COMPLEXITY THEORY AND THE RSA CRYPTOSYSTEM

JAYANTH GARLAPATI

Abstract. This paper will introduce topics in Complexity Theory with the goal of understanding and evaluating the RSA Cryptosystem as well as better understanding the problem of whether P = NP. I will introduce formal notions of algorithms as Deterministic and Probablistic Turing Machines, associated complexity classes of P, NP and RP, and NP-complete languages through Cook's Theorem, working towards understand One-way functions and presenting the Miller-Rabin test, an efficient algorithm for primality testing, which are both central to understanding the RSA cryptosystem. Along the way, I will discuss ways to think about P = NP.

Contents

1. Introduction

The problem of sending a message discretely between two parties has motivated people throughout the ages to develop cryptosystems from simple ciphers to much more complicated systems. Modern cryptographic systems, such as RSA, depend on one way functions, functions that are easy to compute but difficult to invert.

This sounds like an elegant solution, but what exactly does it mean for a problem to be easy or difficult? Complexity theory offers a formal solution to the problem. Complexity theory is a field dedicated to studying the computational difficulty of problems; in particular, it focuses on two questions:

(1) Is a given problem easy or diffucult
(2) Given two problems, which is harder?

2. Complexity Theory

Complexity theory is the study of the practicality of computing problems. The computational difficulty of a problem is defined by the running time of an algorithm, which is measured in terms of the basic operations it uses, such as addition, subtraction, etc.. The running time of an algorithm depends on the size of the input, and the worst case situation is always considered.

To further define the distinction between easy and difficult problems, lets consider an example. The following is an algorithm for testing the primality of a number. We will begin with a general discussion and formalize concepts later.

Algorithm 2.1. *Naive Primality testing*
Input: an integer $N \geq 2$.
Output: true if N *is prime and false otherwise.*
Algorithm:
$D \leftarrow 2$
$P \leftarrow true$
while P *is true and* $D \leq \sqrt{N}$
if D *divides* N *exactly*
then $P \leftarrow false$
else $D \leftarrow D + 1$
end while
output P

This algorithm will end at latest when $D = \sqrt{N} + 1$, so the running time is on the order of $\sqrt{N}$, which will be denoted $O(\sqrt{N})$. The input however is an integer encoded in binary, so it will have length $n = \lfloor logN \rfloor + 1$ where log is base 2. Thus, $N = 2^{n-1}$ and the running time is then $O(2^{n/2})$, which for a 1024 bit number, is 2^{512}. Theoretically this algorithm is a solution to our probleml; however, if we attempted a calculation of this magnitude with the largest computer imaginable, one made out of every atom in the earth, performing 10^{10} calculations per atom, it would not finish if it started at the earth's conception. Clearly, this is not a tenable solution practically. Many modern cryptosystems need an algorithm to check the primality of a number, but clearly one that depends on the input exponentially will not be sufficient. Most algorithms that are useful run in a polynomial amount of time with respect to the input. We will use this observation to define what it means for an algorithm to be practical: an algorithm is tractable or practical if and only if it has polynomial running time. This will be our working definition that will be refined later on.

2.1. Deterministic Turing Machines.

To formalize the notions discussed thus far we will use Turing machines as models for algorithms.

Definition 2.2. A Detereministic Turing Machine or DTM is defined by

(1) A finite alphabet Σ containing the blank symbol $*$
(2) A 2-way infinite tape divided into square, one of which is the starting square. Each square contains an element of the alphabet Σ and all but a finite number of the squares contain the blank symbol $*$
(3) A read-write head that observes a single square at a time and can move left ($\leftarrow$) or right ($\rightarrow$)
(4) A control unit along with a finite set of states Γ including a unique starting state, γ_0, and a set of halting states.

The computation of a DTM is controlled by a transition function.

$$\delta : \Gamma \times \Sigma \rightarrow \Gamma \times \Sigma \times \{\leftarrow, \rightarrow\}$$

Given the current state, γ_{cur}, and the contents of the current square, σ_{cur}, the value of $\delta(\gamma_{cur}, \sigma_{cur})$ tells the machine three things:

(1) the new state for the control unit
(2) the symbol to write in the current square
(3) whether to move the read-write head to the left of to the right by one square

We use Σ_0 to denote $\Sigma \backslash \{*\}$, the alphabet of non-blank squares. We will denote the collection of all finite strings from Σ_0 by Σ_0^*. For $x \in \Sigma_0^*$, we will denote the length of x by $|x|$. The set of strings n from Σ_0^* is denoted by Σ_0^n. The computation of a DTM on an input is the result of repeatedly applying the transition function on the input. If the machine never reaches a halting state then the compution does not complete, otherwise the computation ends when a halting state is reached. A single application of the transition function is called a step. The configuration of a DTM is a complete description of the machine at a particular point in a computation.: the contents of the tape, the position of the read-write head and the current state of the control unit. If a DTM halts on an input, then the content of the tape once the machine halts is called the output.

Let's consider an example. To define a DTM, we need to describe a set of states Γ, the alphabet Σ, and the transition function δ. We will represent the transition function by a list of quintuples. The first two entries of each quintuple represent the current state and the content of the current square, and the last three entries represent the new state, the symbol to write in the current square and the movement of the read-write head.

$$(\gamma_{cur}, \sigma_{curr}, \gamma_{new}, \sigma_{new}, \leftarrow \backslash \rightarrow)$$

The following DTM checks if a given string is a palindrome. For the sake of brevity, I will use the word same in the spot for the new state or the symbol to write if they are the same as the old state or symbol.

Example 2.3. Palindrome

The alphabet is $\Sigma = \{0, 1, *\}$ and the set of states is $\Gamma = \{\gamma_0, \gamma_1, \ldots, \gamma_5, \gamma_T, \gamma_F\}$, γ_0 is the start state, γ_T is the accept state, and γ_F is the reject state. If the machine ever encounters a state/symbol combination that it does not have a rule for then it rejects. To follow the machine, simply start at the starting state, and follow the quintuple that corresponds to what is in the current square, in the case of the first quintuple this is the starting square, and proceed to the next quintuple indicated by the new state and the contents of the square the read-write head is directed to by the previous quintuple.

$(\gamma_0, 1, \gamma_1, *, \rightarrow)$ # found 1, erase it and store as state γ_1

$(\gamma_0, 0, \gamma_2, *, \rightarrow)$ # found 0, erase it and store as state γ_2
$(\gamma_0, *, \gamma_T, *, \rightarrow)$ # empty string - accept (even length input)
$(\gamma_1/\gamma_2, 0/1, \text{same.same.} \rightarrow)$ # go right looking for the end of the string
$(\gamma_1, *, \gamma_3, *, \leftarrow)$ # end of string found, go back looking for 1
$(\gamma_2, *, \gamma_4, *, \leftarrow)$ # end of string found, go back looking for 0
$(\gamma_3, 1, \gamma_5, *, \leftarrow)$ # found matching 1, erase it and start cycle to check next digit
$(\gamma_4, 0, \gamma_5, *, \leftarrow)$ # found matching 0, erase it and start cycle to check next digit
$(\gamma_3/\gamma_4, *, \gamma_T, *, \leftarrow)$ # empty string - accept (odd length input)
$(\gamma_5, 0/1, \gamma_5, \text{same}, \leftarrow)$ # go back to the beginning of string
$(\gamma_5, *, \gamma_0, *, \rightarrow)$ # beginning found, start again

2.2. **Acceptor DTMs and Langauges.** Palindrome is a DTM that does not compute a value, but rather either accepts or rejects an input, thus it is an acceptor DTM. Any set of strings $L \subseteq \Sigma_0^*$ is called a language.

If M is an acceptor DTM, then the language accepted by M is defined as

$$L(M) = \{x \in \Sigma_0^* \mid M \text{ accepts } x\}$$

If M is an acceptor DTM, $L = L(M)$ and M halts on all inputs $x \in \Sigma_0^*$,, then we say that M decides L. There is a correspondence between languages accepted by acceptor DTMs and decision problems. For example we can associate the decision problem PALINDROME with a language

$$L_{\text{PALINDROME}} = \{x \mid x \text{ is a binary string that is a palindrome}\}$$

Definition 2.4. For a general decision problem, Π, we have the associated langauge

$$L_\Pi = \{x \in \Sigma_0^* \mid x \text{ is a natural encoding of a true instance of } \Pi\}$$

2.3. **Time Complexity, Complexity Classes and P.** The obvious way to measure the running time of a DTM is the number of steps it takes before it halts. If a DTM halts on input $x \in \Sigma_0^*$, then its running time on input x is the number of steps the mahine takes during its computation, which is denoted $t_M(x)$.

Definition 2.5. The time complexity of a DTM M that halts on every input $x \in \Sigma_0^*$, to be a function $T_{M:} : \mathbb{N} \rightarrow \mathbb{N}$ given by

$$T_M(n) = \max\{t \mid \text{there exists } x \in \Sigma_0^n \text{ such that } t_M(x) = t\}$$

In practice, we will not generally work with Turing machines since it is much more conveniant to consider higher level descriptions of algorithms, so to talk about complexity, we will accept the Church-Turing thesis, which states that any practical deterministic algorithm can be implemented as a DTM with polynomial running time.

Since the goal of complexity theory is to analyze the difficulty of problems, it useful to consider collections of problems that can all be decided by DTMs with the same bound on their time complexity. A collection of the languages corresponding to these problems is called a complexity class.

One such class is the collection of languages that can be decided in polynomial time, which by our working definition of a tractable problem, is the most immediate class of tractable langauges.

$$\begin{aligned} P \quad = \quad & \{L \subseteq \Sigma_0^* \mid \text{there is a DTM } M \text{ which decides } L \text{ and a polynomial } p(n) \\ & \text{such that } T_M(n) \leq p(n) \text{ for all } n \geq 1\} \end{aligned}$$

If Π is a decision problem for which $L_\Pi \in \mathrm{P}$, we say that there is a polynomial time algorithm for Π.

2.4. **Complexity of Functions.** In addition to considering the feasibilty of solving problems, we will also consider the feasibility of computing functions.

Definition 2.6. The class of tractable functions analgous to the class of tractable problems, P, is

$$\begin{aligned}\mathrm{FP} \quad = \quad & \{f : \Sigma_0^* \to \Sigma_0^* \mid \text{there is a DTM } M \text{ that computes } f \text{ and a polynomial} \\ & p(n) \text{ such that } T_M(n) \leq p(n) \text{ for all } n \geq 1\}\end{aligned}$$

If $f \in \mathrm{FP}$, we say that f is polynomial time computable.

Algorithm 2.7. *Euclid's Algorithm*
Input: binary integers $a \geq b \geq 1$
Output: gcd(a, b)
Algorithm:
$r_0 \leftarrow a$
$r_1 \leftarrow b$
$i \leftarrow 1$
while $r_i \neq 0$
$i \leftarrow i + 1$
$r_i \leftarrow r_{r-2} \mod r_{i-1}$
end-while
output r_{i-1}

The algorithm works by repeatedly dividing the remainder

$$\begin{aligned} a &= q_2 b + r_2 & 0 &\leq r_2 < b \\ b &= q_3 r_2 + r_3 & 0 &\leq r_3 < r_2 \\ r_2 &= q_4 r_3 + r_4 & 0 &\leq r_4 < r_3 \\ &\vdots & &\vdots \\ r_{k-3} &= q_{k-1} r_{k-2} + r_{k-1} & 0 &\leq r_{k-1} < r_{k-2} \\ r_{k-2} &= q_k r_{k-1} + r_k & r_k &= 0 \end{aligned}$$

First, we will check that the algorithm works.

Claim 2.8. $\gcd(a, b) = r_{k-1}$

Proof. $r_k = 0 \implies r_{k-1} | r_{k-2}$. After substitution, $r_{k-3} = (q_{k-1} q_k + 1) r_{k-1}$, thus $r_{k-1} | r_{k-3}$, continuing up the list of equations, we see that $r_{k-1} | r_i$ for $2 \leq i \leq k-2$, which implies that $r_{k-1} | b$ and $r_{k-1} | a$, therefore $\gcd(a, b) = r_{k-1}$. □

Now, we will consider the running time of the algorithm.

Claim 2.9. Euclid's algorithm $\in$ FP

Proof. We need to show that the number of times the while loop repeats is bounded by a polynomial that depends on the size of the input, which is $\log a + \log b$ (log base 2 since a and b are in binary). To do so, we will consider the relative sizes of r_i and r_{i+2} for $2 \leq i \leq k+2$. Since $q_{i+2} \geq 1$, $r_i = q_{i+2} r_{i+1} + r_{i+2}$, and $0 \leq r_{i+2} < r_{i+1}$ or else r_{i+2} would be too large to be a remainder, then $r_{i+2} < r_i/2$. Thus, for every two runs of the while loop, the remainder is reduced by at least $1/2$ and since the

the input is encoded in binary, the running time is then $2\lceil \log a \rceil$, and so Euclid's algorithm $\in$ FP □

In addition to Euclid's algorithm, other functions that can be computed in polynomial time are exponentiation modulo an integer c and integer multiplication. We will use these results later.

3. Non-Deterministic Polynomial time

Consider the problem of deciding if a graph can be 3-colored. Given a graph, it would be difficult to find a legal 3-coloring; we would have to use an algorithm to check every possible coloring, which would run in exponential time. However, given a graph and a correct 3-coloring of the graph, it would be easy to check if the coloring is in fact legal in polynomial time. The languages for which every input that belongs to the language has a succinct certificate that validates that fact make up of the class of languages that can be decided in Non-Deterministic Polynomial Time.

Definition 3.1. A language $L \subseteq \Sigma_0^*$ belongs to NP if there is a DTM and a polynomial $p(n)$ such that $T_M(n) \leq p(n)$ and on any input $x \in \Sigma_0^*$:

(1) if $x \in L$ then there exists a certificate $y \in \Sigma_0^*$ such that $|y| \leq p(|x|)$ and M accepts the input string $x\ y$
(2) if $x \notin L$ then for any sting $y \in \Sigma_0^*$, M rejects the input string $x\ y$

If a problem, Π,belongs to NP, then if a particular instance of the problem is decided to be true, then there is a polynomial length certificate of that fact and if not, then no such certificate exists.

Proposition 3.2. *$P \subseteq NP$*

Proof. If $L \in$ P, then there is a DTM that decided L in polynomial time given only $x \in \Sigma_0^*$ and there is no need for a certificate. In this case, the checking algorithm would simply decide if the input was in the language or not. □

The question of whether or not P = NP is a very important open problem in the field of complexity theory.

3.1. Polynomial Time Reductions.

To better understand the relative difficulty of problems, we introduce the notion of polynomial time reductions

Definition 3.3. If $A, B \subseteq \Sigma_0^*$ and $f : \Sigma_0^* \to \Sigma_0^*$ satisfiying $x \in A \iff f(x) \in B$, then f is a reduction from A to B. If in addition $f \in$ FP, then f is a polynomial time reduction from A to B. When such a function exists we say that A is polynomially reducible to B and we write $A \leq_m B$.

This notation makes sense when we note that if one problem is easy, then the problems that are polynomially reducible to it are also easy, which yields the following,

Lemma 3.4. *If $A \leq_m B$, and $B \in P$, then $A \in P$*

Proof. $A \leq_m B \implies \exists$ a DTM M that computes the function $f : \Sigma_0^* \to \Sigma_0^*$ satistfying $x \in A \iff f(x) \in B$ and a polynomial $p(n)$ such that $T_M(n) \leq p(n)$. $B \in$P $\implies \exists$ a DTM N that decides the language B and a polynomial $q(n)$ such that $T_N(n) \leq q(n)$. We can now construct a polynomial DTM to decide the langauge

A. Given $x \in \Sigma_0^n$, we can pass x to M and it will output $f(x)$, which we can input into N, and accept or reject x depending on whether N accepts or rejects $f(x)$. Thus, our new DTM decides L_A.

To finish, we must show that it does so in polynomial time. $T_M(n) \leq p(n) \implies$ computing $f(x)$ can take at most $p(n)$ amount of time as well as $p(n)$ steps $\implies$ $|f(x)| \leq p(n) + n$, since we started with an input of size n and M can only take $p(n)$ steps and the read-write head can only move one square on the tape per step. Since we pass $f(x)$ to N, this implies that $T_N(p(n) + n) \leq q(p(n) + n) \implies$ the total running time of our new machine is $T_M + T_N \leq p(n) + q(p(n) + n)$. Therefore $A \in$ P. □

By similar proof, we have the following statement,

Lemma 3.5. *If $A \leq_m B$, and $B \in NP$, then $A \in NP$*

And by applying the definition of reducibility twice, we have the following

Lemma 3.6. *$A \leq_m B$ and $B \leq_m C \implies A \leq_m C$*

Polynomial time reductions are limited in that they must convert one instance of a problem into one instance of another problem. However, there are situations where the ability to solve one problem, Π_1, will allow us to solve another problem, Π_2, by solving multiple instances of Π_1. This process defines a more general reduction known as a Turing reduction.

Definition 3.7. A function f is said to be NP-hard if there is an NP-complete langauge L such that $L \leq_T f$, where we identify L with f_L, the function corresponding to the language

$$f_{:L}; \Sigma_0^* \to \{0, 1\}, \qquad f(x) = \begin{cases} 1, & x \in L \\ 0, & x \notin L \end{cases}$$

So an NP-hard function is at least as difficult as any langauge in NP and in addition any language that is NP-complete is also NP-hard.

3.2. **NP Completeness.** We've discussed what it means for one problem to be as hard as another, so the next natural question to consider is whether there is a hardest problem, a problem that is at least as difficult as any other.

Definition 3.8. A language, L, is NP-complete if

(1) $L \in$ NP
(2) if $A \in$ NP, then $A \leq_m L$

Such languages in fact do exist and are rather important in complexity theory, since most languages in NP that are not known to be in P are NP-complete.

Proposition 3.9. *If any NP-complete langauge belongs to P, then P = NP*

Proof. We already know that P $\subseteq$ NP, so we must show that NP $\subseteq$ P. Suppose an NP-complete language, L, belongs to P. L is NP-complete $\implies$ if $A \in$ NP, then $A \leq_m L$. By Lemma 3.4, $L \in$ P $\implies$ if $A \in$ NP, then $A \in P$, so NP $\subseteq$ P and therefore P = NP. □

Proposition 3.10. *If L is NP-complete, $A \in NP$ and $L \leq_m A$, then A is also NP-complete.*

Proof. This follows directly from Lemma 3.6 □

We will now show that an NP-complete language does exist, in particular, that the problem of Boolean Satisfiability is NP-complete. A boolean function is a function $f : \{0,1\}^n \to \{0,1\}$, where 1 is interpreted as true and 0 as false. Basic boolean functions are negation (NOT), conjunction (AND), and disjunction (OR). If x is a boolean variable, then the negation of x is

$$\bar{x} = \begin{cases} 1 & \text{if } x \text{ is false} \\ 0 & \text{if } x \text{ is true} \end{cases}$$

A literal is a boolean variable or its negation. A boolean function f is in conjunctive normal form if it is written as

$$f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m} C_k$$

where each clause, C_k, is a disjuntion of literals. For example the following boolean function is in conjunctive normal form

$$f(x_1, \ldots, x_6) = (x_1 \vee \bar{x}_4) \wedge (x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_5 \vee x_6)$$

A truth assignment for a Boolean function, $f(x_1, \ldots, x_n)$, is a choice of values $x = (x_1, \ldots, x_n) \in \{0,1\}^n$. A satisfying truth assigment is an $x \in \{0,1\}^n$such that $f(x) = 1$. If such an x exists then f is said to be satisfiable. Boolean satisfiability, otherwise known as SAT, is the following decision problem.

SAT

Input:a Boolean function $f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m} C_k$, in CNF

Question: Is f satisfiable?

The following result is known as Cook's theorem and is the Fundemental Theorem of Complexity Theory.

Theorem 3.11. *SAT is NP-complete.*

Proof. A succinct certificate for SAT is a satisfing truth assignment $\implies$ SAT $\in$ NP. So we need to show that $\forall L \in$ NP, we have $L \leq_m$SAT. Let $L \in$ NP, then by the definition of NP, there is a DTM M and a polynomial $p(n)$ sich that $T_M(n) \leq p(n)$ and on any input $x \in \Sigma_0^*$:

(1) if $x \in L$ then there exists a certificate $y \in \Sigma_0^*$ such that $|y| \leq p(|x|)$ and M accepts the input string $x\ y$
(2) if $x \neq L$ then for any sting $y \in \Sigma_0^*$, M refect the input string $x\ y$

We need a polynomial reduction from L to SAT that will take any input $x \in \Sigma_0^*$ and construct an instance of SAT, S_x,that is satisfiable if and only if $x \in L$. Since $L \in$ NP, this is equivalent to saying that S_x is satisfiable if and only if, there exists a certificate $y \in \Sigma_0^*$ such that $|y| \leq p(|x|)$ and M accepts the input string $x\ y$. Thus, the satisfiability of S_x depends on whether the DTM M accepts the input string $x\ y$, so we should use the characteristics of M to define S_x.

To do so, we need to first better describe M. Let the alphabet be $\Sigma = \{\sigma_0, \ldots, \sigma_l\}$ and the set of states be $\Gamma = \{\gamma_0, \ldots, \gamma_m\}$. Let the blanks symbol $*$ be σ_0, the initial state be γ_0 , and the accept state be γ_1. The running itme of M is bounded by $p(n)$, so the tape squares that can be scanned are at most $p(n)$ squares from the starting square, since the read/write head can only move one square per step. We will label the tape squares with integer values using zero for the starting square to

reference them and note that only the contents of square $-p(n), \ldots, p(n)$ will be relevant to the computation of M.

Now consider the following variables:

$$\mathrm{sq}_{i,j,t},\ \mathrm{sc}_{i,t} \text{ and } \mathrm{st}_{k,t}$$

defined as follows:

$\mathrm{sq}_{i,j,t}-$ at time t squares i contains symbol σ_j

$\mathrm{sc}_{i,t}-$ at time t the read-write head is scanning sqaure i

$\mathrm{st}_{k,t}-$ at time t the mahcine is in state γ_k

For $x \in \Sigma_0^*$, we will construct S_x from seven clauses defined using the above variables such that it will be satisfiable only if M accepts x y. To define these clauses, we will often need to express that only one term of a collection of variables $z_1, \ldots, z_n$ is true. We will use the following notation to express that fact in CNF

$$\mathrm{Unique}(z_1, \ldots, z_n) = \left(\bigvee_{i=1}^{n} z_i\right) \wedge \left(\bigwedge_{1 \le i < j \le n} (\overline{z}_i \vee \overline{z}_j)\right)$$

Suppose $\mathrm{Unique}(z_1, \ldots, z_n)$ is true, then both sides of the conjunction are true. $(\bigvee_{i=1}^{n} z_i)$ is true $\implies$ at least one term must be true. Suppose more than one term, z_k and z_l, are true, then $(\overline{z}_k \vee \overline{z}_l)$ will be false which implies that $\left(\bigwedge_{1 \le i < j \le n} (\overline{z}_i \vee \overline{z}_j)\right)$ will be false, which is a contradiction. Thus one term of $z_1, \ldots, z_n$ is true. Conversely, suppose exactly one term, z_p, from the collection $z_1, \ldots, z_n \implies (\bigvee_{i=1}^{n} z_i)$ and $\bigwedge_{\{1,\ldots,n\} \setminus \{p\}} \overline{z}_i$ are true. Thus, $\mathrm{Unique}(z_1, \ldots, z_n)$ is true. Therefore $\mathrm{Unique}(z_1, \ldots, z_n)$ is true if and only if exactly one term of $z_1, \ldots, z_n$ is true.

The seven clauses of S_x will ensure that the computation of M will occur according to the definition of a DTM.

(i) The read/write head can only scan one square at any time.

$$C_1 = \bigwedge_{t=0}^{p(n)} \mathrm{Unique}(\mathrm{sc}_{-p(n),t}, \ldots,\ \mathrm{sc}_{p(n),t})$$

If $\mathrm{Unique}(\mathrm{sc}_{-p(n),t}, \ldots,\ \mathrm{sc}_{p(n),t})$ is true, then at time t the read-write head is scanning exactly one square of the collection $-p(n), \ldots, p(n)$, the relevant range of sqaures we defined earlier. Since C_1 is a conjunction of such Unique statements for $0 \le t \le p(n)$, C_1 is true if and only if at any time the read/write head is scanning only one square.

(ii) Each square contains only one symbol.

$$C_2 = \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{t=0}^{p(n)} \mathrm{Unique}(\mathrm{sq}_{i,0,t}, \ldots, \mathrm{sq}_{i,l,t})$$

Similarly, C_2 is true if and only if at any time, every square contains exactly one symbol.

(iii) The machine is always in a single state

$$C_1 = \bigwedge_{t=0}^{p(n)} \mathrm{Unique}(\mathrm{st}_{0,t}, \ldots, \mathrm{st}_{m,t})$$

Similarly, C_3 is true if and only if at any time t the machine M is in a single state.

(iv) The computation starts correctly

The following conditions define the starting condition of the DTM M. At time $t = 0$ the squares $-p(n), \ldots, -1$ are blank, the squares $0, 1, \ldots, n$ contain the string $x = \sigma_{j_0}\sigma_{j_1}\ldots\sigma_{j_n}$ and the square $n+1$ to $p(n)$ can contain anything (since any string in these squares could be a possible certificate for various DTMs). Also, the starting position of the read/write head must be at the zero square and the initial state is γ_0 :

$$C_4 = \mathrm{sc}_{0,0} \wedge \mathrm{st}_{0,0} \wedge \bigwedge_{i=0}^{n} \mathrm{sq}_{i,j_i,0} \wedge \bigwedge_{i=-p(n)}^{-1} \mathrm{sq}_{i,0,0}$$

(v) The computation ends in acceptance.

At some time $t \leq p(n)$, M enters the accept state γ_1:

$$C_5 = \bigvee_{t=0}^{p(n)} \mathrm{st}_{1,t}$$

(vi) Only the symbol in the current square can change.

Only the symbol in the current sqaure can change from time t to $t+1$

$$C_6 = \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{j=0}^{l} \bigwedge_{t=0}^{p(n)} (\mathrm{sc}_{i,,t} \vee \mathrm{sq}_{i,j,t} \vee \overline{\mathrm{sq}}_{i,j,t+1}) \wedge (\mathrm{sc}_{i,t} \vee \overline{\mathrm{sq}}_{i,j,t} \vee \mathrm{sq}_{i,j,t+1})$$

(vii) The transition function determines the computation of M

If at time t the machine is in state γ_k, the read/write head is scanning square i, and this sqaure contains symbol σ_j then

$$\delta(\gamma_k, \sigma_j) = (\gamma_p, \sigma_q, b)$$

describes the new state (γ_p), the new symbol to write in square i (σ_q), and whether the read-write head moves right or left (We let $b = 1$ if it moves right and $b = -1$ if it moves left)

$$C_7 = \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{j=0}^{l} \bigwedge_{t=0}^{p(n)} \bigwedge_{k=0}^{m} (\overline{\mathrm{st}}_{k,t} \vee \overline{\mathrm{sc}}_{i,t} \vee \overline{\mathrm{sq}}_{i,j,t}) \vee (\mathrm{st}_{p,t+1} \wedge \mathrm{sq}_{i,q,t+1} \wedge \mathrm{sc}_{i+b,t+1})$$

At t if the present state is γ_k, the scanning head is on square i and the symbol in the square is σ_j, then $(\overline{\mathrm{st}}_{k,t} \vee \overline{\mathrm{sc}}_{i,t} \vee \overline{\mathrm{sq}}_{i,j,t})$ is false and thus $(\mathrm{st}_{p,t+1} \wedge \mathrm{sq}_{i,q,t+1} \wedge \mathrm{sc}_{i+b,t+1})$ must be true, which implies that the new state is γ_p, the new symbol in square i is σ_p and the scanning head has moved right or left accordingly, essentially given inputs the transition function will correctly compute the next step in the algorithm.

For simplicity, C_7 is not written in CNF, but can be rewritten in CNF:

$$C_7 = \bigwedge_{i=-p(n)}^{p(n)} \bigwedge_{j=0}^{l} \bigwedge_{t=0}^{p(n)} \bigwedge_{k=0}^{m} (\overline{\mathrm{st}}_{k,t} \vee \mathrm{st}_{p,t+1}) \wedge (\overline{\mathrm{sc}}_{i,t} \vee \mathrm{sc}_{i+b,t+1}) \wedge \overline{(\mathrm{sq}}_{i,j,t} \vee \mathrm{sq}_{i,q,t+1})$$

If we take the conjunction of the above seven clauses to be S_x, we will have an instance of SAT. We need to show that S_x is polynomial in input size. The number is distinct variables is $(2l+2)p(n)^2 + (l+m+1)p(n)$ or $O(p(n)^2)$, which

follows from counting the permutations of the variables $\mathrm{sq}_{i,j,t}$, $\mathrm{sc}_{i,t}$, $\mathrm{st}_{k,t}$. Also, the total number of clauses is $O(p(n)^3)$, which is determined by C_1. Thus S_x is polynomial in input size. In addition, S_x is satisfiable if and only if the DTM M accepts the input $x\ y$ for a certificate $y \in \Sigma_0^*$ in time bounded above by $p(n)$. In fact, given a satifying truth assignment, we can find the certificate for x by looking at $\mathrm{sq}_{n+1,j_{n+1},0}, \ldots, \mathrm{sq}_{p(n),j_{pn)},0}$ which are the contents of the squares $n+1$ to $p(n)$ at time $t = 0$. Therefore for any problem $L \in$ NP, $L \leq_m$ SAT, which implies that SAT is NP-complete. □

Cook's result is very important in complexity theory because it provides a natural example of an NP-complete language, and using Proposition 3.10, which states that if L is NP-complete, $A \in$ NP and $L \leq_m A$, then A is also NP-complete, many thousands of languages of problems in diverse areas have been shown to be NP-complete.

4. Relations Between Compexity Classes, Approaching P = NP

Now that we have introduced several complexity classes, we will explore the relations between them.

4.1. Complements of Langauges.

To understand the relations between classes, we must develop the notion of a complement of a language.

Definition 4.1. If $L \subseteq \Sigma_0^*$ is a language then the complement of L is

$$L^c = \{x \in \Sigma_0^* \mid x \notin L\}.$$

If C is a complexity class, then the class of complements of a languages in C is

$$\text{co-C} = \{L \subseteq \Sigma_0^* \mid L^c \in \text{C}\}.$$

If a language is in P, then we have a DTM that decides L in polynomial time, telling us whether a given input is in the language or not in polynomial time, so by simply reversing the output of the algorithm, we have a polynomial time DTM that decides L^c. Thus, P = co-P. However,The same is not true for NP. From Definition 4.1, A language $L \subseteq \Sigma_0^*$ belongs to co-NP if there is a DTM M and a polynomial $p(n)$ sich that $T_M(n) \leq p(n)$ and on any input $x \in \Sigma_0^*$:

(1) if $x \notin L$ then there exists a certificate $y \in \Sigma_0^*$ such that $|y| \leq p(|x|)$ and M accepts the input string $x\ y$
(2) if $x \in L$ then for any sting $y \in \Sigma_0^*$, M rejects the input string $x\ y$

Thus to verify that a language is in NP, we simply must find a certificate that M will accept, but to to check that a language is in co-NP, we must check every possible certificate to ensure each one is rejected, so we can't use the DTM given by the definition of NP for L and produce a new DTM to check if $L^c \in$ co-NP. The question of NP = co-NP is an open problem in complexity, probably second in importance to the question of whether P = NP. The following proposition provides one avenue from which to attack the problem.

Proposition 4.2. *If L is NP-complete and L belongs to co-NP then NP = co-NP*

We know that P = co-P and P $\subseteq$ NP, so we also have P $\subseteq$ NP $\cap$ co-NP. Another important open problem is whether not P = NP $\cap$ co-NP.

4.2. **Containments between Complexity Classes.** Thus far we know of the following containments between complexity classes P $\subseteq$ NP $\cap$ co-NP $\subseteq$ NP. The question of whether or not these containments are strict are important in complexity theory, in particular the question of whether P = NP is central to the field. There are some results that give us a way to attack the problem. One approach is the p-isomorphism conjecture, which if shown to be true, would imply that P $\neq$ NP.

Definition 4.3. Two languages over possibly different alphabets, $A \subseteq \Sigma_0^*$ and $B \subseteq \Pi_0^*$, are p-isomorphic if there exists a function f such that:

(1) f is a bijection between Σ_0^* and Π_0^*
(2) $x \in A \iff f(x) \in B$
(3) both f and f^{-1} belong to FP

Conjecture 4.4. *All NP-complete languages are p-isomorphic.*

Theorem 4.5. *If p-isomorphism conjecture is true then P $\neq$ NP*

Proof. Suppose P = NP, then all languages in P are NP-complete. However, P contains finite languages which cannot be p-isomorphic with infinite languages. Thus, P $\neq$ NP.

The task of proving this central result then is reduced to proving the p-isomorphism conjecture. □

5. Probabilistic Computation

Many problems that would be difficult to solve using a deterministic Turing machine or a traditional algorithm may be easier to solve if we apply an algorithm that instead of outputing a certain answer, either outputs true or 'probably false' with probablity of at least 1/2. This type of algorithm is known as a probabilistic algorithm. Before formalizing this concept, we will discuss the general idea further with an example.

5.1. Probabilistic Algorithms.

Example 5.1. Let $\mathbb{Z}[x_1, \ldots, x_n]$ denote the set of polynomial in n variables with integer coefficients. Can we efficiently decide whether two such polynomials f and g are identical. If f and g are identical, then $f - g = 0$, so efficiently deciding whether a function is indentically zero will suffice to decide the problem. Consider the following algorithm. We use $a_1, \ldots, a_n \in_R A$ to denote that $a_1, \ldots, a_n$ are chosen randomly and independently from the set A

Algorithm 5.2. *Probablistic algorithm to decide if a polynomial is not identically zero*

Input: an integer polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ of degree k
choose $a_1, \ldots, a_n \in [1, 2, \ldots, 2kn]$
if $f(a_1, \ldots, a_n) \neq 0$
then output true
else output false

If the algorithm ever outputs true, then we know with certainty that the function is not identically zero. If f is not identically zero, then the algorithm will output false only if one of the a_k is a root of the polynomial, which seems to be fairly unlikely. Suppose then we repeated the algorithm a hundred times, randomly and

independently choosing $a_1, \ldots a_n$ each time, it would be very unlikely that we pick a root each of the hundred runs. Thus, if the algorithm did output false all 100 iterations of the algorithm, then we can say with a high degree of certainty that the function is identically zero. This process allows us to avoid checking every certificate; instead we randomly pick one and chcek if it works. The reason this works is that if the polynomial is not identically zero, then there are lots of good certificates, choices of $a_1, \ldots, a_n$, of that fact and the chances a random certicficate will work are very good; however if the polynomial is identically zero then the algorithm will always correctly output false. The following theorem formalizes this intuition. If the probability of an event occuring is at most x/y, we denote this $\Pr[Event] \leq \frac{x}{y}$

Theorem 5.3. *Suppose $f \in \mathbb{Z}[x_1, \ldots, x_n]$ has degree at most k and is not identically zero. If $a_1, \ldots, a_n \in_R [1, \ldots, N]$, then $Pr[f(a_1, \ldots a_n) = 0] \leq \frac{k}{N}$*

Since the algorithm randomly chooses from a set of $2kn$, applying this theorem, we have $\Pr[f(a_1, \ldots a_n) = 0] \leq \frac{1}{2}$, so if the input polynomial is not identically zero, then the probablity of the algorithm correctly outputting true is at least 1/2, but if the algorithm is identically zero, then the output will always be false. At first glance, having only 1/2 chance of being correct if the algorithm is not identically zero doesn't seem great, but if we rewrite the algorithm to repeat itself 100 times, we have that the probablity of incorrectly deciding that a polynomial is identically zero is less than $1/2^{100}$, which is practical terms is excellent.

Now let's consider the efficiency of this procedure. Since the polynomial can be evaluated at $a_1, \ldots, a_n$ in polynomial time, the algorithm is efficient in that sense. However, when we consider probabilistic algorithms, we also need randomness which must also factor into efficiency. The resource of randomness is measured in the number of random bits used during the computation. We assume that we have a source for random bits, such as a sequence of coin-flips, but often algorithms require more than just random bits, such as the one we introduced in the example which require random integers from an interval. The following is one way to produce random integers from random bits.

Algorithm 5.4. *Choosing an integer $a \in_R \{0, \ldots n\}$ using random bits*

Supposing we have an infinite sequence of independent random bits, we can use the following procedure to choose a random integer $a \in_R \{0, \ldots, n\}$. Suppose that $2^{k-1} \leq n < 2^k$.

read k random bits, $b_1, \ldots, b_k$, from our infinite sequence.
If $a = b_1 \ldots b_k \in \{0, \ldots n\}$, where a is encoded in binary
then output a
else repeat

After a single interation the probablity that a number from the interval is chosen is

$$\Pr[a \in \{0, \ldots, n\}] = \frac{n+1}{2^k} > \frac{1}{2}$$

So it is expected that an output will occur after fewer than two iterations and with probablity $1 - 1/2^{100}$, an output will occur after 100 iterations. We can be confident that we will get an output in a reasonable amount of time, but we also need to ensure that the outputted integer will have been chosen at random from

the interval $\{0, \ldots, n\}$. If $m \in \{0, \ldots, n\}$ and we let a_j denote the value of a chose in the jth iteration of this procedure then

$$\begin{aligned}\Pr[\text{Output is } m] &= \sum_{j=1}^{\infty} \Pr[a_j = m \text{ and } a_1, \ldots a_{j-1} \geq n+1] \\ &= \frac{1}{2^k} \sum_{j=0}^{\infty} \left(1 - \frac{n+1}{2^k}\right)^j \\ &= \frac{1}{n+1}\end{aligned}$$

Thus, the integer that is chosen using this method is chosen randomly from the interval.

5.2. **Probabilistic Turing Machines.** Earlier we stated that a problem is tractable or practical if and only if there is a polynomial time algorithm for its solution, but in fact, this algorithm can be either deterministic or probabilistic. We will now fomalize the notion of a probabilistic algorithm as a probabalistic Turing machine or PTM and adjust our definition of a tractable problem.

Definition 5.5. A Probabilistic Turing Machine or PTM is a DTM with an extra tape, the coin-tossing tape, which contains an infinite sequence of uniformly distributed independent random bits. This tape has a read-only head called the coin-tossing head. The machine performs computations like a DTM except the coin-tossing tape can read a bit from the coin-tossing tape in a single step, so the running time of the algorithm will take into account the amount of randomness the algorithm requires.

The transition function of a PTM depends on not only the current state and the symbol in the current square, but also on the random bit in the square currently scanned by the coin-tossing head. The transition function now outputs the new state, the new symbol to write in the current square, and the movements of both the read/write head and the coin-tossing head.

Since the computation of a PTM depends on the random bits used, the running time and whether or not the machine halt on an input are both random variables, so we say that a PTM is halting if it halts after finitely many steps on every input $x \in \Sigma_0^*$ regardless of the random bits used in its computation. the time complexity of a halting PTM, M, is $T_M : \mathbb{N} \to \mathbb{N}$ defined by

$$T_M(n) = \max\{t | \exists x \in \Sigma_0^n \text{ such that } \Pr[t_M(x) = t] > 0\}$$

A PTM has polynomial running time if there is a polynomial, $p(n)$, such that $T_M(n) \leq p(n)$, for every $n \in \mathbb{N}$. If a PTM is not halting we can still define its expected running time

$$ET_M(n) = \max\{t | \exists x \in \Sigma_0^n \text{ such that } \mathrm{E}[t_M(x)] = t\}$$

We can now define the complexity class of languages that can be decided in randomized polynomial time or RP.

Definition 5.6. A language $L \in \mathrm{RP}$ if and only if there is a polynomial time PTM, M, such that on any input $x \in \Sigma_0^n$:

(1) if $x \in L$ then $\Pr[M \text{ accepts } x] \geq 1/2$
(2) if $x \notin L$ then $\Pr[M \text{ accepts } x] = 0$

Our example of a probabilistic algorithm that determine whether a given polynomial is not identically zero could be implemented as a PTM in this class. If a zero polynomial is inputed into the algorithm, the polynomial always rejects, so condition (2) is satisfied and we showed earlier that the probability of incorrectly rejecting a non-zero polynomial is less than $1/2$, so condition (1) is also fulfilled.

Another way of thinking of RP is the collection of languages L such that a random input will be a succicnt certificate for the problem at least $1/2$ of the time, but if $x \notin L$, then no such certificate exists and all inputs will be rejected. This leads to the following ontainments.

If a language belongs to RP, then we can reduce the probability of error by repeating the computation a polynomial number of times as we did with our example algorithm.

Proposition 5.7. *If $L \in RP$ and $p(n) \geq 1$ is a polynomial then there exists a polynomial time PTM, M, such that on input $x \in \Sigma_0^n$:*

(1) if $x \in L$ then $\Pr[M \text{ accepts } x] \geq 1 - 2^{-p(n)}$
(2) if $x \notin L$ then $\Pr[M \text{ accepts } x] = 0$

Proof. $L \in \text{RP} \implies$ there is a polynomial time PTM, N, such that on any input $x \in \Sigma_0^n$:

(1) if $x \in L$ then $\Pr[N \text{ accepts } x] \geq 1/2$
(2) if $x \notin L$ then $\Pr[N \text{ accepts } x] = 0$

We can construct a PTM, M, by repeating the computation of N $p(n)$ times. This new machine accepts the input if and only if the input is accepted on one of the iterations. Since if $x \notin L$ then $\Pr[N \text{ accepts } x] = 0$, then if $x \notin L$ then $\Pr[M \text{ accepts } x] = 0$, satisfying (2) of the proposition. Also, N incorrectly rejects $x \in L$ with probability less than $1/2$, so then after $p(n)$ iterations, the probability of M incorrectly rejecting a good certificate is less than $2^{-p(n)}$, thus if $x \in L$ then $\Pr[M \text{ accepts } x] \geq 1 - 2^{-p(n)}$. The new machine M also runs in polynomial time. Suppose that the running time of N was bounded by a polynomial $O(x^k)$, then if M repeats the computation $p(n)$ times, the running time of M is bounded by a polynomial $O(p(n)x^k)$, thus M is a polynomial time PTM □

Now that we have developed the notion of probabilistic algorithims, we can return to the question of primality testing introduced earlier.

5.3. **Primality Testing.** Many crytographic systems require large random prime numbers. We have already discussed how to choose a random k bit integer given a source of randomness. Now we need to be able to check whether the large number chosen is prime. Earlier we described the naive primality algorithm, which was an example of an intractable algorithm, but in 2002, Agrawal, Kagal and Saxena discovered the striking result that there exists a deterministic polynomial time algorithm for primality testing, that PRIME $\in$ P! However, this algorithm has running time $O(\log^6 n)$ which is still not very efficient. Fortunately, there is an efficient probablistic algorithm for primality testing, the Miller-Rabin algorithm. To discuss the Miller-Rabin algorithm, we need some number and group theory.

Definition 5.8. A group is a set G together with a binary operation $\cdot$ denoted $(G, \cdot)$, satisfying the following conditions

(1) if $g, h \in G$, then $g \cdot h \in G$ (Closure)

(2) if $g, h, k \in G$ then $g \cdot (h \cdot k) = (g \cdot h) \cdot k$ (Associativity)
(3) there exists $\mathrm{I}_G \in G$ such that $\forall g \in G, g \cdot \mathrm{I}_G = g = \mathrm{I}_G \cdot g$ (Identity Element)
(4) $\forall g \in G$, there exists $g^{-1} \in G$ such that $g \cdot g^{-1} = \mathrm{I}_G = g^{-1} \cdot g$ (Inverse Element)

Definition 5.9. If $(G, \cdot)$ is a group, then $H \subseteq G$ is a subgroup of G iff it forms a group under the binary operation of G, more specifically iff it satisfies the following conditions

(1) if $g, h \in H$ then $g \cdot h \in H$
(2) $\mathrm{I}_G \in H$
(3) if $h \in H$, then $h^{-1} \in H$

Theorem 5.10. *Lagrange's Theorem. If H is a subgroup of G then $|H|$ divides $|G|$ exactly*

Corollary 5.11. *If H is a proper subgroup of a group G then $|H| \leq |G|/2$*

Theorem 5.12. *Prime Number Theorem. $\pi(N)$ is the number of prime numbers $p \leq N$*

$$\lim_{N \to \infty} \frac{\pi(N) \ln N}{N} = 1$$

A weaker statement of the theorem provides upper and lower bounds for $\pi(N)$

$$O\left(\frac{N}{\log N}\right) \leq \pi(N) \leq O\left(\frac{N}{\log N}\right)$$

We denote the fact that d divides n by $d \mid n$. If $d \mid n$ and $d \neq n$, then d is a proper divisor of n, and if $d \mid n$ and $d \neq 1$, then d is a non-trivial divisor of n,

Two integers, a and b, are congruent modulo an integer n iff $n \mid a - b$. We denote this by $a = b \bmod n$

If N is a positive integer then the set of residues mod n is

$$\mathbb{Z}_n = \{a \mid 0 \leq a \leq n - 1\}$$

This is a group under addition mod n with identity 0. The set of non-zero residues mod n is

$$\mathbb{Z}_n^+ = \{a \mid 1 \leq a \leq n - 1\}$$

Theorem 5.13. *Chinese Remainder Theorem. If $n_1, n_2 \ldots, n_k$ are pairwise co-prime (that is $\gcd(n_i, n_j) = 1$ for $i \neq j$), and $N = \Pi_{i=1}^k n_i$, then the following system of congruences has a uniqie solution* $\bmod N$

$$\begin{aligned} x &= a_1 \bmod n_1 \\ x &= a_2 \bmod n_2 \\ &\vdots \\ x &= a_k \bmod n_k \end{aligned}$$

Writing $N_i = N/n_i$ and using Euclid's algorithm to find $b_1, \ldots, b_k$ such that $b_i N_i = 1 \mod n_i$, the solution is

$$x = \sum_{i-1}^{k} b_i N_i a_i \bmod N$$

The set of units mod$n$ is

$$\mathbb{Z}_n^* = \{a \mid 1 \leq a \leq n-1 \text{ and } \gcd(a,n) = 1\}$$

This is a group under multiplication mod$n$ with identity 1

If p is prime, then $\mathbb{Z}_p$ is a field.

Theorem 5.14. *Fermat's Little Theorem. If p is prime and $a \in \mathbb{Z}_p^*$ then*

$$a^{p-1} = 1 \bmod p$$

Now we can approach the problem of primality testing. Consider the complementary problem to PRIME

COMPOSITE

Input: integer n

Question: Is n composite?

We will show that COMPOSITE $\in$ RP using the Miller-Rabin algorithm, but the proof will require the following Lemma

Lemma 5.15. *Let $n \geq 3$ be odd and $a \in \mathbb{Z}_n^+$. Write $n-1 = 2^k m$, where m is odd. If either of the following conditions hold, then n is composite*

(1) $a^{n-1} \neq 1 \bmod n$
(2) $a^{n-1} = 1 \bmod n, a^m \neq 1 \bmod n$ *and none of the values in the sequence* $a^k, a^{2k}, a^{4k}, \ldots, a^{2^k m}$ *are congruent to* $-1 \bmod n$.

Proof. By Fermat's Little Theorem, if p is prime, then $\forall a \in \mathbb{Z}_p^+$, $a^{p-1} = 1 \bmod p$, so (1) holds if n is composite. If (2) holds then let b be the last integer in the sequence $a^m, a^{2m}, \ldots$that is not congruent to $1 \bmod n$. Then $b^2 = 1 \bmod n$, but $b \neq \pm 1 \bmod n$, so n divides $b^2 - 1$, but not $b+1$ or $b-1$, thus $b+1$ and $b-1$ must be non-trivial factors of n, so n is composite. □

If $a \in \mathbb{Z}_n^+$, satisfies condition (1), then it is a Fermat witness to the compositeness of n, and if it satisfies (2), then it is a Miller witness. Let the set of Fermat witnesses for an integer n be

$$F_n = \{a \in \mathbb{Z}_n^+ \mid a \text{ is a Fermat witness for } n\}$$

Suppose $a \in \mathbb{Z}_n^+$ is not a Fermat witness for n, then $a^{n-1} = 1 \bmod n \implies n \mid (a^{n-1} - 1) \implies \exists k \in \mathbb{Z}$ such that $a^{n-1} - kn = 1$. Since $\gcd(a,n) \mid a^{n-1} - kn$, then $\gcd(a,n) = 1$. Thus, every $a \in \mathbb{Z}_n^+$ that is not coprime with n, is a Fermat witness for n.

Definition 5.16. A composite integer is a Carmicheal number if the only Fermat witnesses for n are $a \in \mathbb{Z}_n^+$ that are not coprime with n. The smallest such number is $561 = 3 \cdot 11 \cdot 17$.

For Carmicheal numbers there are relatively few Fermat witnesses, but by the following proposition if n is not a Carmicheal number then there are quite a few.

Proposition 5.17. *If n is composite but not a Carmicheal number then $|F_n| > \frac{n}{2}$*

Proof. As we noted earlier, the set of integers less than and coprime to an integer form a multiplicative group $\mathbb{Z}_n^* = \{1 \leq a < n \mid \gcd(a,n) = 1\}$. Denote the elements of this group that are not Fermat witnesses $B = \mathbb{Z}_n^* \backslash F_n$, so

$$B = \{a \in \mathbb{Z}_n^* \mid a^{n-1} = 1 \bmod n\}$$

B is a subgroup of $\mathbb{Z}_n^*$ since

(1) if $a, b \in B$ then $(ab)^{n-1} = a^{n-1}b^{n-1} = 1 \cdot 1 = 1 = 1 \bmod n$, so $ab \in B$
(2) $1^{n-1} = 1 = 1 \bmod n$, so $1 \in B$
(3) If $a \in B$, then $(a^{-1})^{n-1} = (a^{n-1})^{-1} = 1^{-1} = 1 \bmod n$, so $a^{-1} \in B$

Thus B is a subgroup of $\mathbb{Z}_n^*$. Since n is a composite number that is not a Carmicheal number, there exists a Fermat witness that is coprime with $n \implies \exists b \in \mathbb{Z}_n^* \backslash B$, so B is a proper subgroup of $\mathbb{Z}_n^*$. Hence, by the corollary to Langrange's theorem, $|B| \leq (n-1)/2 \implies n - |B| \geq 1/2$ and since $|\mathbb{Z}_n^+| = n - 1$,

$$|F_n| = |\mathbb{Z}_n^+| - |B| > \frac{n}{2}$$

□

Algorithm 5.18. *Miller-Rabin Primality Test*
Input: an odd integer $n \geq 3$
choose $a \in_R \mathbb{Z}_n^+$
if $\gcd(a, n) \neq 1$*, then output 'composite'*
let $n - 1 = 2^k m$*, where* m *is odd*
if $a^m = 1 \bmod n$ *then output 'prime'*
for $i = 0$ *to* $k - 1$
  if $a^{m \cdot 2^i} = -1 \bmod n$ *then output 'prime'*
next i
output 'composite'

Theorem 5.19. *The Miller-Rabin Primality test is a probabilistic polynomial time algorithm. Given input* n

(1) *if* n *is prime then the algorithm always outputs 'prime'*
(2) *if* n *is composite then*

$$Pr[\text{the algorithm outputs 'composite'}] \geq \frac{1}{2}$$

Hence COMPOSITE $\in$ *RP or equivalently PRIME* $\in$ *co-RP*

Proof. The Miller-Rabin test is clearly a probabilistic polynomial time algorithm since it only uses basic operations can be computed in polynomial time such as multiplication, calculation of the greatest common divisor and exponentiation mod n, so we must show that the two conditions hold to complete the proof.

Suppose n is prime, then $\forall a \in \mathbb{Z}_n^+$, $\gcd(a, n) = 1$, by definition of primality, so the algorithm cannot output 'composite' at line 2. The only other way the algorithm could output 'composite' is if $a^m \neq 1 \bmod n$ and none of the values in the sequence $a^m, a^{2m}, a^{4m}, \ldots, a^{2km}$ are congruent to $-1 \bmod n$. In which case, either $a^{n-1} \neq 1 \bmod n$ and a is a Fermat witness or $a^{n-1} = 1 \bmod n$ and a is a Miller witness, but by Lemma 5.15 this is impossible since n is prime. Thus (1) holds.

Now it is left to prove (2). To do so, we must consider two cases.

Case 1. The input n is composite, but not a Carmicheal number.
Suppose the algorithm incorrectly outputs 'prime', then either $a^m = 1 \bmod n$ or $a^{m \cdot 2^i} = -1 \bmod n$ for some $0 \leq i \leq k - 1$. If $a^m = 1 \bmod n$, then $a^{2^k m} = 1 \bmod n \implies a^{n-1} = 1 \bmod n$. If $a^{m \cdot 2^i} = -1 \bmod n$ for some $0 \leq i \leq k - 1$, then $n \mid a^{m \cdot 2^i} + 1$. $(a^{m \cdot 2^{i+1}} - 1) = (a^{m \cdot 2^i} + 1)(a^{m \cdot 2^i} - 1)$, so since $n \mid a^{m \cdot 2^i} + 1$, then $n \mid (a^{m \cdot 2^{i+1}} - 1) \implies a^{m \cdot 2^{i+1}} = 1 \bmod n \implies a^{2^k m} = 1 \bmod n \implies a^{n-1} = 1 \bmod n$. In both cases, $a^{n-1} = 1 \bmod n$, so a is not a Fermat witness, and by Proposition

5.17, we have that if n is composite and not a Carmicheal number, then $|F_n| > \frac{n}{2}$. Since $|\mathbb{Z}_n^+| = n-1$, then $|F_n| > |\mathbb{Z}_n^+|/2$, so since the algorithm chooses $a \in_R \mathbb{Z}_n^+$,

$$\Pr[\text{algorithm outputs 'composite'}] \geq \frac{1}{2}$$

Case 2. The input n is a Carmicheal number.
There are two subcases depending on whether or not n is a prime power. (n is a prime power if $n = p^k$ where p is prime and $k \geq 1$).

We will first examine the case where n is not a prime power.
Let

$$t = \max\{0 \leq i \leq n-1 \mid \exists a \in \mathbb{Z}_n^* \text{ such that } a^{m \cdot 2^i} = -1 \bmod n\}$$

and

$$B_t = \{a \in \mathbb{Z}_n^* \mid a^{m \cdot 2^t} = \pm 1 \bmod n\}$$

Note that if $a \in B_t$, then the algorithm will output 'composite'. Since if the algorithm outputs prime then either $a^m = 1 \bmod n$ or by the definition of t, there exists $0 \leq i \leq t$ such that $a^{m \cdot 2^i} = -1 \bmod n$. In either situation, by similar argument as in Case 1, this implies that $a^{m \cdot 2^t} = \pm 1 \bmod n$, . We need to show that $B_t \leq |\mathbb{Z}_n^*|/2$ to complete the proof in this case.

B_t is a subgroup of $\mathbb{Z}_n^*$
1. $a, b \in B_t \implies (ab)^{m \cdot 2^t} = a^{m \cdot 2^t} \cdot b^{m \cdot 2^t} = (\pm 1) \cdot (\pm 1) = \pm 1 \bmod n$, so $ab \in B_t$
2. $1^{m \cdot 2^t} = 1 \bmod n \implies 1 \in B_t$
3. $a \in B_t \implies (a^{-1})^{m \cdot 2^t} = (a^{m \cdot 2^t})^{-1} = (\pm 1)^{-1} = \pm 1 \bmod n$

Now if we can show that B_t is a proper subgroup of $\mathbb{Z}_n^*$, that is $B_t \neq \mathbb{Z}_n^*$, then we can apply Langrange's Theorem as we did in Proposition 5.17, to show that $|B_t| \leq |\mathbb{Z}_n^*|/2$.

The definition of t implies the existence of $a \in \mathbb{Z}_n^*$ such that $a^{m \cdot 2^t} = -1 \bmod n$. Since $n \geq 3$ is not a prime power, we can factor n as $n = cd$ where $\gcd(c, d) = 1$. Now by the Chinese Remainder Theorem, we have that there exists $b \in \mathbb{Z}_n^+$ satisfying

$$b = a \bmod c$$
$$b = 1 \bmod d$$

The theorem also gives us that $b = a \bmod n$ (this follows from the result $b = \sum_{i-1}^{k} b_i N_i a_i \bmod N$). This implies that $n \mid (b-a) \implies \exists k \in \mathbb{Z}_n^*$such that $b - kn = a$. Note that $\gcd(b, n)$ divides the left side of this equation so $\gcd(b, n) \mid a$ and $a \in \mathbb{Z}_n^* \implies \gcd(a, n) = 1$. Thus, we have that $\gcd(b, n) = 1 \implies b \in Z_n^*$. However,

$$b^{m \cdot 2^t} = a^{m \cdot 2^t} = -1 \bmod c$$
$$b^{m \cdot 2^t} = 1^{m \cdot 2^t} = 1 \bmod d$$

imply that $b^{m \cdot 2^t} \neq \pm 1 \bmod n$, so $b \notin B_t$.Thus, $B_t \neq \mathbb{Z}_n^*$.

The other possibility is that the input n is both a prime power and a Carmicheal number. However, no Charmicheal number is a prime power (the proof for this requires number theory beyond what we have not introduced, so we will just accept it), so this case is not relavant. Therefore COMPOSITE $\in$ RP or equivalently, PRIME $\in$ co-RP. $\square$

6. One-way Functions

As we state earlier, one-way functions are functions that are easy to compute, but difficult to invert. Many cryptosystems depend on the comlpexity theoretic gap created by one-way functions between those trying to share a secret, say Alice and Bob, and an intruder trying to eavesdrop, Eve. Alice can encrypt a message with a one-way function and Bob, with his secret key, can easily invert the function and decrypt the message. Without this key, it is difficult for Eve to invert the function. We need to determine what the nature of the complexity this gap should be for an effective cryptosytem, in essence, how exactly must the one-way function be difficult to invert.

For such a gap to exist, there must be some limit to Eve's computational resources, but it would be unreasonable to assume that Alice and Bob have any greater computational abilities than Eve, so we will assume that all parties can only perform polynomial time computations. This means that for it to be easy for Alice and Bob to encrypt and decrypt a message, there must be a polynomial time algorithm to do so, but what does it mean for it to be difficult for Eve to decrypt the message. One option would be for Alice and Bob to use a system in which the problem of decryption is NP-hard for Eve, but this will not necessarily be a secure system since a problem can be NP-hard if many instances are easy, but a few instances are particularly difficult because hardness is measured in the worst-case situation. Hence, it is not sufficient for the problem of decryption to be NP-hard for an effective cryptosystem.

On the other hand, it is also unreasonable to suppose that it impossible for Eve to retrieve the message since if she were to randomly guess a message, then she would still have a very small chance of guessing the correct message.

So given these requirements, a reasonable level of security to demand from a cryptosystem is if Eve uses any probabilistic polynomial time algorithm then the probability that she decrypts a random encrypted message is negligible, where negligible is defined as follows

Definition 6.1. A function $r : \mathbb{N} \to \mathbb{N}$ is negligible if for any polynomial $p : \mathbb{N} \to \mathbb{N}$, there is an integer k_0 such that $r(k) < 1/p(k)$ for $k \geq k_0$. A negligible function is one that is less than any polynomial after a finite number of integer values.

This definition ensures that if Eve repeats an attack that has a negligible probablity of succeeding a polynomial number of times, she is still unlikely to succeed since each attempt would be represented a seperate negligible functon r_i and the probability that one attempt would succeed would be the sum of these functions, and since each of these functions is small relative to any polynomial for k sufficiently large, their sum is small as well.

In addition to these secuirty requirements of inverting a one-way function, we need to impose one more condition. To invert $f(x)$ means to find the preimage of a value. To invert $y = f(x)$ means to find z such that $f(z) = y$. Let the set of preimages be $f^{-1}(f(x)) = \{z \in \{0,1\}^* \mid f(z) = f(x)\}$. We want a function to be difficult to invert because the preimage is difficult to find, not because the preimage is too long to invert in polynomial time. To avoid this problem, we will suppose that the input to any inverting algorithm will include both the image to invert, $f(x)$, as well as the length of x in unary. So if $|x| = k$, then the input to an inverting algorithm would be $(f(x), 1^k)$ and the algorithm should output a

preimage $z \in f^{-1}(f(x))$. This ensures that at least one preimage of $f(x)$ can be written in polynomial time. We can now give a formal definition for a one-way function

Definition 6.2. A function $f : \{0,1\}^* \to \{0,1\}^*$ is one-way iff
(1) f is computable in polynomial time
(2) For any probabilistic polynomial time algorithm A, the probability that A successfully inverts $f(x)$, for random $x \in_R \{0,1\}^k$, is negligible.
using our definition of negligible, we can rewrite 2 as
(2') For any positive polynomial $q(\cdot)$ and any probabilistic polynomial time algorithm A the following holds for k sufficiently large:

$$\Pr[A(f(x), 1^k) \in f^{-1}(f(x)) \mid x \in_R \{0,1\}^k] \leq \frac{1}{q(k)}$$

One computation that is believed to be very difficult is factoring the product of two large prime numbers. Currently, the most efficient factoring algorithms are the Quadratic Sieve and the Number Field Sieve, both are probabilistic algorithms that have expected runnings times of $O(\exp(\sqrt{c_1 \ln N \ln\ln N}))$ and $O(\exp(c_2(\ln N)^{1/3}(\ln\ln N)^{2/3}))$, where $c_1 \simeq 1$ and $c_2 \leq (64/9)^{1/3} \simeq 1.923$. The following is the formal statement that given a product of two large primes, an instruder should have a negligible chance of factoring it.

Assumption 6.3. *Factoring Assumption. For any positive polynomial $r(\cdot)$ and probabilitic polynomial time algorithm A, the following holds for k sufficiently large*

$$Pr[A(n) = (p,q)] \leq \frac{1}{r(k)}$$

where $n = pq$ and p, q are random k-bit primes.

Proposition 6.4. *Under the factoring assumption, Prime multiplication, pmult, is a one-way function.*

Proof. pmult is computable in polynomial time since it is simply multiplying two integers, so condition (1) of Definition 6.2 is satisfied, and the factoring assumption gives us (2) of the definition. □

Since this proof depends on the factoring assumption, it is still not certain that pmult is a one-way function. If one-way functions do in fact exist, there are large implications for complexity theory such as the following.

Theorem 6.5. *If one-way functions exists, then $NP \neq P$*

7. The RSA Cryptosystem

The RSA cryptosystem works as follows

(1) Setup. Bob secretly chooses two large distinct prime p and q, and then forms his public modulus $n = pq$. He then chooses his public exponent e to be coprime to $(p-1)(q-1)$, with $1 < e < (p-1)(q-1)$. The pair (n, e) is his public key that he publishes. His private key is the unique integer $1 < d < (p-1)(q-1)$ such that

$$ed = 1 \bmod (p-1)(q-1).$$

(2) Encryption. Alice has a message M she wishes to send to Bob. She splits the message into a sequence of blocks $M_1, M_2, \ldots, M_t$ where each M_i satisfies $0 \leq M_i < n$. She knows Bob's public key, (n, e), and she uses it to encrypts these blocks as

$$C_i = M_i^e \bmod n$$

and sends the encrypted blocks,$C_1, C_2, \ldots, C_t$ to Bob.

(3) Decryption. Bob decrypts the blocks using his private key d by raising each encrypted block to the dth power resulting in

$$M_i = C_i^d \bmod n$$

Proposition 7.1. *Decryption in RSA works*

Proof. From the setup, we have that $ed = 1 \bmod (p-1)(q-1) \implies \exists k \in \mathbb{Z}$ such that $ed = 1 + k(p-1)(q-1)$. Thus we have

$$\begin{aligned} C_i^d = (M_i^e)^d = M_i^{e \cdot d} = M_i^{1+k(p-1)(q-1)} \\ = M_i (M_i^{p-1})^{k(q-1)} = M_i \bmod p \end{aligned}$$

Since either p divides M_i, in which case $M_i = 0 \bmod p$ and both sides are 0, or p does not divide M_i, in which case $\gcd(p, M_i) = 1$, then by Fermat's Little Theorem, we have $M_i^{p-1} = 1 \bmod p \implies (M_i^{p-1})^{k(q-1)} = 1 \bmod p \implies M(M_i^{p-1})^{k(q-1)} = (M_i^e)^d = M_i \bmod p$. By similar argument, we have

$$(M_i^e)^d = M_i \bmod q$$

p and q are distinct primes so by the Chinese Remainder Theorem,

$$C_i^d = (M_i^e)^d = M_i \bmod n$$

□

Now it is left to show that the setup and encryption\decryption are easy for Alice and Bob and that the system is secure, or deciphering the message is an intractable problem for Eve.

First, we will show that the setup is easy. Bob can choose two random k-bit integers using the random integer algorithm (5.4), and test their primality using the Miller Rabin Algorithm in polynomial time. Note that a k-bit integer can be at most $N = \sum_{i=1}^{k} 2^{k-1}$. By the Prime Number Theorem, $\pi(N)$, the number of primes $p \leq N$ is bounded below by a polynomial $O\left(\frac{N}{\log N}\right)$. For k large enough, $O\left(\frac{N}{\log N}\right) \geq \frac{1}{k}$ thus Bob can expect to find a prime in less than k picks. He then forms n by multiplication, a basic operation. To find e, he chooses k-bit integers at random untill he finds one that is co-prime with $(p-1)(q-1)$. To find d, he can use Euclid's algorithm. Therefore, the setup can be completed in polynomial time.

Next, we will show that encryption/decryption are easy. Both encryption and decryption require only exponentiation mod$n$, so both can be done in polynomial time.

Finally, we will discuss the security of the system. Eve faces a difficult problem, computing eth roots mod$n$ since she needs to find the private key. One way to do this would be for her to factor n to find p and q and she construct the private key herself as Bob did during the setup. Thus, if factoring is easy, the RSA is insecure; however, the converse may not be true. Whether or not the problem of factoring

and breaking RSA are equivalent is a large open problem, the solution to which will better cement RSA's security.

Acknowledgements. I'd like to thank my mentor David Chudzicki for his guidance and feedback.

References

[1] Talbot, John and Dominic Welsh. Complexity and Cryptography. Cambridge: Cambridge University Press, 2006.
[2] Rothe, Jörg. Complexity Theory and Cryptology. Berlin: Springer, 2005.