Chapter 6

Mathematical Background 2

The last chapter gave a brief introduction to number theory. It was attempted to keep the level to a minimum required for an understanding of the cryptographic algorithms that will be encountered throughout this course. Although elements of number theory are used in algorithms such as RSA, other mathematical methods are used in other algorithms. This is because cryptographers are worried that the techniques used in one algorithm may not offer sufficient security indefinitely[1].

As a result of the fears of (say) the improved methods of factoring large numbers, cryptographers have turned to other methods such as **abstract algebra** to find security. As with number theory, the level of maths introduced will be enough to provide a basic understanding of the common cryptographic algorithms. Some topics that will be discussed are:

- Groups
- Rings
- Fields
- Polynomial arithmetic

As well as these topics we will also be looking at an area known as **complexity theory** which allows us to determine how efficient a particular algorithm is.

6.1 Abstract Algebra

Although this section is entitled "abstract algebra" we will only be looking at a very small subset of what this subject has to offer. There are three main ideas here that need to be grasped:

1. Group $\{G, \cdot\}$
2. Ring $\{R_g, +, \times\}$

[1] Again it must be stressed that generally cryptographic algorithms are not provably secure and the cryptographers paranoia comes mainly from the fact that possible attack methods have improved over the years.

3. Field $\{F, +, \times\}$

These are basically three different types of sets along with some operation(s). These sets contain elements which are not necessarily numbers. One of the things that is required for our purposes is to be allowed to combine two elements together using some operation (usually called addition and multiplication although as will be seen these aren't quite the same as what one might be used to and generally in abstract algebra we are not limited to ordinary arithmetical operations) and to obtain a result that is also in the set. In other words, we would like to work within the confines of the set. The classification of each set is determined by the axioms which it satisfies as shall be seen.

Before continuing here are some explanations of what some of the notation used represents. Don't let the notation confuse you, it is only used to explain the concepts using as little English as possible.

- $\in$: Is an element of.
- $\forall$: For all or for every.
- iff: If and only if.
- $\exists$: There exists.
- Z: This is the set of integers (i.e. whole numbers: positive, negative and zero).
- Z^+: This is the set of positive integers[2].
- Q: This is the set of rational numbers (i.e. numbers that can be expressed as quotients m/n of integers, where $n \neq 0$).
- R: This is the set of all real numbers.
- Z_n: This is set of numbers modulo n i.e. $Z_n = \{x \,|\, 0 \leq x \leq (n-1)\}$.
- $S = \{x \,|\, x \in Z\}$: This is read "S is the set of all element x such that x is an element of Z", i.e. $S = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

6.1.1 Group

A **Group** $\{G, \cdot\}$ is a set under some operation $(\cdot)$[3] if it satisfies the following 4 axioms:

1. **Closure** (A_1): For any two elements $a, b \in G,\ c = a \cdot b \in G$
2. **Associativity** (A_2): For any three elements $a, b, c \in G,\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$

[2]Remember that 0 is not considered a positive integer so $Z^+ = \{1, 2, 3, \ldots\}$
[3]This can mean addition, multiplication or some other operation.

3. **Identity** (A_3): There exists an **Identity** element $e \in G$ such that $\forall_{a \in G},\ a \cdot e = e \cdot a = a$.

4. **Inverse** (A_4): Each element in G has an inverse i.e[4]., $\forall_{a \in G}\ \exists_{a^{-1} \in G},\ a \cdot a^{-1} = a^{-1} \cdot a = e$.

If in addition the set follows the axiom:

5. **Commutativity** (A_5): For any $a, b \in G$, $a \cdot b = b \cdot a$.

it is then said to be an **Abelian group**.

Although you might be used to exponentiation as repeated multiplication, in terms of groups, exponentiation is actually repeated application of the group operator. Therefore we might have a^3 and this would equal $a \cdot a \cdot a$. So if the operation was addition then a^3 would in fact be $a + a + a$. Also we have $a^0 = e$ which for an additive group is 0; and $a^{-n} = (a^{-1})^n$. A group is said to be **cyclic** if every element of the group G is a power a^k (where k is an integer) of a fixed element $a \in G$. The element a is said to generate G or be a **generator** of G. A cyclic group is always abelian and may be finite or infinite.

If a group has a finite number of elements it is referred to as a **finite group** and the **order** of the group is equal to the number of elements in the group. Otherwise, the group is an **infinite group**.

6.1.2 Ring

A **Ring** $\{R_g, +, \times\}$ is a set with two binary operations[5] *addition* and *multiplication* that satisfies the following axioms:

1. **Abelian Group under addition** ($A_1 \rightarrow A_5$): It satisfies all of the axioms for an Abelian group (all of the above) with the operation of *addition*. The identity element is 0 and the inverse is denoted $-a$.

2. **Closure under multiplication** (M_1): For any two elements $a, b \in R_g$, $c = ab \in R_g$

3. **Associativity of multiplication** (M_2): For any elements $a, b, c \in R_g,\ (ab)c = a(bc)$

4. **Distributive** (M_3): For any elements $a, b, c \in R_g,\ a(b + c) = ab + ac$

If in addition the Ring follows the axiom:

[4]This reads, *for all* values a which are elements of G, *there exists* a value a^{-1} which is also an element of G etc.

[5]A binary operation is a mapping of two elements into one element under some operation.

5. **Commutativity** (M_4): For any $a, b \in R_g$, $ab = ba$.

it is then said to be a **commutative ring**. If in addition the commutative ring follows the axioms:

6. **Multiplicative Identity** (M_5): There is an element 1 in R such that $a1 = 1a = a$ for all a in R.
7. **No Zero Divisors** (M_6): If $a, b \in R_g$ and $ab = 0$ then *either* $a = 0$ *or* $b = 0$.

it is then said to be an **Integral domain**.

6.1.3 Field

A **Field** $\{F, +, \times\}$ is a set with two binary operations *addition* and *multiplication* that satisfies the following axioms:

1. **Integral Domain** ($A_1 - M_6$): It satisfies all of the axioms for an Integral domain (all of the above).
2. **Multiplicative Inverse** (M_7): Each element in F (except 0) has an inverse i.e., $\forall_{a \neq 0 \in F} \exists_{a^{-1} \in F},\ aa^{-1} = a^{-1}a = 1$.

In ordinary arithmetic it is possible to multiply both sides of an equation by the same value and still have the equality intact. This is not necessarily true in finite arithmetic[6]. In this particular type of arithmetic we are dealing with a set containing a finite number of values. The set of real numbers is an infinite set and is not really useful for working with on computer systems due to the limited amount of memory and processing power. It would be much easier if every operation the computer performed resulted in a finite value that was easily handled. This is where finite fields come into play. Closure is the property that causes the result of a binary operation on an ordered pair of a set to be a part of that set also. The term *ordered pair* is important as it is not generally the case that $a \cdot b = b \cdot a$.

Just to restate, Groups, Rings and Fields are all sets defined with either one or more binary operations. For example a set may be a group under one binary operation but not under another because it may obey the axioms $A_1 \rightarrow A_4$ under the first operation but not under the second. Figure 6.1 summarises the hierarchical structure of the group, ring and field. It can be seen that the group is defined under addition. Although a group is defined under operations other than addition, a ring requires that the group be defined with only addition.

[6] In fact, if you were working over the set of integers (which is infinite), this wouldn't be true either as there is no such thing as $1/n$ where n is an integer.

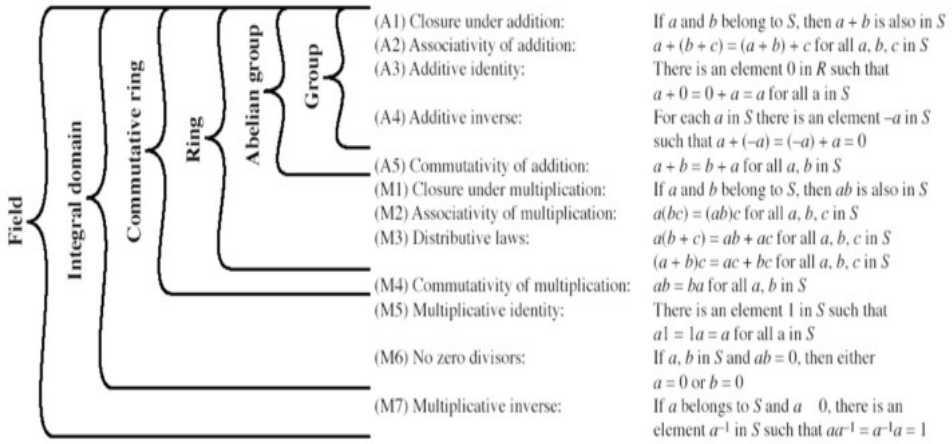

Figure 6.1: Group, Ring and Field

6.1.4 Polynomial Arithmetic

We need to understand some things about arithmetic involving polynomials before continuing. If you recall, a **polynomial** is an *expression* of the form:

$$f(x) = a_n x^n + a_{n-1} x^{x-1} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i \tag{6.1}$$

where $a_n \neq 0$, the **degree** of the polynomial is equal to the value of the integer $n \geq 0$ and the **coefficients** are the set $S = \{a_n, a_{n-1}, \ldots, a_1, a_0\}$ which is known as the **coefficient set**. If the value of $a_n = 1$ then the polynomial is said to be **monic**. If $n = 0$ then we simply have a constant known as a **constant polynomial**.

As an example we can set $n = 8$ and $S = \{1, 0, 0, 0, 1, 1, 1, 1, 1\}$ and we get the following polynomial:

$$\sum_{i=0}^{8} a_i x^i = x^8 + x^4 + x^3 + x^2 + x + 1$$

This polynomial has important significance for us as it is used in the AES standard as we shall see later.

As we are going to use these for cryptographic methods we will want to do some form of arithmetic on them. What sort of arithmetic can we do? Well there are three classes as follows:

1. Ordinary polynomial arithmetic, using the basic rules of algebra.

2. Polynomial arithmetic in which the arithmetic on the coefficients is performed modulo p; that is the coefficients are in $Z_p = 0, 1, \ldots, p-1$.

3. Polynomial arithmetic in which the coefficients are in Z_p and the polynomials are defined modulo a polynomial $m(x)$ whose highest power is some integer n.

When in school we were usually asked to solve the polynomials for certain values of x, e.g.:

$$\begin{aligned} x^2 + 3x + 2 &= 0 \\ \Rightarrow x &= -1, -2 \end{aligned}$$

However, for our purposes we are not interested in evaluating the polynomial for a certain value of x. Because of this, the variable x is sometimes known as **indeterminate**.

Considering the first class of operations above, the next question we want to ask is can we operate on polynomials using the four basic arithmetical operation of addition, subtraction, multiplication and division? Well consider two polynomials $f(x) = x^3 + x^2 + 2$ and $g(x) = x^2 - x + 1$. If we add these we get:

$$f(x) + g(x) = x^3 + 2x^2 - x + 3$$

this would seem to suggest we can add (and it turns out we can).

If we subtract them:

$$f(x) - g(x) = x^3 + x + 1$$

this would seem to suggest we can subtract (and it turns out we can).

If we multiply them:

$$f(x) \times g(x) = x^5 + 3x^2 - 2x + 2$$

this would seem to suggest we can multiply (and it turns out we can). An interesting point to note about multiplication is that the order of the product polynomial is equal to the sum of the orders of the two factors. In this case $n_{prod} = n_{f(x)} + n_{g(x)} = 3 + 2 = 5$.

What about division? Well it turns out that division requires that the set of coefficients $S = \{a_n, a_{n-1}, \ldots, a_1, a_0\}$ be a field. In other words S must satisfy the conditions described in section 6.1.3. Don't worry too much about why this is the case but just take it to be true. Remember for our purposes we really only want enough mathematics to help us understand the algorithms. Later if you are interested you can go deeper into the whole subject.

In general, division will produce a quotient and a remainder so we can write:

$$\begin{aligned} \frac{f(x)}{g(x)} &= q(x) + \frac{r(x)}{g(x)} \\ f(x) &= q(x)g(x) + r(x) \end{aligned} \tag{6.2}$$

for two polynomials $f(x)$ and $g(x)$.

6.1.5 Galois Fields

We saw earlier that a *field* obeys axioms $A_1 \rightarrow M_7$. It is possible for a set with an infinite number of elements to be a field. For example the set of real numbers is a field under the usual arithmetical operations. In cryptography however we are not interested in infinite fields because they cannot be worked with in practice (due to memory limitations etc.). What cryptographers want instead are *finite fields*. These are simply fields with a finite number of elements.

It turns out that the order of a finite field must be the power of a prime p^n where $n > 0$. The finite field of order p^n is normally written GF(p^n). The GF stands for **Galois Field** in honour of the mathematician who first studied them[7]. When $n = 1$ Galois fields take on a different structure than when $n > 1$. We will mainly be interested in GF(p) for some prime p and GF(2^n) (2 being the main prime of interest due to computers operating in binary).

Addition and multiplication in a Galois field are done modulo m(x), where m(x) is an irreducible polynomial of degree n. A polynomial m(x) over a field F is called **irreducible** if and only if m(x) cannot be expressed as a product of two polynomials both over F, and both of degree lower than that of m(x). By analogy to integers, an irreducible polynomial is also called a prime polynomial.

As an example of a Galois field we can look at GF(2^3). This has the following elements: $\{0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1\}$ and an irreducible polynomial is $x^3 + x + 1$. In binary Galois fields all polynomials are monic due to the fact that the coefficients are taken from the set $S = \{0, 1\}$.

Earlier we stated that Z_m was the set of integers modulo m. If $m = p$ where p is some prime, then we have $Z_m = Z_p = \{0, 1, 2, \ldots, p-1\}$. This, together with the arithmetic operations modulo p, form the finite field of order p. Since p is the power of a prime, this field is a Galois field. Table 6.2 shows the properties of modular arithmetic for integers in Z_m. If you compare this to figure 6.1 then it can be seen that Z_m is a commutative ring. It was seen in the last chapter that if some number $a \in Z_m$ is relatively prime to m then there exists a value a^{-1} such that $aa^{-1} \equiv 1 \pmod{m}$, i.e. the multiplicative inverse of a exists. However if $m = p$ for some prime p then each nonzero element in Z has a multiplicative inverse because each $a \in Z_p$ is relatively prime to p. Also if $ab = 0$ for $a, b \in Z_p$ then either $a = 0$ or $b = 0$. It can therefore be seen that Z_p is a finite field. This is fine for GF(p) where $n = 1$ because p is a prime but what about the case where $n > 1$? In this case p^n will not be a prime.

In the general case, if $m = p^n$ where p is again some prime and n some integer greater than 1, m will not be prime. This poses problems as Z_m will not form a finite field when the arithmetical operations are done modulo m. Remember we need this if we wish to use division as one of our operations. However, if we add two restrictions to

[7] Unfortunately Galois was killed in a dual at the age of 21.

Property	Expression
Commutative laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive laws	$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$ $[w + (x \times y)] \bmod n = [(w + x) \times (w + y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 \times w) \bmod n = w \bmod n$
Additive inverse ($-w$)	For each $w \in Z_n$, there exists a z such that $w + z \equiv 0 \bmod n$

Figure 6.2: The number of operations required to factor an integer of size n.

our arithmetical operations (which follow the ordinary rules of polynomial arithmetic using the basic rules of algebra) it is possible to keep Z_m as a finite field. These restrictions are:

1. Arithmetic on the coefficients is performed modulo p. That is, we use the rules of arithmetic for the finite field Z_p (for GF(2^n) this is basically the XOR operation).

2. If multiplication results in a polynomial of degree greater than $n - 1$, then the polynomial is reduced modulo some irreducible polynomial $m(x)$ of degree n. That is, we divide by $m(x)$ and keep the remainder. For a polynomial f(x), the remainder is expressed as $r(x) = f(x) \bmod m(x)$.

It can be shown that the set of all polynomials modulo an irreducible nth-degree polynomial $m(x)$ satisfies the axioms for $A_1 \rightarrow M_7$ above and therefore forms a finite field. This is a very important point because if we were working in GF(2^8) and we reduced the polynomial modulo 256 then this wouldn't form a finite field and we might have to think about using 251 because this is the closest prime to 256. This isn't efficient because we might be using a processor that operates on 8 bit words which allows representation of $2^8 = 256$ values. The other values ($251 \rightarrow 256$) would not be used.

Although more could be said on finite fields we will leave it there for the moment. We will be coming back to this when we study the AES algorithm in the next chapter. Stallings gives some good examples on finite fields which may help to clarify things a little more. In the next section we are going to look at something completely different to what we have been doing up to this point.

6.2 Complexity theory

All cryptographic algorithms require time and space (memory) to execute. Clearly what is desired is an algorithm that executes as quickly as possible with the minimum number of resources possible, however, often this is not feasible. **Complexity Theory** deals with the resources required during computations to solve a given problem. The most common resources (for example) would be *time* and *space*. How much time is it going to take to solve a particular problem? Generally time is a valuable resource so we would like to keep it to a minimum. Also, how much memory (space) are we going to require to solve the problem? The PC in the lab only has 128MB so if it takes more than that we are going to have to buy more. Complexity theory allows us to work out how costly different algorithms (such as DES and RSA) are going to be. In fact, if you think about it, it also allows us to analyse attacks on cryptosystems as these will be algorithmic in nature as well. Using complexity theory we can therefore determine whether a particular attack is feasible against a cryptographic algorithm and can therefore give some indication of the security of an algorithm.

Of course one could say that the time the execution of an algorithm takes depends on the speed of the particular computer on which it is running, as well as the amount of memory it has. This is true, so what we would like is a method of comparing different algorithms that is independent not only of the speed and amount of RAM a particular machine has, but also of the size of the input (which is normally denoted by n).

A measure of the efficiency of an algorithm is known as its **Time Complexity**. The time complexity of an algorithm is defined to be $f(n)$ if for all n and all inputs of length n, the algorithm takes at most $f(n)$ steps. Thus for a given processor speed and a given size of input (n) the time complexity is an *upper bound* on the execution time of the algorithm. However, the definition is not precise:

- The meaning of "step" is not precise: Is it a single processor machine instruction or a single high-level language machine instruction? Fortunately, each definition of "step" is in general related to each other by a multiplicative constant and for large n these constants are not important. What is important however is the speed at which the relative execution time is growing with increasing n. For example, In RSA, if we are concerned about whether to use a 50 digit ($n = 10^{50}$) or a 100 digit ($n = 10^{100}$) key, it is not necessary (or possible) to know exactly how long it would take to break each size of key. However in ballpark figures, we can determine the level of effort and how much extra relative effort is required for larger key sizes.

- An exact formula for $f(n)$ is generally not available but that is not important. We need only an approximation and are interested primarily in the rate of change of $f(n)$ as n increases to very large values.

There is a standard mathematical notation known as the "big O" notation which is used to characterise the time complexity of algorithms. By definition:-

$$\begin{aligned} &f(n) = O(g(n)) \text{ iff } \exists\ a, M \in Z^+ \text{ such that} \\ &|f(n)| \leq a \times |g(n)| \text{ for } n \geq M \end{aligned} \tag{6.3}$$

For example we might say that an algorithm runs in $O(n^2)$ time. This means that for an input of size n the running time of the algorithm will be proportional to n^2. So if we double the input size n the running time will increase by a factor of 4. It must be emphasised that the "big O" notation does not tell us the running time of an algorithm. It only tells us how its performance changes with the size of the input.

Anything that takes a fixed amount of resources takes $O(1)$. For example if the amount of memory required by an algorithm is always 64MB then we can say that the algorithm takes O(1) space (and similarly for time).

Lets say we want to evaluate the following polynomial:

$$p(x) = a_n x^n + a_{n-1} x^{x-1} + \ldots + a_1 x + a_0$$

Some fairly inefficient coding is shown in figure 6.3. In this implementation, each term (subexpression) is evaluated seperately. It could have been more efficient however. Each $a_i x^i$ requires $i + 1$ multiplications. Computing all n terms requires

$$\sum_{i=0}^{n} (i+1) = \frac{(n+2)(n+1)}{2}$$

multiplications. The time complexity of the algorithm B is :

$$\begin{aligned} f(n) &= \frac{(n+2)(n+1)}{2} \\ f(n) &= \frac{n^2}{2} + \frac{3}{2}n + 1 \end{aligned}$$

We can show that in this case $f(n) = O(n^2)$ (i.e. $g(n) = n^2$). We must find $M \leq n$ and a such that

$$|f(n)| \leq a \cdot |g(n)|$$

If we let $M = 4$ and $a = 1$ then equation 6.3 is satisfied as suggested by table 6.1. In this case we can say that $O(f(n)) = n^2$.

In general, the "big O" notation makes use of the term that grows fastest:

- $O(ax^7 + 3x^3 + sin(x)) = O(x^7)$
- $O(e^n + an^{10} = O(e^n))$
- $O(n! + n^{50}) = O(n!)$

```
Declare n, i, j integer;
Declare a, s array[100] real;
Declare x, p real;
Read(x, n);
for i = 0 to n do        \* Each of n terms of the f(x) *\
   {
      s[i] = 1;
      read(a[i]);
      for j = 1 to i do
         {
            s[j] = x × s[i];        \* Calculate x^i and store in s[j]*\
         }
      s[i] = a[i] × s[i];         \* Calculate a_i x^i and store*\
   }
p = 0;
for i = 0 to n do
   {
      p = p + s[i];
   }
write("Value of function at", x, "is", p ".");
```

Figure 6.3: Badly written pseudocode.

n	n^2	$f(n) = \frac{n^2}{2} + \frac{3}{2}n + 1$	$g(n) = n^2$	function ok?
3	9	10	9	No
4	16	15	16	Yes
5	25	21	25	Yes
6	36	28	36	Yes
7	49	35	49	Yes

Table 6.1: This table shows the value of the function $f(n)$ for different n.

An algorithm with an input of size n is said to be

a). Linear if the running time is $O(n)$.

b). Polynomial if the running time is $O(n^t)$ for some constant t.

c). Exponential if the running time is $O(t^{h(n)})$ where t is some constant and $h(n)$ is a polynomial in n.

Generally a problem that can be solved in polynomial time is considered feasible whereas anything worse than polynomial time is considered computationally infeasible - especially exponential time.

N.B. If n is small enough, even complex algorithms become feasible.

For example, most discussions on cryptanalysis of RSA centre on the task of factoring n into its 2 prime factors. The best known methods for doing this have a time complexity of:

$$f(n) = e^{\sqrt{ln(n) \cdot ln(ln(n))}} \tag{6.4}$$

A graph of this equation is shown in figure 6.4. The "big O" is exponential running time so this is considered computationally infeasible (with big n, i.e. $n > 150$) and hence RSA is considered secure.

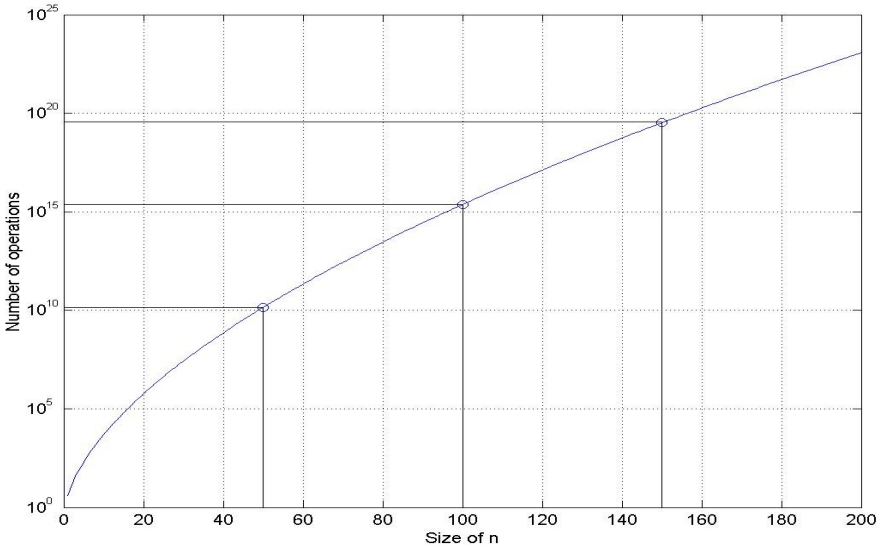

Figure 6.4: The number of operations required to factor an integer of size n.