MATH3001, Project in Mathematics

Computational Complexity

Alicia Brogan, 201218628

Contents

1 Introduction

Computational complexity is concerned with finding how hard a problem is to solve and is focussed on the efficiency of a computation rather than the computability itself. Computational efficiency is defined as "quantifying the amount of computational resources required to solve a given task" in (Arora & Barak 2009, p.xx). For example, sorting lists into size order is seen as a very easy task, however, as the list grows it becomes increasingly time consuming to sort through the whole list. This is because at every stage of sorting, there are more comparisons to be made. The most intuitive method of sorting a list would arguably be to first find the smallest number in the list and and put it in position 1, then do the same with the remaining list, repeating until there are no numbers left to compare[1]. Another way to sort a list of numbers would be to use the merge sort algorithm. This works by dividing the list(s) in half repeatedly until you have lists with one number in each, which can be considered sorted since it only contains one element. Then by merging the adjacent sorted lists repeatedly, until eventually all sorted sublists will be merged and thus sorted. Either method proves that the problem is computable, however, the second is far quicker (and therefore more computationally efficient). Splitting lists is such an effective method that if you were asked to guess a number between 1 and 100, it is proven to be doable in 7 or less guesses by asking if the number is smaller or greater than 50, then splitting the remaining list in half again and again until you find the number. This illustrates that just because something is theoretically proven to be computable, this does not mean much if the algorithm simply takes too much time to do the task, as this effectively makes the problem incomputable in real life scenarios.

2 Modelling computation

For us to be able to measure the efficiency of a computation, there must be a model of computation. This model would ideally be such that any physically realisable computation could be simulated. Since we are specifically looking into the efficiency of the computations, the machine must also minimise loss in efficiency. A simple mathematical model called the Turing machine, described by Alan Turing in 1936, is capable of doing this.

2.1 Language of computation

To express a computation theoretically, we must first define a formal language in which to represent the problem. This language is derived from an alphabet of finite symbols that is denoted Σ. This alphabet can be any set of symbols. Some common examples are the English alphabet, so $\Sigma = \{a, b, c, ..., z\}$ and the Binary Number System[2], making $\Sigma = \{0, 1\}$. Then, Σ^* is defined as the finite set of strings of symbols from Σ. Allowing Σ to be the English alphabet as before, then $abcde \in \Sigma$ for example.

Now, the definition for the language of a binary number system (Arora & Barak 2009, p.3) can be found as:

Definition 2.1.1 (Language)**.** *A **language**, L, or **decision problem** is identified as a boolean function f with the subset $L_f = \{x : f(x) = 1\}$ of $\{0, 1\}^*$.*

[1] This is a basic presentation of the selection sort algorithm. For further information on this algorithm and other sorting algorithms such as merge sort, see (Sedgewick & Wayne 2011, section 2.1)

[2] Binary digit or "bit" (A single 0 or 1) is described as "the most elemental unit of information" in (Christ & Wernli 2014, section 13.3.1).

This definition implies that the language can be interpreted not only as being the answer of all possible solutions to the problem being computed, but can be seen as the problem itself. The problem therefore becomes 'if given an input string $\sigma \in \Sigma^*$, is $\sigma \in L$?'

Example 2.1.2. *Here is an intuitive example of this. Let $\sigma = \{a, b, c, ..., z\}$ such that σ^* is the set of all strings made from the alphabet. We can then let language L be all valid English words. The problem then becomes 'given $\sigma \in \Sigma^*$, is σ a valid English word?'*

2.2 (Deterministic) Turing Machines

With these concepts in mind, we are now able to describe a simplistic model of computation capable of capturing these problems. The (Deterministic) Turing machine is an abstract computer that can compute problems when given a set of instructions. The setup is simple compared to the sheer power of the machine, which we will see later on. Now, we formalise a definition for a Turing Machine, which can be found in (Sipser 1996, p.168).

Definition 2.2.1 (Turing Machine)**.** *A **Turing Machine** (TM) is a 7-tuple, $\{Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$, where Q, Σ, Γ are all finite states and:*

- *Q is the set of states,*
- *Σ is the input alphabet not containing the blank symbol, denoted $\square$,*
- *Γ is the tape alphabet, where $\square \in \Gamma$ and $\Sigma \subset \Gamma$,*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,*
- *$q_0 \in Q$ is the start state,*
- *q_{accept} is the accept state, and*
- *q_{reject} is the reject state, where $q_{reject} \neq q_{accept}$.*

Here, the 7-tuple M=$\{Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$ describes a given Turing Machine's program, which comprises the software. An important distinction is between Σ and Γ. Σ describes the **input** alphabet and **does not** contain the blank symbol $\square$. Γ is the **tape** alphabet and **does** contain the blank symbol, as well as being the union of the blank symbol and the input alphabet. As the names imply, the input alphabet are the symbols you are able to input into the machine and the tape alphabet is the alphabet of symbols that can appear on the tape, which is dependent on what you input. There is then a (finite) set of states in which the Machine can be in. These are 'designators' of the instructions to be carried out. The set must contain a **start** state, which the machine begins in, and two **halt** states such that if the machine moves into one of them, the computation stops. If it stops on the accept state, the input string will be accepted and similarly, in the reject state the string will be rejected. Since these two halt states cannot equal each other, a given input string can only be accepted **or** rejected (or in some cases, the machine can never halt and continue forever). The δ function is the partial function[3] which lays out specific instructions, which can be represented as a state table or a state transition diagram. It essentially determines all the steps of the algorithm being programmed. The function takes as input a certain state (which cannot be either halt state since they lead to the machine stopping) and symbol that is being read at that given step. It then outputs a state in which the program should now move to (which could be to

[3]It is partial since some inputs may not give an output.

move to the same state it is already in), replaces the input symbol (which could be with the same symbol), and outputs one symbol from $\{L, R\}$, representing moving either left or right in the input string respectively.

Example 2.2.2. *Suppose machine M is a Turing Machine in state $q \in Q$ and the current symbol being read by the machine is $\gamma \in \Gamma$ and define δ to be $\delta(q, \gamma) = (q', \gamma', L)$. This means that when γ is read on state q, the machine moves into state q', rewrites the symbol as γ' and moves the machine to read the symbol directly to the left of the present γ.*

A Turing Machine, just like a modern-day computer, comprises of hardware and software. In our definition only the software is discussed since the hardware simply makes the machine physically realisable. The hardware consists of:

1. an infinite tape which is divided into cells. This tape can be defined as a one-way tape such as $\mathbb{N}$ or a two-way tape such as $\mathbb{Z}$. The input string is always written from cell 0 with a single symbol in each cell. Every other cell contains the blank symbol. If the TM is required to store information, it can do so on the tape.

2. a read/write head that can read the current cell and edit it, so the TM can rewrite the symbol in the cell to anything in Γ. The read/write head always begins on cell 0.

3. a finite state control which issues commands for the machine to undertake; as the name implies, it controls what state the machine is in. This always starts in q_0 and **if** it halts, will always finish in a halt state.

NB-Suppose you are using a one-way tape, i.e. one that has a start point at one end. Suppose you are on the first cell and told to move left, how does the machine deal with this? It simply stays on the left-end symbol in that move as it can't just fall off the end of the tape, as mentioned in (Sipser 1996, p.168).

2.3 Example of a Deterministic Turing Machine

To properly see how the DTM works, here is an example I have written using Morphett's Turing Machine Simulator. Our machine determines whether an input string has an even number of 0s or not. Here, $\Sigma = \{0, 1\}$, so Σ^* is the set of all binary strings. The language, L, is every binary input which has an even number of 0s, for example 0110. The code is as seen in Appendix A. Following the syntax of the program, at every line there is a tuple describing everything in δ. From left to right, the 5-tuple reads

$$\{Q_{currently}, \Gamma_{currently}, \Gamma_{next}, r, Q_{next}\},$$

where $Q_{currently}, \Gamma_{currently}, \Gamma_{next}, r, Q_{next}$ denotes the current state, the current symbol being read, the symbol that will replace the current symbol, moving to the right and the next state the machine will enter respectively.

This is a very simple example of a TM and so here, the machine always moves the read head to the right. In other more complex Turing Machines, of course where there is an "r" in the tuple, there could be a "l" instead, denoting moving left or a "*", denoted below. Also, we may note that for this example, $\Gamma_{currently} = \Gamma_{next}$ since this machine never changes the symbol on the tape.[4]

As it says in the syntax on Morphett (n.d.), the "*" symbol can be used in two ways:

[4]For other examples, see Morphett's Simulator. The "Binary Palindrome" program shows why it may be necessary to change symbols and move to the right **or** left. In this program's case, it must 'clear' symbols to blank symbols once they have been checked to match and then must go back to the beginning of the tape.

- In $\Gamma_{currently}$, * can denote any symbol, since it just implies that it does not matter what the symbol currently is, i.e. the next 'step' the machine takes is not dependent on the current symbol.

- In Γ_{next} and Q_{next}, * denotes no change in symbol or state. It can also denote keeping the read/write head on the same symbol.

The TM decides whether there is an even amount of 0s or not, in essence, by using the fact that the machine started with an even amount of 0s (namely, 0 of them) that it has read and then changes between odd and even states as the number of 0s increases until we reach the end of the string, at which time the machine will enter a halt state and accept or reject the string.

The machine has 5 states in total, one being the initial state, two being the halt states and states odd and even. As said above, the machine knows it starts with an even amount of 0s and so the initial state does the same as the even state. If they read a 0,

- State 0 is the initial state. As said above, the machine knows it starts with an even amount of 0s and so when reading cell 0, if it reads a 1, then there are still no 0s and the machine enters the even state. If it reads a 0, then it knows there is an odd number of 0s and moves into the odd state.

- State even determines whether the machine has an even number of 0s, given there is an even number of zeros up to this cell. It therefore has the same tuple as state 0, except for the first element being the different states.

- State odd does the same as state even, however it is given that before this cell, there was an odd number of 0s. Therefore, if the cell contains a 0, then there are now even 0s and the machine moves into state even. If the cell contains a 1, the machine stays in state odd(Therefore, in either state odd or even, if the machine reads a 0 next, it changes to the other state, and if it reads a 1, it stays in the same state).

- State accept halts the machine and accepts the string as having an even number of 0s. The machine will always halt once it reaches the final cell of the input string, since once it reads a blank cell, it knows that the string is over. If the machine reads a blank symbol in the even state, it accepts the input string since up until the end of the string, there were an even number of 0s.

- State reject halts the machine and rejects the string. If the machine reads a black symbol in state odd then it moves into the reject state.

2.4 Deciders

A Turing machine of this kind is called a **decider**. This means that on every (finite) input string, the machine will always halt, since intuitively the machine has then decided whether the input string has the property we are testing for or not.

Definition 2.4.1. *A language L is **recognized** by M if $L = \{x \in \Sigma^* : M \text{ accepts } x\}$. L is **recognizable** if some DTM recognizes it.*

This means that given the Turing Machine is a decider, the Turing machine solves the problem L and so M can be considered an **algorithm** for solving L.

Changing the definition of a Turing Machine slightly, we can define a TM that computes functions; replacing q_{accept} and q_{reject} with q_{halt} (a halt state), and keeping everything else

the same. Given input/(s), the machine is capable of outputting a result based on a rule, which is equivalent to the output of a function for that given input/(s). This can be done with any finite number of input values. When the machine enters this halt state, as the name suggests the machine will halt and the string on the tape, written from cell 0, will be outputted. These machines will only describe partial functions, since we are only interested in going from $\mathbb{N}$ to $\mathbb{N}$, and so some functions may not produce an output. Hence, a Turing Machine, M, can be associated with a partial function, $f : \mathbb{N} \to \mathbb{N} \cup \{\Box\}$, where the machine M will represent $x \in \mathbb{N}$ in binary form, (Immerman 2018).

2.5 Changing the model

The definition of a Turing Machine is not the important part of the model, it is the concept of such a machine. Changing small parts of the definition does not affect how powerful the machine is or what it is able to decide. For example, in 2.3, we briefly mentioned that "*" could denote "no move" of the read head for the program used. This is not in our definition of a TM in 2.2.1, since the set is defined as $\{L, R\}$. Changing our transition function from

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\} \tag{1}$$

to

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, *\} \tag{2}$$

has no consequence on the machine, since a TM, M, with transition function 1 can simulate a TM, N, with transition function 2. This is clear since for M, * could be simulated by moving left and then right whilst not changing the symbol, for example. This can also be said for changing the alphabet of the TM, or allowing for different types of tape (one-way or two-way)(Arora & Barak 2009, p.16).

This is also true of the **multitape Turing Machine**. Formally a multitape Turing machine can be defined as:

Definition 2.5.1 (Multitape Turing Machine)**.** *A **Multitape Turing Machine** is described with a transition function, δ,*

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, *\}^k,$$
$$\delta(q_i, a_1, ..., a_k) = (q_j, b_1, ..., b_k, L, R, ..., L)$$

where k is the number of tapes, q_i, q_j are specific states, $a_1, ..., a_k, b_1, ..., b_k$ are the symbols from Γ the machine is reading on each tape and the symbols the state will change those symbols to respectively, and the set $\{L, R, ..., L\}$ is the set of size k, denoting the moves of the reader for each tape, (Sipser 1996, p.167).

This is an extension of the definition for a single tape Turing Machine. The machine is able to comprehend many tapes, rather than just one.

The changes you can make to the Turing Machine are vast. Similar changes like this, found in *Variation of Turing Machine* (2019), also include but are not limited to:

- Multi-dimensional Tape Turing Machines- Not only can the head move left and right, it can also move up and down,
- Multi-head Turing Machines- These have a single tape with multiple heads that can simultaneously read different cells.

Another interesting variation of the Turing Machine, is the **nondeterministic Turing machine**, (NDTM). This is different from the forementioned Turing machines since these were all **deterministic**. Given an input, the previous machines had to follow an involuntary, linear path through the machine, which the delta function describes. This is not the case with NDTMs. According to Sipser (1996), the transition function of a nondeterministic Turing machine is:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

The $\mathcal{P}$ implies the possibilities of each tuple the transition function can enter.The NDTM works instead by carrying out every possibility, each splitting into different paths, at each state. So, at the initial state, every possible way to then proceed on that input is carried out simultaneously and so on for each state reached by each path. If **any** of these paths end in an accept state, the computation can halt and the machine accepts the input. Otherwise, all paths will end in the reject state and the input is rejected.

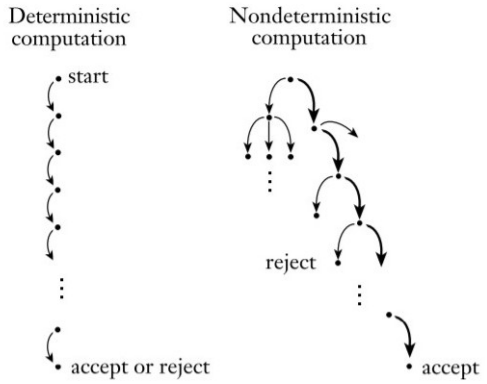

Figure 1: The computational paths taken by a DTM and NDTM, both ending in an accept state, as shown in Sipser (1996).

All these variants on Turing Machines may seem more powerful than a TM described by 2.2.1. It is natural to assume that a TM with another tape, for example, would be capable of computing a larger class of problems than a single-tape TM, however, this can be shown to not be the case. Each of the above are as powerful as one another, due to the fact that each can indeed be simulated by 2.2.1.
We can show that a multitape TM, M, can be simulated by a single-tape TM, S:

Theorem 2.5.2. *Every multitape Turing machine has an equivalent (meaning it has another model that is capable of computing the same problems) single-tape Turing machine.*

Proof. We can prove this generally for a multitape Turing Machine, M, with k tapes, thus proving the claim for every multitape TM.
Let M be a multitape TM with a read/write head for each tape. Suppose M has k inputs, each written on one tape of M from cell 0, surrounded by the blank symbol, $\square$. By introducing new symbols to the tape alphabet of the single-tape TM, we can write and compute these k inputs on a single tape. Let S be the single-tape single-head TM we are simulating M with. A delimiter symbol, $\#$, is introduced to specify boundaries between each input, by putting a $\#$ in the cell separating each input. Symbols a_i, where $0 \leq i \leq k-1$, pairing each original symbol a_i are introduced. These are used to show where each of M's read heads would be in M. They act as virtual read/write heads. We now have a model on a single tape to simulate the multitape, where the tape alphabet of S is $\Sigma \cup \{\#\} \cup \{a_i : 0 \leq i \leq k-1\}$ and

everything else described in 2.2.1 remains the same. This idea can be found in (Sipser 1996, p.177), however this proof is rewritten in my own words. □

This also shows how a multi-head single-tape Turing Machine can be simulated by 2.2.1. This would be the equivalent machine as S, just with real tape heads and not dotted symbols. Clearly, this can be simulated by changing the tape heads to the dotted symbols, a_i as described above. The same can also be said for NDTMs, found in (Sipser 1996, p.178):

Theorem 2.5.3. *Every nondeterministic Turing machine has an equivalent deterministic Turing machine.*

The general idea is given here to be convince that the statement above holds.
The same principle of finding a DTM, D, to simulate any NDTM, N is used. N can be represented as a specific tree for a given input, as seen in 1, and we use D to follow each branch of the tree in a breadth-first search manner. This machine would check each branch as it is followed for an accept state. Only if this happens, D moves into the accept state and accepts the input. Hence, if the input is not accepted, the machine never halts.

In fact, it can (and will later) be shown that any variation of the TM can be simulated by 2.2.1.

2.6 λ-calculus

Very briefly, λ-calculus is another type of computational model, created around the same time as TMs by Alonzo Church, from the perspective of functions. They use **abstraction** to apply functions to argument, Alama & Korbmacher (2021). The function abstractions work by following rules to write it into the correct form.

Example 2.6.1. *The simplest example to explain the form of λ-calculus is the identity function, mapping an argument (input) to itself.*
The function:

$$f(x) = x \tag{3}$$

becomes

$$\lambda x.x \tag{4}$$

The abstractor λ is followed by the variable of the function, followed by a dot, followed by the expression on the right-hand side in 3, and hence becomes 4.
Then, we need to compute what a certain argument a is when λ is applied to it.

Example 2.6.2. *For example, for the λ-term $\lambda x[x^2 - 2 \cdot x + 5]$, applying λ to $a = 3$:*

$$\begin{aligned}(\lambda x[x^2 - 2 \cdot x + 5])3 &= 3^2 - 2 \cdot 3 + 5 \\ &= 9 - 6 + 5 \\ &= 8\end{aligned}$$

This can be done further with any number of variables, and thus λ-calculus creates a computational model for an algorithm as a function. As shown, there are no states used like they are in Turing Machines, which, as known, represent the algorithms with its machine. λ-calculus, with its applications in logic, is completely different from a mechanical perspective and in the fields the idea of computation is being applied to. λ-calculus computation and TM

computation would therefore seem to appear completely separate and unrelated, however we will see this is not the case.

Church eventually concluded that everything that was computable over the natural numbers has an equivalent λ-term. In his 1936 paper, Church, p.356 says:

"define the notion ... of an effectively calculable function of positive integers by identifying it with the notion of a recursive function of positive integers (or of a lambda-definable function of positive integers"

Effectively calculable means that the function is computable and these make up the same set as the set of λ-terms. This leads to **Church's thesis**, Copeland (2000):

A function of positive integers is effectively calculable only if recursive.

Therefore, we have that anything that **can** be calculated, can be done so using λ-calculus.

2.7 Church-Turing thesis

Turing also concluded similarly of his machines, around the same time independently of Church. In Turing's 1948 paper, he states that:

LCMs [logical computing machines: Turing's expression for Turing machines] can do anything that could be described as "rule of thumb" or "purely mechanical".

Here, purely mechanical means the same as being effectively calculable. It is clear now that a TM described as in 2.2.1 can also be used to describe any λ-term from λ-calculus. The equivalence of these notions is known as the **Church-Turing thesis**[5]. First named this in 1967 by Stephen Kleene, the thesis is described in Sipser (1996) as:

Intuitive notion of algorithms equals Turing Machine algorithms.

So computationally, these different concepts are capable of solving the exact same problems because they are equivalent and so a problem is computable by a TM if and only if it can also be computed in λ-calculus.

These two models do not just model each other; they actually are capable of simulating any computation that can be modelled by any given model.Arora & Barak (2009) states that "every physically realisable computation device can be simulated by a Turing machine", which is a very powerful concept as it means we can limit our attention when considering models to just the Turing Machine. Not only do they simulate them, they do so with little loss of efficiency, since the set of all problems solvable "efficiently" is at least equal to any other method of computation, (Arora & Barak 2009, p.9).

3 Complexity

Turing Machines are not meant for practical use; they are a tool in which to define computable and thus derive theorems concerning their capabilities, which include their running times. As mentioned in 1, in practice it is useful to know that an algorithm can decide a problem, however, if it will take thousands of years to do so then this is about as useful as it being uncomputable.

Algorithms can vary in time taken because number of steps by the algorithm has to increase

[5]Thesis because it is not proven, however, it is generally accepted as true, with no counterexamples disproving the statement.

every time the input length increases. It is hard to predict exactly how long an algorithm will take, but the relationship between run time and input length can give much insight into its efficiency, since run time for a decidable machine only becomes significant as the input length increases. This is known as **asymptotic analysis**, (Sipser 1996, p.276). When the input in short, the run time of different algorithms won't compare much. Efficiency can measure a variety of things, not just time. If a machine were to take up more space, for example, but the same amount of time, it can clearly be seen as less efficient, just in a different way. Here, we will only focus on **time** efficiency.
Formally, the running time of an algorithm, in (Sipser 1996, p.276), is:

Definition 3.0.1 (Running time)**.** *Let M be a deterministic Turing machine that halts on all inputs. The **running time** or **time complexity** of M is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M "runs in time $f(n)$" and that M is an "$f(n)$ time Turing machine". Customarily we use n to represent the length of the input.*

This definition is a **worst-case** analysis, since $f(n)$ is the **maximum** number of steps for any input of that length. This definition does not take into the account the time taken for any physical processes a TM would carry out, such as the time taken to actually perform a step by moving the read/write head. These can be ignored since we are only interested in the TM as a theoretical model. Even if we wanted to study a physical model, the time taken by these would be insignificant compared to the total run time taken by TM, as said in (A Mollin 2006, p.66).

3.1 Big-O Notation

Notation for asymptotic analysis is called **asymptotic notation**. One type of asymptotic notation is big-O notation, defined in (Sipser 1996, p.277) as:

Definition 3.1.1 (Big-O notation)**.** *Let $\mathbb{R}^+$ be the set of nonnegative real numbers. Let f and g be functions, $f, g : \mathbb{N} \to \mathbb{R}^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,*

$$f(n) \leq cg(n).$$

*When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** for $f(n)$, or more precisely, that $g(n)$ is an **asymptotic upper bound** for $f(n)$, to emphasize that we are suppressing constant factors.*

Big-O notation is "the order of magnitude of the complexity"(A Mollin 2006, p.67). This means that big-O defines an upper bound on the magnitude of the number of steps the algorithm *could* take for a given input n, as long as n is sufficiently large.
The equals sign in this definition can be seen as misleading, since $f(n) = O(g(n))$ could imply that, similarly, $O(g(n)) = f(n)$, which is always false since $g(n)$ is an upper bound for $f(n)$. An alternate notation for $f(n) = O(g(n))$, introduced by I.M. Vinogradov in 1937, is $f << g$, found in (A Mollin 2006, p.67). It implies the one-way nature of $f(n) = O(g(n))$ much better.
For example, as de Bruijn (2010) states on page 7, $O(x) = O(x^2)$ is true, however, $O(x^2) = O(x)$ is false. Since $= O(x)$ means "something less than x" in de Bruijn (2010), it is clear why this is not the case. Clearly, everything smaller than x is smaller than x^2, but everything smaller than x^2 is not smaller than x^6. This also shows how $O(g(n))$ is not unique. It is just an upper bound for f, and so can be *any* upper bound, so long as it satisfies 3.1.1.

[6] N.B. As with other mathematical functions, the variable can be named as anything, and so de Bruijn's $O(x)$ is no different to $O(n)$.

Example 3.1.2. *Starting with a function I have randomly picked, let $f(n) = 6n^4 + 5n^2 + 12n + 4$. Then to find $O(g(n))$, we must find g(n) by choosing the term with the highest order, namely $6n^4$. We then disregard the 6 since it is a constant and so $g(n) = n^4 \implies f(n) = O(n^4)$ is an upper bound on f.*
In line with our definition, if we let $c = 7$ and $n_0 = 4$, then $f(n) \leq cg(n)$ for all $n \geq n_0$. We can also say that any $g_{other}(n) = n^x$ where $x \geq 4$, since $O(n^4)$ is an upper bound.

As mentioned in 1, the merge sort algorithm is more efficient than the selection sort algorithm. We can quantify this. As described, the selection sort would take $n - 1$ comparisons (steps) to find the smallest number. This is because it compares down the list, comparing the smallest previously with the next. If the next is smaller, it swaps them, thus meaning the smallest will always get to position 1 in $(n - 1)$ steps. The next smallest would then take $(n - 1) - 1 = n - 2$ steps and so on until the last number takes 1 step to sort.

$$\begin{aligned}(n-1) + (n-2) + ... + 2 + 1 &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)((n-1)+1)}{2} \\ &\qquad \text{(By the summation formula for } n\text{)} \\ &= \frac{1}{2}(n^2 - n)\end{aligned}$$

So selection sort has a complexity $O(n^2)$ since as n increases, n^2 dominates the expression.
The merge sort can be shown to have a complexity of $O(n \log_2 n)$, shown briefly in (Sedgewick & Wayne 2011, p.274):
For any input n, the list will be sorted by halving the input list each time, and so the number of times it does this process is always $x = 2^N$, (N is a natural number), where x is the next integer for which 2^N will be an integer (so for 7, the next smallest number of the form 2^N is 8 and so $N = 3$) and so the tree has precisely N levels in it. Halving the lists each time means there are N sorts and so $\log_2 x$ steps. It then takes n steps per divide of list, and so there are $nN \implies n \log_2 n$ maximum steps in the algorithm and so complexity $O(n \log_2 n)$ is an upper bound on time taken, which is quicker than $O(n^2)$ for all sufficiently sized n.

We can also show that the TM described in 2.3 has complexity $O(n)$, since every string is decided in at most n steps. This is because the function checks each digit of the input once and then makes a conclusion, thus only ever taking as many steps as there are digits in the input.

3.2 Complexity Classes

The complexity of problems and their related algorithms can be sorted into different classes of "hardness". These classes are sets of problems, whose algorithms have all been categorised as taking up similar amounts of a resource (as mentioned, we are interested in time).

The first defined class we will look at is **P**. All the problems above, with their algorithms defined in big-O notation, are examples of problems in class **P**. Generally, this is because all these problems can be solved in **polynomial time**, meaning that an upper bound can be expressed for them in the form $O(n^k)$ for some constant $k > 0$.

NB-This is clearly true for all above algorithms except for the merge sort, whose complexity can be defined as $O(n \log_2 n)$, which is not in the form $O(n^k)$. However, we already know that the selection sort is bounded by $O(n^2)$ and the merge sort is quicker for large n, so merge sort can also be bound by $O(n^2)$ and so is in **P**.

Class **P** is defined in (Sipser 1996, p.286), as:

Definition 3.2.1 (Class **P**). ***P*** *is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,*

$$\boldsymbol{P} = \bigcup_k TIME(n^k).$$

$TIME(n^k)$ *refers to the set of every problem that is computable in time* n^k.

The class **P** is the link between things that are seemingly efficient and thus "capture the notion of decision problems with "feasible" decision procedures"Arora & Barak. Hence, if we can find an algorithm that is in class **P**, the Turing Machine can describe an algorithm that a well-designed PC will be able to run in fractions of seconds.

The next class has a similar description; some problems can be *verified* in polynomial time. Many puzzles belong to this set, with Sudoku being one. The commonly 9x9 grid problem can be accepted as being a lot harder to solve from scratch than it is to check that a solution is correct. The reason is that there is no efficient algorithm (yet) for finding the answer to a Sudoku problem, they must be done by directed methods and intuition. However, you can clearly check a solution efficiently and so in polynomial time. This informally describes the class **NP**.
We must first define the notion of a verifier to then define the class **NP**.

Definition 3.2.2 (Class **NP** definition I). *A* ***verifier*** *for a language A can be described by a deterministic Turing machine, namely M, where*

$$A = \{w : M \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

for an input string w *and some string* c*, which is the solution of the computation. A* ***polynomial time verifier*** *is the Turing Machine M that describes language A.* ***The class NP*** *is the set of problems that have polynomial time verifiers. The time for M to run is based solely on* $|w|$*, the length of the input string.*

Since a guess to these problems can be verified in polynomial time, they can be decided by a nondeterministic Turing machine in polynomial time. This would be the equivalent of running every possible path of the NDTM simultaneously to decide whether the input is accepted or not. Similar to the definition for class **P**, when $NPTIME(n^k)$ refers to every problem verifiable in time n^k and so decidable by an NDTM in n^k steps;

Definition 3.2.3 (Class **NP** definition II)**.**

$$\boldsymbol{NP} = \bigcup_k NPTIME(n^k).$$

An example of a problem in **NP** is presented in (Sipser 1996, p.296). Algorithms are useful in every field of mathematics to solve problems. This problem is a graph theory problem that, just like a Sodoku, we can check by an algorithm but as of yet there is no efficient algorithm to compute a solution.[7]

[7]For more examples of problems in **NP**, see (Arora & Barak 2009, p.40).

Example 3.2.4. *Here, we look at the problem of finding whether a given graph contains a* ***clique****. First, we must define a clique: A clique is a subgraph*[8] *of an undirected graph in which every two distinct nodes are connected by an edge.* k***-clique*** *denotes a clique with* k *nodes (vertices).*
If we want to find out if a certain graph has a k*-clique, there is no efficient algorithm for this but you can efficiently check whether a given solution is in the set of* k*-cliques for that graph (which could be the empty set with no* k*-cliques for given* k*).*
The problem we are asking, denoted $CLIQUE$ *for inputs* G *and* k *which denote a given graph and the number of nodes in the clique, is therefore:*

$$CLIQUE = \{\langle G, k\rangle : G \text{ is an undirected graph with a } k\text{-clique}\}.$$

First, a machine that describes a TM as in 3.2.2; a DTM, denoted V *which* ***verifies*** *a given input string that* ***could*** *be a* k*-clique, denoted* c*.*

$V =$ *"on input* $\langle\langle G, k\rangle, c\rangle$ *:*

1. *Test whether* c *is a subgraph with* k *nodes in* G*.*

2. *Test whether* G *contains all edges connecting nodes in* c*.*

3. *If both pass,* ***accept****; otherwise,* ***reject****."*

Now, a machine like the one described in 3.2.3; an NDTM, denoted N*, which* ***decides*** $CLIQUE$ *in nondeterministic polynomial time by simultaneously checking* ***all*** *possible outcomes. This has been changed from the example machine in the text to match my interpretation of nondeterministic Turing machines.*

$N =$ *"on input* $\langle G, k\rangle$ *:*

1. *Nondeterministically select all subsets* c *of* k *nodes of* G*.*

2. *Test whether* G *contains all edges connecting nodes in each* c *simultaneously.*

3. *. If any pass,* ***accept****; otherwise,* ***reject****."*

Theorem 3.2.5. $CLIQUE$ *is in* ***NP****.*

Proof. See Turing machines, described above, in 3.2.4. □

3.3 Polynomial-time reducibility and NP-completeness

NP-complete problems are the foundations to one of the greatest unsolved problems in mathematics presently, which will be discussed later, does **P** = **NP**? In order to define **NP**-completeness, we must define what is meant by **polynomial time reducibility**.
Suppose we have an algorithm A_1 that we know computes the language L_1. We also have another language L_2 we would like to compute. If L_1 can be solved in polynomial time then so can L_2 if we can reduce it to L_1. This motivates the definition for a **polynomial-time reducibile** function, from Sipser (1996):

Definition 3.3.1 (polynomial-time reduction)**.** *A language* L_2 *is* ***polynomial-time reducible*** *to language* L_1*, written* $L_2 \leqslant {}_P L_1$*, if a polynomial time computable function* $f : \Sigma* \to \Sigma*$ *exists, where for every* w*,*

$$w \in L_2 \iff f(w) \in L_1.$$

The function f *is called the* ***polynomial time reduction*** *of* L_2 *to* L_1*.*

[8] A subgraph, S, is a graph that is contained within another graph where all the nodes and edges in S are also in G, so $S \subseteq G$.

Since complexity classes are sets of languages, they have the same properties of all other sets, such as sets intersecting and the existence of subsets. A subset of the class **NP** is the class of all languages which are **NP**-complete. These problems are "at least as hard"(Arora & Barak 2009, p.43) as any other problem in the class **NP**. Let x be a language in **NP**-complete. This means that every language in **NP** has a polynomial time reduction to x, and so every problem in **NP** takes at most as much time as x to compute. To note, if every language in **NP** has a polynomial reduction to a language y, this does not mean it is in **NP**-complete.

Definition 3.3.2 (**NP**-complete)**.** *A language x is **NP-complete** if $x \in$ **NP** and if $z \in$ **NP**, then there exists a function from z to x that is a polynomial time reduction.*

3.4 P=NP

Since **NP**-complete problems are *as difficult* as any other in **NP** and thus the *hardest* in **NP**, if we find that these have an efficient algorithm then so do all problems in **NP**; we can deduce this since every problem in **NP** can be transformed into a problem in **NP**-complete. So, we know **NP**-complete $\subseteq$ **NP**, but where does **P** lie?

Theorem 3.4.1. ***P*** $\subseteq$ ***NP***

Proof. Let language L be in the class **P**. Then, by definition, there is a deterministic Turing Machine, M, that computes L in polynomial time. Hence, we can verify a solution in polynomial time by simply running M and checking the solutions given by the machine and the known solution match. This could clearly be simulated by another Turing Machine, N, which is identical to M but has an extra step to compare the solution and then accept it and halt. So N would verify L, which is in **P**, in polynomial time and so L is also in **NP**. Thus, every L in **P** is also in **NP**. □

It is clear that all problems that can be solved in polynomial time can be verified in polynomial time; but is every problem we do not have an efficient solution to intrinsically harder than the problems in **P**, or have we just not managed to find efficient solutions to these problems? This is the unsolved problem of whether **P=NP** or not. If they are equal, this would mean that all the problems we have struggled to solve, a computer would be able to solve as easily as it can sort a list. This would advance all mathematics and science vastly overnight. As said in (Arora & Barak 2009, p.58):

> If **P=NP**...then the world would be mostly a computational Utopia.

Since problems in **NP**-complete are as hard to solve as any other in **NP**, if we find a polynomial reduction of just one problem in **NP**-complete to a problem in **P**, then all the problems in **NP**-complete and thus **NP** must be only as hard as any in **P**, which would imply **NP**$\subseteq$**P**. With this, and 3.4.1, we could deduce that **P=NP**. However, at present no reductions have been found and no counterproof to **P=NP** has been found and so the question remains unsolved.

4 Number Theory

Thus far, we have only seen applications of computational complexity for which we *want* efficient computations. In some cases, we may need our problem **not** to be solved and so a hard computation would be very useful. One application of this is in cryptography, where we want to send secret messages that cannot be read by anyone they weren't intended for.

To understand these new methods of cryptography, introduced in the 1970s (A Mollin 2006, p.157), and why they are currently secure, we must first understand the key mathematical concepts that underlie them. Many of these lie in number theory.

4.1 Modular arithmetic

Congruences are a way in which we can denote the remainder of an integer, a, when divided by an integer, n. Gauss wanted a way in which instead of writing $a = b + nk$, where b is the remainder and k is the integer value of whole ns that go into a, he defined the congruence, as stated in (Sipser 1996, p.17) as:

Definition 4.1.1 (Congruence). *If $n \in N$, then we say that a is **congruent** to b modulo n if $n|(a - b)$, denoted by*

$$a \equiv b \mod n.$$

Clearly, if a has remainder b on division by n then n divides $(a - b)$. For example, since 7|(65-2), $65 \equiv 2 \mod 7$.

4.2 Prime Numbers

Prime numbers underpin some of the methods widely used for cryptography and understanding them and some theorems surrounding them is important in understanding why these methods of encryption work.
First, we must formally define what a prime number is.

Definition 4.2.1 (Prime number). *A **prime number** is a natural number bigger than 1, that is not divisible by any natural number except itself and 1, (A Mollin 2006, p.6).*
*A number that is natural and **not** prime is called **composite**.*

Theorem 4.2.2 (The Fundamental Theorem of Arithmetic). *Let n be an integer such that $n > 1$. Then n has a factorisation into a product of prime powers (existence). Moreover, if $n = \prod_{i=1}^{r} p_i = \prod_{i=1}^{s} q_i$, where p_i and q_i are sets consisting of only primes. Then $r = s$, and the primes p_i and q_i are the same if their order is ignored (uniqueness), (loosely taken from (A Mollin 2006, p.9)).*

Proof. First, we must prove **existence**. To do this, we need to show that every integer n such that $n > 1$ can be written as a prime factorisation. Towards a contradiction, assume there is at least one natural number greater than one that does **not** have a prime factorisation. By the well ordering principle[9], this means that there must be a smallest natural number with this property, name this x. This x can be either prime or not prime:

- If x is prime: since any prime number is a trivial prime factorisation, namely itself, x would have a prime factorisation, which is a contradiction, and therefore x cannot be prime.
- If x is not prime, so x is composite: Let x be composite such that $x = yz$, where y and z are distinct integers greater than 1. Clearly, they must both be smaller than x to multiply to give x. Since x is the smallest natural number that cannot be written as a prime factorisation, y has a prime factorisation, as does z. Let $\prod_{i=1}^{r} p_i$ and $\prod_{i=1}^{s} q_i$ be

[9]The well ordering principle is a property of the natural numbers; Every nonempty subset of the positive integers has a least element, as found on *The Well-ordering Principle* (n.d.) This is because there is a starting point for the natural numbers, namely 1. Now, if this doesn't satisfy the property, then there is still a next smallest, namely 2. This continues until an integer satisfying the property is found, thus there must be a smallest. This can be formally proved by induction

the prime factorisations of y and z respectively. Then, $x = \prod_{i=1}^{r} p_i \cdot \prod_{i=1}^{s} q_i$ and so $x = \prod_{i=1}^{r+s} t_i$, where $t_i = q_i \cup p_i$. And so x is a product of primes, which is a contradiction.

Therefore, by contradiction, no such integer greater than 1 that does not have a prime factorisation exists. Therefore, every integer n such that $n > 1$ can be written as a prime factorisation, as required.
Now, we prove **uniqueness**. To do this, we show that if $n = \prod_{i=1}^{r} p_i = \prod_{i=1}^{s} q_i$, then every p_i is equal to some q_i. Towards a contradiction, assume that there exists at least one integer greater than 1 such that it does not have a unique prime factorisation. Then, by the well ordering principle, there must be a smallest number that satisfies this property, let this be n. So, $n = p_1 p_2 ... p_r = q_1 q_2 ... q_s$. Since these prime factorisations are unique, we know that $p_1 | q_1 q_2 ... q_s$. Now, by Euclid's lemma[10], we know that $p_1 | q_i$ for some q_i. Without loss of generality we can let this q_i be q_1 since order does not matter and since q_1 and p_1 are prime, $p_1 = q_1$. Hence, $\frac{n}{p_1} = p_2 ... p_r = q_2 ... q_s$, which too are distinct. $p_1 | n$ since it is a factor and so $\frac{n}{p_1}$ is an integer smaller than n with distinct prime factorisations, which is a contradiction since n is the smallest natural number to have a unique prime factorisation. Thus, by contradiction, there exists no such natural number to have a unique prime factorisation, and so every prime factorisation is unique, as required. □

Now, we prove Fermat's Little Theorem. This fundamental theorem of number theory has important applications for us in primality testing, which will be looked at later.

Theorem 4.2.3 (Fermat's Little Theorem)**.** *If $a \in \mathbb{Z}$ and p is a prime such that $\gcd(a, p) = 1$, then*

$$a^{p-1} \equiv 1 \bmod p.$$

Proof. To prove Fermat's Little Theorm, we will first prove that $a^p \equiv a \bmod p$, then dividing both sides by a gives the required result. To prove this, we use mathematical induction on a.
First, we fix the prime number p.
We do the initial case: Letting $a = 1$, $1^p \equiv 1 \bmod p$ is clearly true.
Now, we can assume that $a^p \equiv a \bmod p$ is true for *some* a and we see what happens for $a+1$.
By the binomial theorem,

$$(a+1)^p = a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^2 + ... + \binom{p}{p-1} a + 1.$$

Now we can take $\bmod p$ of both sides for the required result:

$$\begin{aligned}
(a+1)^p \bmod p &\equiv [a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^2 + ... + \binom{p}{p-1} a + 1] \bmod p \\
&\equiv a^p + 1 + \binom{p}{1} a^{p-1} + \binom{p}{2} a^2 + ... + \binom{p}{p-1} a \bmod p \\
&\equiv a^p + 1 \bmod p
\end{aligned}$$

(since $\binom{p}{k}$ is always divisible by p so we know all terms except $a^p + 1$ have remainder 0 on division by p)

$$\equiv a + 1 \bmod p$$

by the inductive step $a^p = a \bmod p$ so a^p has remainder a on division by p

$$\equiv [a+1] \bmod p$$

[10]Euclid's lemma is as follows: If a prime p divides ab, a and b are integers, then p divides a or b. Originally written and proven by Euclid in book 7 of "The Thirteen Books of the Elements":"If c, a prime number, measures ab, c will measure either a or b". Measure here means divides.

Thus, by mathematical induction, $a^p \equiv a \bmod p$ for all $a \in \mathbb{N}$ and so as said, this proves $a^{p-1} \equiv 1 \bmod p$, (*Fermat's Little Theorem* n.d.). □

4.3 Primality testing

In RSA encryption, the prime numbers we will have to use will be large in order that the time taken for factorizing $n = pq$ is large enough. In Rivest et al. (1978), they recommend "using 100-digit (decimal) prime numbers p and q". There is currently no perfect algorithm for testing whether a number is prime. However, there are many tests which give us primes with a certain level of accuracy.
Fermat's little theorem is the principle behind these tests. If p is prime then $a^{p-1} \equiv 1 \bmod p$ also implies the contrapositive statement

$$\text{if } a^{p-1} \not\equiv 1 \bmod p, \text{ then p is } \textbf{not} \text{ prime.}$$

So, we can determine if any number is not prime. Unfortunately, the converse of Fermat's little theorem is not true so we cannot test if a number is prime, only if it isn't.

Example 4.3.1. *If we let $n = 377$, as found in (A Mollin 2006, p.198), and let $a = 2$. Then $2^{376} \equiv 94 \not\equiv 1 \bmod 377$. Therefore, we can assume that 377 is not prime.*

Now we will look at the **Miller-Rabin test**. It gives us a good indication as to whether an input is prime. If the input is not prime, the algorithm will always output "composite", but if it is prime, there is a chance the algorithm will mistakenly label the number as "composite". The algorithm is as follows, and can be found in (A Mollin 2002, p.87):
We want to test whether n is prime or not. Let $n - 1 = 2^t m$, so m is odd and $t \in \mathbb{N}$.

1. Pick a random integer a such that $2 \leq a \leq n - 2$.

2. Compute $x \equiv a^m \bmod n$.
   If $x \equiv \pm 1 \bmod n$, then algorithm halts and outputs "probably prime".
   If $t = 1$, then the algorithm halts and outputs "definitely not prime".
   Otherwise let $j = 1$ and move to (3).

3. Compute $x \equiv a^{2^j m} \bmod n$.
   If $x \equiv 1 \bmod n$, then the algorithm halts and outputs "definitely not prime".
   If $x \equiv -1 \bmod n$, then the algorithm halts and outputs "probably prime".
   Otherwise let $j = j + 1$ and move to (4).

4. If $j = t$–1, go to (5), otherwise, go to (3)

5. Compute $x \equiv a^{2^{t-1} m} \bmod n$.
   If $x \not\equiv -1 \bmod n$, then the algorithm stops and outputs "definitely not prime".
   If $x \equiv -1 \bmod n$, then the algorithm halts and outputs "probably prime".

4.4 Euclidean Algorithm

The Euclidean algorithm is important to understand a later application of complexity theory, RSA encryption. First, we state a lemma that helps us prove the Euclidean algorithm, as found in (A Mollin 2006, p.2).

Lemma 4.4.1 (The Division Lemma)**.** *If $a \in \mathbb{N}$ and $b \in \mathbb{Z}$, then there exist unique integers $q, r \in \mathbb{Z}$ with $0 \leqslant r < a$, and $b = aq + r$.*

This is reasonable to assume from 4.1, where Gauss wrote remainders as congruences instead of being in the form $a = b + nk$. We see how this and $b = aq + r$ are of the same form and are closely linked. The Euclidean algorithm can now be proved.

Theorem 4.4.2 (The Euclidean Algorithm)**.** *Let $a, b \in \mathbb{Z}(a \geq b > 0)$, and set $a = r_{-1}, b = r_0$. By repeatedly applying the Division Lemma, we get $r_{j-1} = r_j q_{j+1} + r_{j+1}$ with $0 < r_{j+1} < rj$ for all $0 \leqslant j < n$, where n is the least nonnegative number such that $r_{n+1} = 0$, in which case $\gcd(a, b) = r_n$, (A Mollin 2006, p.3).*

Proof. Here, we see that the sequence $\{r_k\}$ produced by the repeated Division Lemma is decreasing and since all r_k are nonnegative integers, this sequence is clearly bounded, meaning every $\{r_k\} \geq 0$, until the algorithm terminates on $\{r_n\}$(this happens since $\{r_{n+1}\}$ would equal 0). We now make use of the properties $\gcd(x, 0) = x$ and $\gcd(a, b) = \gcd(b, q)$ if $a = br + q$, as stated in *The Euclidean Algorithm* (n.d.).
This implies that $\gcd(a, b) = \gcd(r_i, r_{i+1}) = ... = \gcd(r_n, 0) = r_n$, as required. □

Theorem 4.4.3 (Extended Euclidean Algorithm)**.** *As stated in (A Mollin 2006, p.12): Let $a, b \in \mathbb{N}$, and let q_i for $i \leq n + 1$ be obtained from the Euclidean Algorithm to find g = gcd(a, b), n is also as in 4.4.2. If $s_{-1} = 1, s_0 = 0$, and*

$$s_i = s_{i-2} - q_{n-i+2} s_{i-1},$$

for $i \leq n + 1$, and then

$$g = s_{n+1} a + s_n b.$$

This shows us some of the principles for how our encryption system will work.

5 (Public-Key) Cryptography

Cryptography is no new idea and has only recently used computational complexity to its advantage. People have always wanted to hide secret messages from people they were not written for. In some form, cryptography has been around for thousands of years, with reference to methods used to hide messages going back to ancient times. In The Lives of the Twelve Caesars, Caesar is described as using a way to hide messages from prying eyes:

> ...if there was occasion for secrecy, he wrote in cyphers; that is, he used the alphabet in such a manner, that not a single word could be made out. The way to decipher those epistles[11] was to substitute the fourth for the first letter, as d for a, and so for the other letters respectively.

The way in which Caesar disguised, or *encrypted* or *enciphered*, these messages is a very simple method of *encryption*; the method of disguising a message. There is an original message, which one wants to hide, the *plaintext*, and a disguised message, called the *ciphertext*. Once this ciphertext has been encapsulated and sent it is known as a *cryptogram*. The cryptogram now needs to be *decrypted* or *deciphered* to be understood, which is the transforming of the ciphertext into the plaintext so the original message can be read. These transformations, both encryption and decryption, can be described by one-to-one functions:

[11] An epistle is a letter.

Definition 5.0.1 (Encryption/Decryption function)**.** *An **encryption function** is a one-to-one function*

$$E_e : \mathcal{M} \mapsto \mathcal{C},$$

*where e denotes the enciphering **key** used to encrypt the message, so for every $m \in \mathcal{M}$ (the set of units in the plaintext), $E_e(m) = c$ for some $c \in \mathcal{C}$ (the set of units in the ciphertext).*
*Similarly, a **decryption function** is a one-to-one function*

$$D_d : \mathcal{C} \mapsto \mathcal{M},$$

where d is the decryption key. So, let $E_e(m) = c$, then $D_d(c) = m$.

For example, shifting letters 5 backwards in the alphabet to transform the message would mean that the key is 5, since this is the key to how to encrypt or decrypt the message. Caesar's method is simple since it uses the same method to encrypt a message as it does to decrpyt it. This symbol substitution also requires both sender and receiver both know how to decrypt the message, and so anyone who is trying to read a message not intended for them (known as an 'enemy' *interfering* with the message) may have access to the method used to encrypt the message, since there is no way of sending this secretly.
We now define what a cryptosystem is, as found in (A Mollin 2002, p.6)

Definition 5.0.2 (Cryptosystem)**.** *A cryptosystem or cipher is comprised of a set E_e of enciphering functions and a set $D_d = E_e^{-1}$ of deciphering functions, which corresponds to the former set in such a way that for each e, there exists a unique d such that $D_d(E_e(m)) = m$ for all $m \in M$.*
*Individually e and d are called **keys**, and (e, d) is called a **key pair**. The set of pairs $\in \{(m, E_e(m)) : m \in M\}$ is called a **cipher table**.*

We see here that all decryption functions are inverses of encryption functions, which makes sense since the inverse operation reverses a function, taking you back from the output symbol to the input, and so decrypting the plaintext.

Definition 5.0.3 (Symmetric-Key Cryptosystems)**.** *A cryptosystem is called **symmetric-key** if for each key pair (e, d), the key d is "computationally easy" to determine knowing only e ,and similarly e is easy to determine knowing only d, (A Mollin 2002, p.7).*

The method in which Caesar is described to have used is therefore a symmetric-key cryptosystem (more specifically a **shift cipher**), since it is easy to find out how to reverse the transformation for an enemy in polynomial time by finding the decryption key. Hence, in the case for Caesar, $d = -e = -4$. It is therefore clearly not going to work for encrypting modern-day online banking, for example. For this, we need a type of cryptosystem that won't easily give up the decryption key, even if the encryption key is known to an enemy.

5.1 One-way functions

One concept we can use in order to try and find such a cryptosystem where knowing e doesn't mean you could find out what d is.

Definition 5.1.1 (one-way Function)**.** *A one-to-one function f from a set M to a set C is called **one-way** if $f(m)$ is "easy" to compute for all $m \in M$, but for a randomly selected c in the image of f, finding an $m \in M$ such that $c = f(m)$ is computationally infeasible. In other words, we can easily compute f, but it is computationally infeasible to compute f^{-1}.(A Mollin 2002, p.53)*

Suppose we have a cryptosystem we want to be secure from enemy interference. If we allowed E_e to be a one-way function, then an enemy would not be able to decrypt the message, since the inverse operation of E_e^{-1} is computationally inefficient to solve and thus D_d is also unsolvable in efficient time. However, we also need to be able to decrypt the messages that are sent to us. If we use any one-way function, we most likely won't be able to retrieve data once it's encrypted, rendering it useless. Burning a piece of paper with your message on it is a one-way function, sending everything in the plaintext to an empty symbol but this seems hardly useful when trying to send a secret message to someone. Instead, we need to find specific one-way functions that, when we have the right information, become much simpler to reverse.

Definition 5.1.2 (Trapdoor function)**.** *A **trapdoor function** is a one-way function that also has a **trapdoor**. This means that there exists an additional piece of information which allows us to easily find the solution to $f(m) = c$ where $c \in \operatorname{Im} f$.*

Therefore without this trapdoor, the function would be the same as in 5.1.1 and so it would be infeasible to try and find m from $f(m) = c$. So, a trapdoor function would give us a way to encrypt and decrypt messages, whilst never having to make anything about the decryption key public.

5.2 Public-Key Cryptography

Public-key cryptography, or assymmetric cryptography, is different to symmetric-key cryptography since it only requires the decrption key be kept secret but the encryption key can be made public to anyone. This means that anyone is able to send a cryptogram using the public key, however, only the person who knows the secret decryption key can understand them. It also means that no secret key has ever had to be sent to or from a sender or receiver that could've too be intercepted.
An analogy of this is that **Bob** has a wall safe. He leaves it open, allowing anyone to put a message in and lock the safe. **Alice** puts a message in the safe and then locks it. Now, no one can read Alice's message since it is hidden until Bob unlocks the safe with a combination only he knows. This reveals Alice's original message.

This makes public-key cryptosystems secure, under the assumption that these functions *are* one-way and do not have a polynomial time reduction.
Now we look at one of the oldest and most widely spread public-key cryptosystems used today to keep a lot of data online secure, from bank details to e-mails (A Mollin 2002).

5.3 RSA cryptosystem

The first to publish such a cryptosystem that used a trapdoor function were Ron Rivest, Adi Shamir and Leonard Adleman, hence making the name RSA after their surnames. The principals it is built on, all of which are in number theory, are described in 4. Next, we see how these concepts work together to make the cryptosystem:

First, two prime numbers must be picked, name these p amd q. For a computer these will be very large prime numbers, which we will discuss later. These primes multiplied now make a composite number and let this be $pq = n$. By the Fundamental Theorem of Arithmetic, $p \cdot q$ is the **unique** prime factorisation of n. Now, we need to pick an integer d such that $1 < d < (p-1)(q-1)$ and e is coprime to $(p-1)(q-1)$, i.e. $\gcd(d, (p-1)(q-1) = 1$. This d is to be picked at random by the sender. A unique integer e can then be computed,

using the Euclidean algorithm, to satisfy

$$ed \equiv 1 \bmod (p\text{–}1)(q\text{–}1).$$

Now we define the private and public keys. The **public key**, the key we publish, is (e, n) and the **private key**, the key in which we keep secret, is (d, n). We also must keep p and q secret, and thus $(p\text{–}1)(q\text{–}1)$.

The encrpytion and decryption of this algorithm are simple to compute mathematically. From the original paper, Rivest et al. (1978), where M is the original message and C is the ciphertext, we have:

$$C \equiv E_{(e,n)}(M) \equiv M^e \bmod n$$
$$D_{(d,n)} \equiv C^d \bmod n.$$

Thus, if Alice wants to send Bob, who has his own RSA cryptosystem, a private message, she can:

1. obtain his public key (e, n).
2. Use e and n to hide his original message, M, by calculating $E_{(e,n)}(M) \equiv C$.
3. Now, she can send Bob C via any communication line, since no one but Bob can understand it.

Then, once Bob receives C, he has the information needed to decrypt it, by calculating $D_{(d,n)}(C) \equiv M$, using d, which Bob has kept hidden from anyone else.
Here is a simple example, as found on *RSA Encryption* (n.d.).

Example 5.3.1. *Let the creator choose primes $p = 11$ and $q = 17$. Next, he randomly chooses $e = 3$. Now, he computes $pq = 187$. Then, he must calculate $(p-1)(q-1) = 160$. d can then be calculated since $ed \equiv 1 \bmod (p\text{–}1)(q\text{–}1)$ and we have e, p and q. He finds $d = 107$. He then knows the encrption key, which he publishes. This is (e, n)=(3, 187). The decryption key is (d, n)=(107,187).*

Now, a sender wishes to send the creator the number 72, thus the message m is 72. To encrypt it, the sender calculates the ciphertext, c; $c = m^e \bmod n = 72^3 \bmod 187$. The ciphertext then equals 183. Now that it is encrypted, it has almost been locked and is now sent as a cryptogram. The creator of the system is the only person with the key to unlock it, thus it is secure.

When the creator wants to read the cryptogram, he calculates $c^d = 183^107 = 72 \bmod 187$. As we will prove, this will always work and convert c back into m.

5.4 Mathematics underlying RSA

By 5.3.1, we can see that RSA works, but we need to understand *why* it works and prove that it works. Here, we are asking if $D_{(d,n)}(E_{(e,n)}(M)) = M$ for all M and thus if $(M^e)^d = M \bmod n$ for all M.

Proposition 5.4.1. *For every M, $(M^e)^d = M \bmod n$ when $n = pq$, where p and q are prime and e and d satisfy $ed \equiv 1 \bmod (p\text{–}1)(q\text{–}1)$.*

Proof. Firstly, by the extended Euclidean algorithm and the construction of e, we know that there exists integers

$$\begin{aligned} ed + b(p-1)(q-1) &= \gcd(d, p\text{–}1)(q\text{–}1) \\ &= 1, \end{aligned}$$

and so

$$ed = 1 - b(p-1)(q-1). \tag{5}$$

We can now show that $(M^e)^d = M \bmod p$. Noting that $(M^e)^d = (M^{ed})$ by simple exponent rules, we can simply calculate that

$$\begin{aligned}
(M^e)^d &= M^{1-b(p-1)(q-1)} \\
&= M \cdot M^{(p-1)(b-bq)} \\
&= M \cdot M^{(p-1)^{(b-bq)}} \\
&= M \cdot 1^{(b-bq)} \bmod p \\
&\qquad\qquad \text{By Fermat's little Theorem} \\
&= M \bmod p
\end{aligned}$$

Similarly relabelling p for q, we get $(M^e)^d = M \bmod q$.
From this, we can assume that $(M^e)^d - M$ is divisible by p, since $(M^e)^d$ has remainder M when divided by p. The same can be said for dividing $(M^e)^d - M$ by q. Since p and q are distinct primes, pq must also divide $(M^e)^d - M$, due to its unique prime factorisation containing both p and q.
Hence, $(M^e)^d$ has remainder M on division by $pq = n$, and thus,

$$(M^e)^d = M \bmod n,$$

as required. □

6 Security of RSA

6.1 Factorizing n

The security of RSA encryption is based on the assumption that there is no easy way of factorizing a composite number into its prime factorisation. We cannot prove that RSA encryption is secure; just like any other cryptosystem, they only test of security is seeing if people can break it, (Rivest et al. 1978, p.11). It would seem that the inverse of multiplying prime numbers (factorizing) is more difficult than multiplying the primes. Finding the factors of a natural number n is called the **Integer Factorization Problem (IFP)**. Formally, the problem is:
Given $n \in \mathbb{N}$, find primes p_j for $j = 1, 2, ..., r \in \mathbb{N}$ with $p_1 < p_2 < ... < p_r$ such that $n = \prod_{j=1}^{r} p_j^{e_j}$, (A Mollin 2002, p.93).
This means that we must find the unique prime factorisation of n described in 4.2.2 to solve IFP. Since we can **verify** a factorization in polynomial time, the problem is clearly in **NP**. Therefore, if we found a polynomial time reducible function from IFP to a problem in **P**, we would be able to factorize n efficiently. This would mean that an enemy could work out p and q, and thus could work out $(p-1)(q-1)$, and easily find out d, breaking the whole system and making it easy for interceptors to decrypt messages.
There are indeed methods we can use to factorize composite numbers, but they are inefficient.
One of the oldest methods of factorizing is **trial division**. This involves dividing n by every natural number up to $\sqrt{n}$ (no factors of n can be larger than $\sqrt{n}$. This would appear to have a complexity of $0(\sqrt{n})$ since it checks $\sqrt{n}$ numbers then halts). This is misleading, because the

input on any computer is written in binary. A number x needs to be converted into binary, which will have $\log_2 x$ bits. Name this b. Therefore, if input x takes $\sqrt{x}$ steps, then we would be taking $O(2^{\sqrt{x}}) = O(2^{\frac{b}{2}})$ steps, since $\sqrt{x}$ uses half the bits as x does, since we are working in base 2, (*Big-O: Prime Factors and Pseudo-Polynomial Time* 2019). As the input number gets very large (in (A Mollin 2006, p.207), this is defined as $n > 10^8$), it becomes very inefficient to compute the factors of n with this algorithm.

7 Ethical impacts

If someone were to work out a proof showing that **P=NP**, the world would certainly change almost overnight. Many of the problems we struggle with would become computationally easy, leading to breakthroughs in every area of science. A few examples from (Arora & Barak 2009, p.58) are:

- Engineers would be able to use software that enables them to design new machines to solve problems in jobs.
- Any hypothesis that is physically testable, returning data, could then be run through a machine determining the most likely hypothesis.

These are clearly in **NP** because we can clearly tell if a machine can do a task, or determine the most likely hypothesis by Occam's Razor; a philosophical concept which infers that the easiest solution is the most likely and so the algorithm would be able to pick the hypothesis that fits the data best). To do this, a polynomial time reducible transformation must be found between a problem in **NP** and a problem in **P**.
However, the RSA encryption that handles all our private information, messages and everything else we do online would fall through. Anyone could intercept this information and everything from online banking to Facebook would be unsecure.
A world where RSA encryption is unsecure is however still somewhat inevitable. As quantum computing research advances, quantum computers will be able to use an algorithm, namely Shor's algorithm, to solve these factoring problems much quicker than a modern computer. In Baumhof (2019), a modern day computer is said to take "300 trillion years to break a RSA-2048 bit encryption key", however, a "perfect Quantum Computer could do this in 10 seconds".
However this change occurs, RSA will most likely not be secure in the future, and this is most likely going to be due to advancements in quantum computing.
Most people suspect that **P**$\neq$**NP**, however, if **P** did indeed equal **NP**, science would have some of the biggest breakthroughs to date.

References

A Mollin, R. (2002), *RSA and Public-Key Cryptography*, A CRC Press Company.

A Mollin, R. (2006), *An Introduction to Cryptography - Discrete Mathematics and Its Applications*, second edn, Taylor and Francis.

Alama, J. & Korbmacher, J. (2021), 'The lambda calculus'.
URL: *https://plato.stanford.edu/archives/spr2021/entries/lambda-calculus/*

Arora, S. & Barak, B. (2009), *Computational Complexity: A Modern Approach*, Cambridge University Press, Cambridge.

Baumhof, A. (2019), 'Breaking rsa encryption – an update on the state-of-the-art'.
URL: *https://www.quintessencelabs.com/blog/breaking-rsa-encryption-update-state-art/*

Big-O: Prime Factors and Pseudo-Polynomial Time (2019).
URL: *https://nestedsoftware.com/2018/12/18/big-o-prime-factors-and-pseudo-polynomial-time-55cp.69665.html*

Christ, R. D. & Wernli, R. L. (2014), Chapter 13 - communications, *in* R. D. Christ & R. L. Wernli, eds, 'The ROV Manual (Second Edition)', second edition edn, Butterworth-Heinemann, Oxford, pp. 327–368.
URL: *https://www.sciencedirect.com/science/article/pii/B9780080982885000130*

Church, A. (1936), *An Unsolvable Problem of Elementary Number Theory*, The Johns Hopkins University Press, Baltimore, Maryland.

Copeland, J. (2000), *The Church-Turing Thesis*, Stanford Encyclopedia of Philosophy, Stanford, California.

de Bruijn, N. (2010), *Asymptotic methods in analysis*, dover edition edn, Dover Publications.

Euclid (1908), *The Thirteen Books of the Elements*, Vol. 2, Cambridge University Press.

Fermat's Little Theorem (n.d.).
URL: *https://artofproblemsolving.com/wiki/index.php/Fermat%27s_Little_Theorem*

Immerman, N. (2018), 'Computability and complexity'.
URL: *https://plato.stanford.edu/archives/win2018/entries/computability/*

Morphett, A. (n.d.), 'Turing machine simulator'. The link to the machine I have written in the online website is http://morphett.info/turing/?014fc82d33c1020aeb059202a4d29c67.
URL: *http://morphett.info/turing/turing.html*

Rivest, R., Shamir, A. & Adleman, L. (1978), 'A method for obtaining digital signatures and public-key cryptosystems'.

RSA Encryption (n.d.).
URL: *https://brilliant.org/wiki/rsa-encryption/*

Sedgewick, R. & Wayne, K. (2011), *Algorithms*, Pearson Education.

Sipser, M. (1996), *Introduction to the Theory of Computation*, 1st edn, International Thomson Publishing.

The Euclidean Algorithm (n.d.).
URL: *https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm*

The Well-ordering Principle (n.d.).
URL: *https://brilliant.org/wiki/the-well-ordering-principle/*

Tranquillus, C. S. (1909), *The Lives of the Twelve Caesars*, George Bell and Sons, London.

Turing, A. (1948), *Intelligent Machinery*, Edinburgh University Press, Edinburgh.

Variation of Turing Machine (2019).
URL: *https://www.geeksforgeeks.org/variation-of-turing-machine/*

A Turing Machine Example code

```
; This Turing machine takes an input string and outputs accept if
   there is an even number of 0s
; and outputs reject if there is an off number of 0s.
; Machine starts in state 0 in cell 0.

; State 0: is the first symbol a 0 or 1?
0 0 0 r odd
0 1 1 r even
0 _ _ * accept     ; Empty input, so there was an even number of 0s
   .

; State odd: Determines if there is an odd or even number of 0s up
   to this cell, given there
; was an odd amount up to the last cell.
odd 0 0 r even
odd 1 1 r odd
odd _ _ * reject

; State even: Determines if there is an odd or even number of 0s up
    to this cell, given there
; was an even amount up to the last cell.
even 0 0 r odd
even 1 1 r even
even _ _ * accept

; State accept
accept * * * halt-accept ; There were an even number of 0s in the
   input string.

; State reject
reject * * * halt-reject ; There was an odd number of 0s in the
   input string.
```