the right combination of irreducibles that form a polynomial in the semigroup algebra and divide the input polynomial into another polynomial in the semigroup algebra.

```
alg.isIrreducible( x^6 + 1 , True )
>> False
alg.isIrreducible( x^8 + x^7 + x^4 + 1 , True)
>> True
alg.isIrreducible( x^16 + x^9 + 1, False)
>> False
alg.isIrreducible( x^21 + x^19 + 1 , False)
>> True
```

3.2 Factoring $x^6 + 1$ using the `factorization()` member function

Overview. Upon a call to `factorization(input_var)`, a list of all possible factorizations of the input element is made using Sage's method `factor()`. This is when the private method `__Tree()` is called. The factorization is a largely recursive algorithm, and `__Tree(element, factorlist, master)` implements that by receiving a possible divisor of the element to be factored, its factor list, and the governing element, i.e. the element for which the recursive tree is finding the irreducible components, named here as master.

In general, `__Tree()` works by checking every possible expression of the master as a product of two smaller polynomials in the semigroup algebra. For each such expression, `__Tree()` calls on itself for each of the two factorizing polynomials with arguments: `element`, its factor list, and `element` again, as the algorithm now seeks to see if that polynomial can be further factored. This process keeps on going until `factorlist` is depleted, and if a factorization of the master has yet to be found, it is irreducible and is added to the dictionary as such. After `__Tree()` makes these calls, the recursive algorithm finds the irreducible components of the factorization, and they are joined to form a new list decomposition of the polynomial, which is then added to the dictionary.

More concretely, each call to `__Tree()` starts by checking if `factorlist` is empty. An empty `factorlist` implies that the `master`'s factorization reached its full depth , and without it having a factorization into at least two polynomials in the semigroup algebra, the current master is irreducible, as mentioned above. However, if `factorlist` isn't empty, then the algorithm checks if `element` is a factor of `master` in the semigroup algebra, by checking if it, in itself, is in the semigroup algebra, and if the remaining polynomial, `master/element`, is also in the semigroup algebra. If element isn't in the semigroup algebra, it is added to the dictionary with an empty list as it's decomposition, implying it's not in the semigroup algebra. If both conditions are true, the algorithm calls on another `__Tree()`, with both divisors as masters, to find out their decomposition recursively.

After the check for factorizations is done, a recursive call for `__Tree()` is made for every possible element that can be a factor of `master`, by composing all the elements in factor list minus each one, creating unique polynomials.

An in-depth example. The factorization of x^6+1 in $F_2[x^2,x^3]$ is $(x^4+x^2+1)\cdot(x^2+1)$ and $(x^3+1)\cdot(x3+1)$. In this part, a deep look will be taken into how this factorization is computed in the following call:

```
group.factorization(x^6 + 1)
>>[[x^2 + 1, x^4 + x^2 + 1], [x^3 + 1, x^3 + 1]]
```

First, a factor list is generated, namely $[x+1,x+1,x^2+x+1,x^2+x+1]$ and stored in `factorlist`. Then the first call to the main recursive function,

`__Tree(x^6+1,factorlist,x^6+1)`, is made.

The first iteration of the call results in nothing, as `master` is yet to be added to the dictionary and the possible factor `element`, equals `master`. Then, new divisors are proposed in the `for` loop, each unique element of `factorlist` is removed, and a polynomial of the remaining factors, is constructed and passed to `__Tree()`. Note that duplicate polynomials are not passed as we iterate over `set(factorlist)`, so each unique factor is generated exactly once. Starting with the polynomial that has x^6+1's factor list with the first element removed,

$$[x+1,\, x^2+x+1,\, x^2+x+1]$$

. The polynomial is simply the multiplication of these generating polynomials, $x^5+x^4+x^3+x^2+x+1$. Since x^6+1 has only two different polynomials in its `factorlist`, there is only one pending call, a call with the factoring element with factor list $[x+1,\, x+1,\, x^2+x+1]$, that is, x^4+x^3+x+1 (Figure 1).

In the first recursive call

`__Tree(x^5 + x^4 + x^3 + x^2 + x + 1, its factorlist, x^6 + 1),`

the program verifies that `factorlist` is not empty and the element does not equal `master`, and continues to check if `element` is in the semigroup algebra. Since $x^5+x^4+x^3+x^2+x+1$ is not in the semigroup algebra, the program

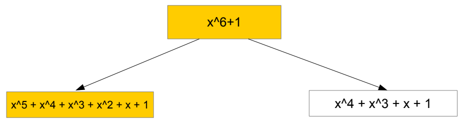


Figure 1: The first recursive call to `__Tree`. The orange boxes denote currently executing calls, while the white boxes denote pending calls.

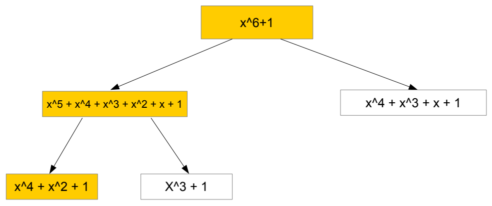


Figure 2: The next recursive calls to `__Tree`. The orange boxes denote currently executing calls. The white boxes denote pending calls.

continues to create further recursive calls, just as before. `element`'s factorlist is $[x+1, x+1, x^2+x+1]$ so it will generate two recursive calls to `__Tree()`, First with element $(x^2+x+1)\cdot(x^2+x+1) = x^4+x^2+1$ and then with element $(x+1)\cdot(x^2+x+1) = x^3+1$ (Figure 2).

In the first call the element, x^4+x^2+1, is in the semigroup algebra. After checking that `master / element` $= x^2+1$ is also in the semigroup algebra, the algorithm checks if x^2+1 is in the dictionary. Since it is not, a call to

```
__Tree(x^2 + 1, [x + 1, x + 1], x^2 + 1)
```

is made. Notice that the master is now x^2+1, as the algorithm now focuses on finding its factorization, in order to add the complete decomposition to x^6+1's dictionary entry. In the first call, the proposed factorization element x^2+1 is again equal to its master, and a recursive call is made again. This time `factorlist` is [x + 1, x + 1], so only one call with `element` $= x+1$ is made. $x+1$ is not in the semigroup algebra so an empty entry is added to the dictionary with $x+1$ as key, indicating that it isn't included in the group. Yet another call is made to `__Tree()` with x + 1 as the element, but this time `factorlist` is empty. Since it is empty, and `master`$= x^2+1$ is not yet in the dictionary, then it must be an irreducible in the semigroup algebra, and is thus added as such. The function then returns all the way back to the original call with x^6+1 as `master` (Figure 3).

The program proceeds to check if x^4+x^2+1 itself is in the dictionary. It isn't, so a call

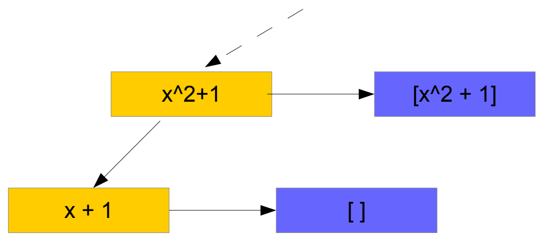


Figure 3: The `__Tree()` call for dictionary value addition of `master/element`. The orange boxes denote currently executing calls. The purple boxes indicate dictionary additions.

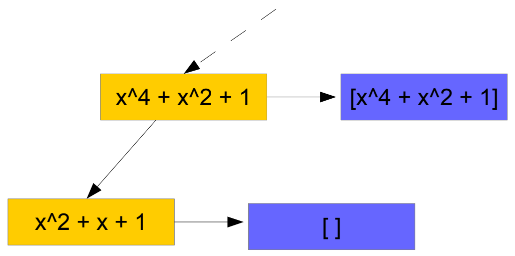


Figure 4: The call for dictionary value addition of `element`. Orange boxes indicate currently executing calls. Purple boxes indicate dictionary additions.

```
__Tree(x^4 + x^2 + 1, [x^2 + x + 1, x^2 + x + 1], x^4 + x^2 + 1)
```

is made to find out its decomposition. Since the factor list is $[x^2 + x + 1, x^2 + x + 1]$, a call with $x^2 + x + 1$ as the element will be made, and as $x^2 + x + 1$ is not in the semigroup algebra, it will be added to the dictionary with an empty list as its value, and the dictionary value of $x^4 + x^2 + 1$ will be itself, as it is irreducible (figure 4).

The program returns back to the original `__Tree` call, with $x^6 + 1$ as `master`. After decomposing both `element`, $x^4 + x^2 + 1$, and `master/element`, $x^2 + 1$ and adding them to the dictionary, the program can now add the new-found factorization of $x^6 + 1$ to it's dictionary. This is accomplished by iterating over every factorization of `element` and `master/element`, in this case only $x^2 + 1$ and $x^4 + x^2 + 1$, and adding them as a unique factorization of $x^6 + 1$ in the dictionary (Figure 5).

The algorithm now goes further into the other possible factorizations of $x^6 + 1$ from smaller polynomials constructed from the factors of $x^4 + x^2 + 1$. Since its factorlist is $[x^2 + x + 1, x^2 + x + 1]$, only one call is made,

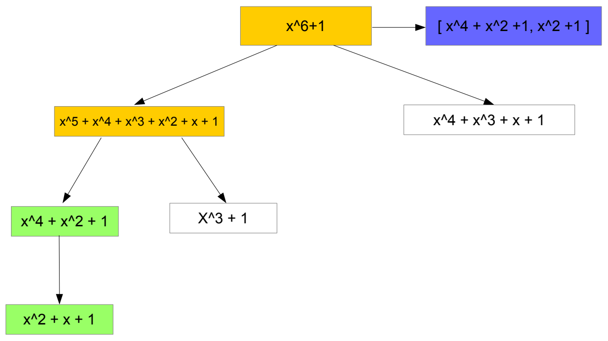


Figure 5: Further searches for possible factorization. Orange boxes indicate currently executing calls. Purple boxes indicate dictionary additions

```
__Tree(x^2 + x + 1, [x^2 + x + 1], x^6 + 1).
```

In that call, nothing new will be done as x^2+x+1 is known not to be in ring and its factorlist will be itself, and another identical call will be made to `__Tree()`, which will return due to an empty factorlist. Thus, the program will return and go into the call to x^3+1 as the dividing element,

```
__Tree(x^3 + 1, [x + 1, x^2 + x + 1], x^6 + 1).
```

The factorlist of x^3+1 is $[x+1,x^2+x+1]$, which isn't empty. The program then checks if it is in the semigroup algebra, and then if master/element, x^3+1, is in ring. Since the answer to both inquiries is True, a call

```
__Tree(x^3 + 1, [x + 1 ,x^2 + x + 1], x^3 + 1)
```

is made to find its decomposition.

Since `element == master`, calls to find the possible factors of x^3+1 is made, one for each possible factor. `factorlist =` $[x+1,\ x^2+x+1]$, so one call with each factor as element will be made. It is apparent that both are not in the semigroup algebra so they will be added to the dictionary with an empty list as a value, and `x^3+1` will be added as the only element in its list, as it is irreducible (Figure 6).

Now the program returns back to the main tree, i.e where master is x^6+1. The program checks if element is in the dictionary, and because element is x^3+1 and it has just been added to the dictionary, the program continues on without doing any additional calculations. In the next step, a new unique factorization is added to the master's dictionary value list, $[x^3+1,x^3+1]$.

The algorithm requires to check if there are other possible factors of x^6+1 that can be made using some of the factors of x^3+1. However, x^3+1 has a

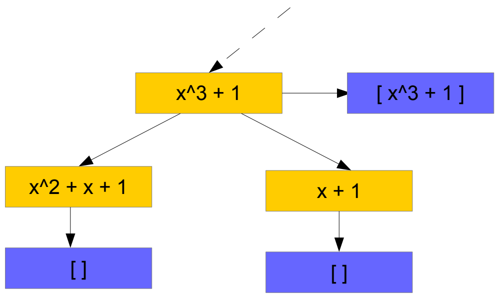


Figure 6: The call for dictionary value addition of `element`. Orange boxes indicate currently executing calls, Purple boxes indicate dictionary additions

factor list $[x + 1,\ x^2 + x + 1]$, which already has been examined above to have only polynomials not in the semigroup algebra and thus irrelevant.

Both x^4+x^2+1 and x^3+1 have been explored, and the algorithm returns all the way back to the very first `__Tree()` call, where it will now check x^4+x^3+x+1 as a possible factor of $x^6 + 1$ with

```
__Tree(x^4 + x^3 + x + 1, its factorlist, x^6 + 1).
```

In this function the program checks if $x^4 + x^3 + x + 1$ is in the semigroup algebra, and since it isn't, the program continues continues to find smaller possible factors. Its factor list is $[x + 1,\ x + 1\ ,\ x^2 + x + 1]$, and so there will two calls with elements $x^3 + 1$, $x^2 + 1$: (Figure 7).

```
__Tree(x^3+1, [x + 1, x^2 + x + 1], x^6 + 1)
    __Tree(x^2+1, [x + 1, x + 1], x^6 + 1)
```

The program will go into the first call, the one with `element` being $x^3 + 1$. Here, the same will occur as with it did with $x^3 + 1$ prior. However, since the dictionary entry for $x^3 + 1$ already exist, there is no need for computing its decomposition. At the phase where new decompositions are added to the master's, $x^6 + 1$, dictionary value, the values of element and `master/element`, both $x^3 + 1$ will be compared to the existing decompostions, and since $[x^3 + 1,\ x^3 + 1]$ is already in the dictionary, it will not be added.

Afterwards, smaller possible factorizations that consist of the factors of x^3+1 will be checked, but since its factor list is $[x + 1\ ,\ x^2 + x + 1]$ it is clear that they are not in the semigroup algebra and thus will not have any influence on the results.

Next, the program will return to the `__Tree()` call with $x^4 + x^3 + x + 1$ as element, and a call

```
__Tree(x^2+1, [x^2 + 1], x^6 +1)
```

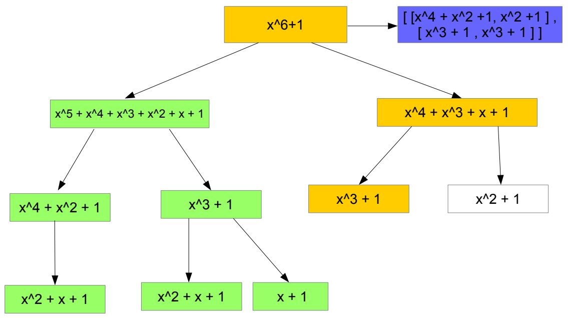


Figure 7: Search for further decompositions. Orange boxes denote currently executing calls, purple boxes denote dictionary additions, and green boxes show fully executed calls

is made. Here, it is found that both `element`, x^2+1, and `master/element`, x^4+x^2+1, are in the semigroup algebra, and thus are candidates for a factorization of x^6+1. Since both already have their decomposition in the dictionary, the program proceeds to try and add them to the x^6+1's dictionary list. However, this factorization is already in the dictionary, so no new factorizations are added. After this step, the program continues to look for possible factorizations with smaller polynomials that have some of the factors of x^2+1. The factor list of x^2+1 is $[x+1,\ x+1]$, and both factors aren't in the semigroup algebra so it can be easily deducted that they're not going to be factors of x^6+1.

The program then returns all the way up to the original call to `__Tree()` by `factorization()`. Since all possible factorizations of x^6+1 have been checked, it is safe to claim that its dictionary entry now contains the complete decomposition of x^6+1, and thus it is returned to the caller of this method (Figure 8).

3.3 Reducibility check of $x^8+x^6+x^4+1$ using the isIrreducible recursive implementation

Overview. Upon a call to `isIrreducible(polynomial)`, the algorithm generates the factor list of the polynomial, and binds a new class member `self.isRed` to a `True` value. This implementation assumes the polynomial is irreducible, and if it finds out that it isn't, it returns. Then, a call to

```
__reducibilityTree(element.factor(),factorlist)
```

is made. In each call to `__ReduciblityTree()`, the algorithm first checks if factorlist is empty or if `self.isRed == False`, and returns back if either is found to be true. In the former case, an empty factorlist signifies that the bottom of the recursion has been reached and the element is the trivial divisor, and

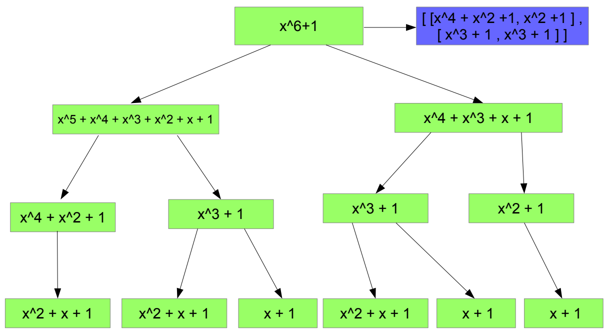


Figure 8: Fully searched recursive tree for the decomposition of $x^6 + 1$. Orange boxes denote currently executing calls, purple boxes denote dictionary additions, green boxes show fully executed calls, white boxes indicate pending calls

thus no factorization has been found. The latter is a check if a factorization has been found, since in that case the call to `isIrreducible()` has fulfilled it's goal in figuring out if it's input element is irreducible, and thus there is no need for further computation. `__ReducibilityTree()` is the recursive function that traverses down the possible factors of the input polynomial. Just like `factorization()`, `__reducibilityTree()` checks if `element` is in the semigroup algebra, and if `master/element` is in the semigroup algebra. If both are in the semigroup algebra, then `master` is factorable and the function can return to `isIrreducible()` and return `False`. Otherwise, it continues to transverse down the possible divisors in a fashion identical to that of `factorization()`.

An in-depth example. The polynomial $x^8 + x^6 + x^4 + 1$ in $F_2[x^2, x^3]$ is reducible. Below is a review of how the `isIrreducible()` recursive function verifies reducibility in the following call:

```
group = NumericalSemigroupAlg([ 2 , 3 ] , 2)
x = group.vars[0]
group.isIrreducible( x^8 + x^6 + x^4 + 1, True )
>> False
```

The polynomial $x^8+x^6+x^4+1$ is reducible in the semigroup algebra generated by $[x^2,\ x^3]$ in the semigroup algebra mod 2. It's decomposition is non-unique, and is $(x^2 + 1) \cdot (x^6 + x^2 + 1)$ and $(x^4 + x^3 + x^2 + 1) \cdot (x^4 + x^3 + x^2 + 1)$. upon calling the member function

`isIrreducible(x^8+x^6+x^4+1,True)`

the program runs the same preparation steps as `factorization()`. First, the semigroup algebra is generated using the sage function `PolynomialRing()`. Following these steps, a factorlist $[x + 1,\ x + 1,\ x^3 + x + 1,\ x^3 + x + 1]$ is

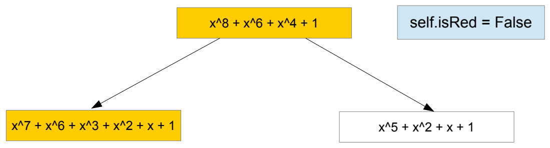


Figure 9: Recursive call of __reducibilityTree. Orange boxes denote currently executing calls, white boxes indicate pending calls

made to transverse down the divisors of $x^8 + x^6 + x^4 + 1$. Then the command `self.isRed = True` indicates that, unless shown otherwise, the input polynomial is, in fact, irreducible. Then, a call is made:

```
__ReducibilityTree(self, element,factorlist)
```

Next, a check is made if `element` and `master/element` are in the semigroup algebra, much like the one done in `factorization()`. If both conditions hold, then a reducibility has been proven, as two polynomials that are in the semigroup algebra exist which together multiply to create the input polynomial. The first call is false, as element is equal to master.

Next, a for loop call on `__ReducibilityTree()` is made for each possible divisor of $x^8 + x^6 + x^4 + 1$. Since `factorlist` contains two pairs of identical polynomials, only two such calls are necessary, as more calls will just produce duplicates. A first call is made on $(x+1)\cdot(x^3+x+1)\cdot(x^3+x+1) = x^7+x^6+x^3+x^2+x+1$, followed by a call on $(x+1)\cdot(x+1)\cdot(x^3+x+1) = x^5+x^2+x+1$, which will be made after the recursive call on

```
__ReducibilityTree(x^7 + x^6 + x^3 + x^2 + x + 1)
```

(Figure 9). In the recursive call, factorlist is not empty, and `self.isRed` is `False`, so the program continues to check if $x^7 + x^6 + x^3 + x^2 + x + 1$ is in the semigroup algebra by calling `__checkSemigroup(element)`. Since it returns `False`, the program continues without checking if `master/element` is in the semigroup algebra. The program continues then to make further recursive calls to find divisors in the semigroup algebra. Since factorlist is $[(x+1), (x^3+x+1), (x^3+x+1)]$, two calls will be made, one with element being $(x^3+x+1)\cdot(x^3+x+1) = x^6+x^2+1$ and another with $(x+1)\cdot(x^3+x+1) = x^4+x^3+x^2+1$ as the dividing element (Figure 10).

In the next recursive call, the program first checks if factorlist is empty or `self.isRed == True`. Since both conditions are not satisfied, the program continues. It then inquires if either element, $x^6 + x^2 + 1$, is in the semigroup algebra. Since it is, the program checks if `master/element`, namely $x^2 + 1$, is in the semigroup algebra. Since it is, there are at least two polynomials in the semigroup algebra that compose the input element, so `master` is reducible. The program then changes `self.isRed = True` and returns to the caller function.

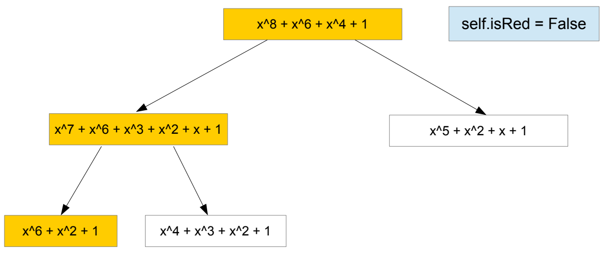


Figure 10: Recursive calls of `__reducibilityTree()`. Orange boxes denote currently executing calls, white boxes indicate pending calls

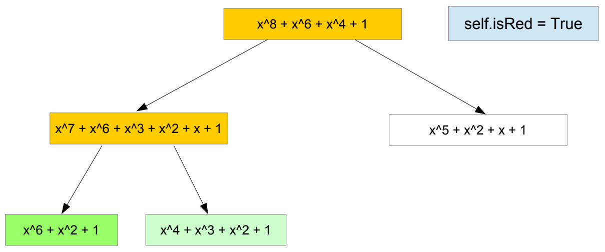


Figure 11: Full execution of irreduciblity check using `isIrreducible()`. Orange boxes denote currently executing calls, green boxes show fully executed calls, and white boxes indicate pending calls. Transparency indicates calls that returned since `isRed == True`

The program is not aware of the change in value of `self.isRed` and thus makes a recursive call to $x^4 + x^3 + x^2 + 1$ as the dividing element. In the new call, the program checks if `factorlist` is non empty as if `self.isRed == True`. Since the second condition is satisfied, the program returns to the caller function without doing any calculations (Figure 11).

The program is back in the call with element as $x^7+x^6+x^3+x^2+x+1$ during its recursive call phase. As there are no other recursive calls to make, the function returns to it's caller, the original call to `__recursiveTree()`. Here, too, the program does not know yet that the input polynomial has been shown reducible, so it calls on `__recursiveTree()` with x^5+x^2+x+1 as `element`. In the recursive call the program checks again if factorlist is empty and if `self.isRed == True`, which holds, so the program returns to the previous call.

The original call now finished cycling through possible divisors, so it returns to the user call to `self.isIrreducible()` and returns `self.isRed`, letting the user know that $x^8 + x^6 + x^4 + 1$ is in fact reducible(Figure 12).

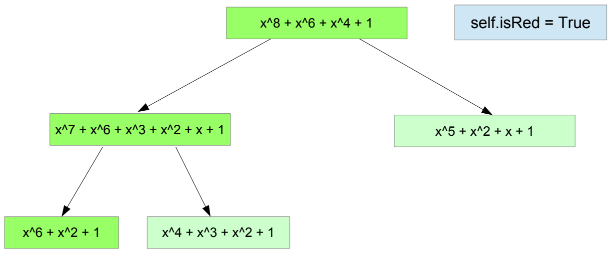


Figure 12: Orange boxes denote currently executing calls, Green boxes show fully executed calls, white boxes indicate pending calls. Transparency indicates calls that returned since `isRed == True`.

3.4 Reducibility check of $x^8 + x^6 + x^4 + 1$ using the non-recursive implementation of `isIrreducible()`

Overview. If a polynomial is factored into all its comprising irreducible polynomials, then the reducibility of the polynomial implies there exist a diving polynomial comprised of some of the factors of the original polynomial. This subset of factors is guaranteed to be in the powerset of the factor list of the original polynomial. Thus, given reducible polynomial P with factor list $F_P = \{f_1, f_2, ..., f_n\}$, the powerset $PS(F_P)$ will contain at least one factorization of a polynomial p in the semigroup algebra such that P/p is also in the semigroup algebra (and has a factorization in the powerset as well).

Lemma 3.1. Any reducible polynomial P with factor list F has at least one factor p_1 with factor list f_1 such that $|f_1| \leq |F|/2$

Proof. Given a reducible polynomial P with factorlist $F_P = \{f_1, f_2, ..., f_n\}$, and one possible factorization $p_1 p_2 = P$, let f_1 be the factorlist of p_1 and f_2 be the factor list of p_2. Then $f_1 \bigcup f_2 = F$ has to hold. Thus $|f_1| + |f_2| = |F|$. Thus if $|f_1| \geq |F|/2$, then $|f_2| \leq |F|/2$ has to hold, and otherwise $|f_1| \leq |F|/2$ and the lemma is proven □

From Lemma 3.1 it follows that it is sufficient to iterate through only the combinations of reducibles of F_p of length $\leq |F_P|/2$. And so, the non recursive function works in a straightforward manner. First, it iterates through the set of factorlist, looking if there is a simple irreducible in the semigroup algebra such that the input element divided by the irreducible produces a polynomial in the semigroup algebra, thus indicating the input polynomial is in the semigroup algebra. If no such polynomial is found, a combinatorial approach is taken. For every combination length from 2 to `floor(len(factorlist/2))`, all the possible combination are produced of that length. Each combination is a factorlist of a polynomial that is a possible divisor of the original polynomial. That factorlist gets reduced to that polynomial, and a check is made whether the polynomial is in the semigroup algebra, and if so is the complement that will result in the

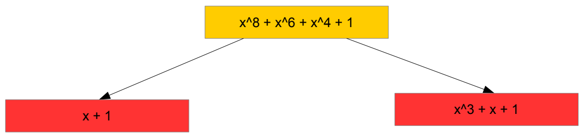


Figure 13: The call to all combinations of size 1 were not in the semigroup algebra and thus did not indicate reducibliity. Red boxes denote factorization checks that returned `False`, orange boxes denote currently executing call

original polynomial upon multiplication is in the semigroup algebra. If yes, then the original polynomial is certainly reducible and the function returns the value `False`. Otherwise, the function keeps running until all the combinations from all the lengths are exhausted. Once that happens, it is implied the original polynomial is irreducible, and thus the function returns with the value `True`.

An in-depth example. The polynomial $x^8 + x^6 + x^4 + 1$ in $F_2[x^2, x^3]$ has been shown previously to be reducible. Below is a review of how the `isIrreducible()` non-recursive function verifies reducibility in the following call:

```
group = NumericalSemigroupAlg([ 2 , 3 ], 2)
x = group.vars[0]
group.isIrreducible( x^8 + x^6 + x^4 + 1 , False )
>> True
```

Upon receiving an input polynomial, the program creates a factor list, just like the one in `isIrreducible()` and `factorization()`. The factor list of $x^8+x^6+x^4+1$ is $[(x+1), (x+1), (x^3+x+1), (x^3+x+1)]$. To find out the number of elements to make combinations of, the program runs `floor(len(factorlist)/2)`. Since the length of factorlist is 4, only combinations of 1 and 2 elements out of the 4 elements in factorlist need to be reviewed. It then proceeds to iterate over the set of factorlist, namely $[(x + 1), (x^3 + x + 1)]$. First, the program checks $x + 1$. A call to `__checkSemigroup(x + 1)` reveals that it is not in the semigroup algebra, so the program continues to check $x^3 + x + 1$. Another call to self `__checkSemigroup(x^3 + x + 1)` reveals that it is also not in the semigroup algebra. The program then proceeds to iterate through all the possible combinations (Figure 12).

The program then proceeds to look at combinations of size 2. The set of these combinations is $[x^2 + 1, x^4 + x^3 + x^2 + 1, x^6 + x^2 + 1]$. The program first checks if $x^2 + 1$ is in the semigroup algebra. Since it is, it checks if $(x^8 + x^6 + x^4 + 1)$ / $(x^2 + 1) = x^6 + x^2 + 1$ is in the semigroup algebra. As the check also returns `True`, the input element $x^8 + x^6 + x^4 + 1$ is reducible, and the function returns `False` (Figure 13).

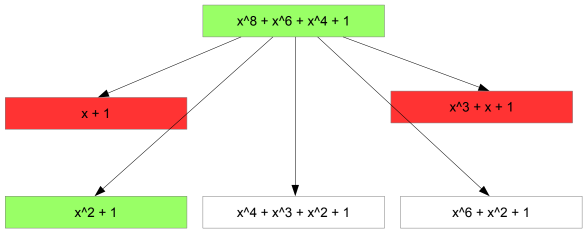


Figure 14: After finding a factorization the function returns True. Red boxes denote factorization checks that returned False, green boxes indicate a factorization check that verified reducibility, white boxes indicate calls that were not executed due to reducibility already being established

4 Data on Atomic Density

4.1 Methodology

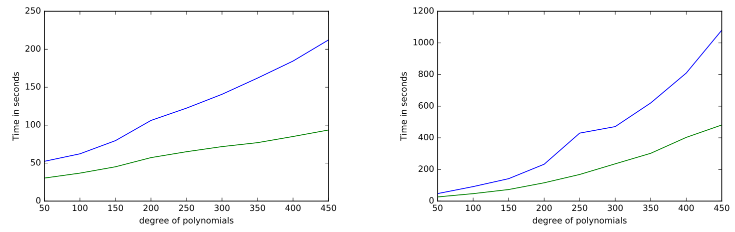
When exploring reducibilty patterns in higher degrees, exhaustive searching over all the polynomials of a certain degree is unfeasible, as they will take much too long. For a degree N, there are 2^N polynomials of that degree.

In order to get meaningful results about the proportion of reducible polynomials of higher degrees, a sampling algorithm had to be implemented. The algorithm iterates over a range of degrees. For every specified degree, it generates $10,000$ polynomials and runs a reduciblity check on each.

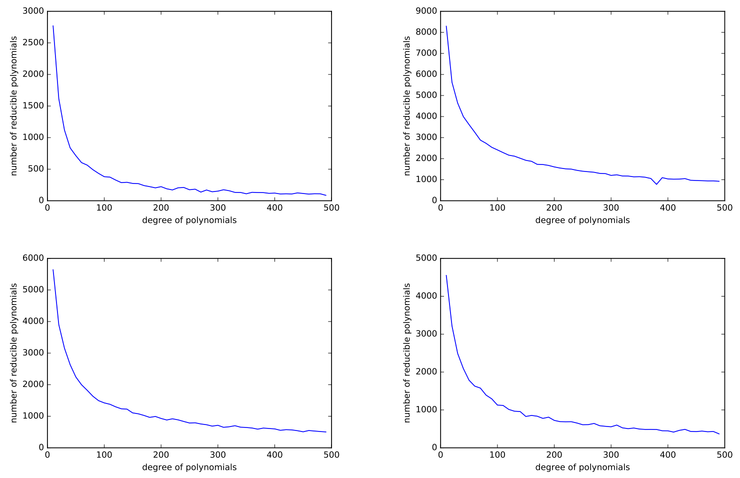
Since the reducibility check algorithm has to run over hundreds of thousands of polynomials, it is imperative to devise an efficient algorithm. The recursive `isIrreducible()` algorithm was the first attempt at implementing a viable algorithm. It is a simplified version of the recursive factorization algorithm, as described in the previous section. The second algorithm takes a different approach, and directly iterates over the possible factors of the input polynomial.

In order to find which is more efficient, 10000 polynomials were generated from degree 50 to 500 in intervals of 50 at the time. The same polynomials were checked for the reducibility by the two algorithms and the time that each implementation took to calculate if the input polynomials were reducible was recorded and later plotted. In the first trial, both were checked for reducibility of polynomials in the semigroup algebra $F_2[x^2, x^3]$.

According to the top left graph depicted in figure 15, the non-recursive `isIrreducible()` is about twice as fast as recursive version. Furthermore, it seems that both implementations have a linear increase in time taken as the degree increases. The superiority of the non-recursive implementation can be explained as a result two factors. First, a major decrease in computation speed arises for the non-recursive version due to not iterating over all possible divisors of the input polynomial, like the recursive version does, but rather over at most half of them. This process cuts down the time for guaranteeing a non-reducible



Left: Time plot w.r.t increasing degree in ring 2. Right: Time plot w.r.t degree in ring 4



Upper left: $F_2[x^2, x^3]$. Upper right: $F_2[x^4, x^5, x^6, x^7]$. Lower left: $F_5[x^2, x^3]$. Lower right: $F_4[x^2, x^3]$

Figure 15: Plots for Data on Atomic Density

polynomial is really so, as non-reducible polynomials will force the program to exhaust the list of possible divisors it iterates over.

Another likely reason for the increase in computation speed is due to the elimination of recursiveness. Any reducible polynomial has at least 2 irreducible factors. While one is guaranteed to be of degree less than half of the degree of the input polynomial, having more than two irreducible components increases the likely-hood that different polynomials of relatively small degrees are factors of the input polynomial. Thus, implementing a breadth-first search, such as the non-recursive version, increases the chances of finding one of those polynomials quickly. However, a depth-first search, such as `isIrreducible()`, will go through all the possible divisors that contain a specific factor first, before advancing to the next divisor, so it could be forced to delve relatively deeply into the recursion before finding a divisor.

In another trial, both functions were tested for the reducibility of polynomials in the semigroup algebra $F_4[x^2, x^3]$ (Top right graph in figure 15). The results are similar to that of the first. It seems as though the non-recursive implementation is consistently faster by about a factor of 2.

4.2 The Semigroup Algebra $F_2[x^2, x^3]$

This is the same semigroup and ring used in the previous section for the examples for class usage. The benefits of it are plentiful. It is easy to understand and do computations on, as well as verify results. Additionally, the semigroup algebra preserves the same ring properties as the field of all polynomials. That is polynomial p_1, p_2, p_3 such that $p_1 p_2 = p_3$. then these polynomials $p_1^{'}, p_2^{'}, p_3^{'}$ in the semigroup algebra 2, i.e. where every coefficient is taken $mod(2)$ preserve the equality $p_1^{'} p_2^{'} = p_3^{'}$. The same holds for all other operations. It can be easily observed in the graph in figure 15 that the proportion of reducibles slowly approaches 0 as the degree increases.

The generating function of the polynomials to test reducibility on, for every degree:

```
genlist=[sum([x**degree]+[(random.choice([0,1])*(x**j))
if j in semigroup else 0 for j in [0..degree-1]])
for m in [0..9999]]
```

4.3 The Semigroup Algebra $F_2[x^4, x^5, x^6, x^7]$

A semigroup generated by $\langle m, m+1, ..., 2m-1\rangle$ has interesting properties, as it allows for the existence only of polynomials of the form $\sum_{i=m}^{\infty} a_i x^i$. The semigroup [2,3] is an example of this kind of polynomials. Another example of these polynomials is the semigroup algebra $F_2[x^4, x^5, x^6, x^7]$. As can be observed in the graph generated in section 4.1. This group shows a similar tendency for the proportion of reducible polynomials over 10,000 to decrease to 0 as the degree of the polynomials increases. However, it does so much slower than the rate of decrease for $F_2[x^2, x^3]$.

The generating code for the random polynomials:

```
genlist=[sum([x**degree]+[(random.choice([0,1])*(x**j))
if j in semigroup else 0 for j in [0..degree-1]])
for m in [0..9999]]
```

4.4 The semigroup Algebra $F_5[x^2, x^3]$

the semigroup algebra $F_5[x^2, x^3]$ is a different ring of polynomials that preserves the same property of $F_2[x^2, x^3]$ of equality to polynomials in the semigroup algebra of all integers. The semigroup algebra $F_5[x^2, x^3]$ also exhibits consistent reduction in the ratio of reducible polynomials to total polynomials generated. It appears to decrease slower than $F_2[x^2, x^3]$ but slightly faster than $F_2[x^4, x^5, x^6, x^7]$

Generating function for every degree:

```
genlist=[sum([x**degree]+[(random.choice([0,0,1,2,3,4])*(x**j))
if j in semigroup else 0 for j in [0..degree-1]])
for m in [0..9999]]
```

4.5 The semigroup Algebra $F_4[x^2, x^3]$

Unlike the previous algebras, the semigroup algebra $F_4[x^2, x^3]$ does not preserve the property of equality to polynomials in the semigroup algebra of all integers. That means that certain powers of x will be lost when multiplying. The results of reducibility, however, remain similar. Generating function for every degree:

```
genlist=[sum([x**degree]+[(random.choice([0,1,2,3])*(x**j))
if j in semigroup else 0 for j in [0..degree-1]])
for m in [0..9999]]
```

5 Future Work

The graphs provided in Section 4 provide visual evidence about the density of reducible polynomials in every degree. It can be easily observed that as the degree increases, the ratio of irreducible polynomials to the total number of polynomials is going to 0. While changes in semigroup and ring create slight variations in the number of irreducible polynomials out of $10,000$, the pattern of steady decrease remains. However, this visual evidence is obviously unsatisfactory. A rigorous mathematical proof needs to be established.

The computations performed in Section 4 are useful in that they provide a good sense of what needs to be proven. The observation that the fraction of irreducible polynomials tends to 0 allows future research to focus on proving it. In the semigroup algebra $F_2[x^2, x^3]$, this decrease is the most significant, as the rate of decrease is the fastest. Furthermore, this field is the easiest one to research, as it's properties are easily understood. Thus the following conjecture arises:

Conjecture 5.1. Let $I(d)$ be the number of polynomials in the field $F_2[x^2, x^3]$ that are irreducible of degree d. Then $\lim_{d\to\infty} \dfrac{I(d)}{2^d} = 0$.

The denominator 2^d is used since for every polynomial with degree at most d adding x^{d+1} will result in a polynomial of degree $d+1$, thus the number of polynomials of degree at most d is half of those of degree $d+1$. and the number of polynomials of degree 0 is one.

The natural continuation for future work would be to extend the proof to all polynomials of the form $\langle m, m+1, ...2m-1\rangle$. The data generated in the previous section supports the case that the property of a diminishing proportion of irreducibles with an increase in degree holds for rings generated by this sort of semigroups. In the data section, it can be observed that the decrease holds for the semigroup algebra $F_2[x^4, x^5, x^6, x^7]$. It is not far-fetched from this point to extend this conjecture to all semigroups of this form:

Conjecture 5.2. For any $m \in \mathbb{N}$ Let $I(d)$ be the number of polynomials in the semigroup algebra $F_2[x^m, x^{m+1}, ..., x^{2m-1}]$ that are irreducible of degree d. Then $\lim_{d\to\infty} \dfrac{I(d)}{2^d} = 0$.

So far the conjectures applied only to rings modulo 2. However, the data in Section 4 shows that this need not be the only ring in which the property of diminishing irreducible polynomials holds. As apparent in the data generated for polynomials in the semigroup algebra $F_4[x^2, x^3]$ as well as $F_5[x^2, x^3]$, the number of irreducibles as a fraction of total generated polynomials decreases as well. Thus, the following conjecture arises:

Conjecture 5.3. Fix $m, r \in \mathbb{N}$. Let $S = \langle m, m+1, ..., 2m-1\rangle$, and $I(d)$ be the number of polynomials in the semigroup algebra $F_r[S]$ that are irreducible of degree d. Then $\lim_{d\to\infty} \dfrac{I(d)}{r^d} = 0$.

A more advanced area for future research would be to find an exact number of irreducible polynomials of any degree. From other research it has been concluded that there exists a recurrence relation from the semigroup algebra of all polynomials of degree 2. Additional research could be done to find a similar recurrence relation for the numerical semigroup algebra defined by $F_2[x^2, x^3]$. If such relation can be found, retroactively proving that the ratio of irreducible of any degree to the number of polynomials of that degree could be trivial. However, the ratio problem could be probably solved faster without having to find the recurrence relation.

From the new recurrence relation a further question arises about finding the equation to define a recurrence relation for rings generated by an arbitrary semigroup $S = \langle m, m+1, ...2m-1\rangle$. This type of semigroup is chosen due to the similar properties to ring generated by the basic $\langle 2, 3\rangle$ semigroup for which a solution is proposed to be found first. These problems are posed with the eventual goal of forming a general solution for an arbitrary semigroup algebra.