Computation

ian.mcloughlin@atu.ie

March 13, 2023

We consider the limits and efficiency of computation. We are largely only interested in syntax.

Languages

Start with any finite set A and call it an alphabet. For example, the set $A = \{0, 1\}$. We define strings of length i over A recursively. The ϵ stands for the empty string[1].

[1] It's notoriously difficult to depict.

$$A^0 = \{\epsilon\}; \quad A^i = \{as \mid a \in A, s \in A^{i-1}\}; \quad i \in \mathbb{N}$$

The Kleene star A^* of the alphabet A is then union of all A^i for $i \in \mathbb{N}_0$. As usual, $\mathbb{N}_0$ is the set of natural numbers including zero[2].

[2] $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$

$$A^* = \bigcup_{i \in \mathbb{N}_0} A^i$$

A language L over an alphabet A is a subset of A^*:$L \subseteq A^*$. A language can be finite like $L_1 = \{00, 11, 000, 111\}$ or infinite like the language L_2 of binary representations[3] of the prime numbers.

[3] We will use the usual representation with the most significant bit at the left and no leading 0's.

$$L_2 = \{10, 11, 101, 111, 1011, \ldots\}$$

Note that the strings in a language all have finite length whether the language is finite or infinite. The length of a string s is denoted $|s|$. For example, $|0011| = 4$ and $|\epsilon| = 0$.

Turing machines

Turing machines encapsulate the concept of computation. Designed to be as simple as possible, they consist of a single read/write head, an infinite tape of cells that can contain symbols from an alphabet, and a table of instructions.

At any point in time, the head is over a single cell of the tape and is in any one of a finite number of states. The cell contains some symbol from an alphabet called the tape alphabet. A special symbol denotes an empty cell, called the blank symbol[4].

[4] You can say the cell is empty in that case, if you want.

Turing machines perform one step at a time, which involves reading the symbol in the current cell, overwriting it with a symbol[5], moving to the cell to the left or right, and changing state[6].

[5] Possibly the same symbol, another way of saying the symbol stays in the cell unchanged.

[6] Again, possibly changing to the same state it is already in – another way of saying it stays in the same state.

Each Turing machine is defined by a set of rules given in a table called a state table. The table tells the machine what to do for the given state it is in and the given symbol in the current cell. Creating these rules is analogous to writing a computer program.

We define a (deterministic[7]) Turing machine M by a 7-tuple M:

[7] All Turing machines will be assumed to be deterministic until specified later.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_f)$$

where

Q is a finite set of states;

Σ is the input alphabet, not containing the blank symbol $\sqcup$;

Γ is the tape alphabet, a superset of Σ containing $\sqcup$;

δ is a map: $\delta : (Q \setminus \{q_a, q_f\}) \times \Gamma \to \Gamma \times \{L, R\} \times Q$;

q_0 is the initial state;

q_a is the accept state, which is a terminal state[8];

q_f is the reject state, also a terminal state.

[8] A terminal state stops the machine – a terminal state has no rules in the state table.

Before the Turing machine begins its operations, a finite number of consecutive non-blank cells contain symbols and the head is over the left-most of these. These non-empty cells form the Turing machine's input. They must form a string over the alphabet Σ. The Turing machine head starts at the left-most cell containing the input. All of the other cells on the tape are blank. The machine can find the end of the input – if it wants to – by moving right until it encounters a blank cell.

When the Turing machine operates on an input, there are three possibilities. The first two are similar: the machine can accept the input by ending in the accept state q_a or it can reject the input by ending in the fail state q_f. In these cases the machine stops upon entering these states, which is often called halting. The third possibility is that the machine might never stop operating.

Note that the set of states and the tape alphabet are both finite, so the number of rules for the machine (the state table) is finite too. Thus for the machine to not halt, it must get caught in some form of infinite loop.

The subset of Σ^* (the Kleene star of the input alphabet) containing all elements that are accepted by the Turing machine M is called the language of the Turing machine $L(M)$:

$$L(M) \subseteq \Sigma^*$$

A Turing machine that halts on all inputs is called a decider and is said to decide its language.

Note it is possible for two different Turing machines M_1 and M_2 to accept or decide the same language: $L(M_1) = L(M_2)$. There is nothing surprising about that. We think of a Turing machine as representing (or being) an algorithm. Lots of common computational problems have several different common algorithms to solve them[9].

[9] For example, to sort a list we can use quick sort, bubble sort, or heap sort, among others.

State tables

A Turing machine can be encapsulated in a five-columned table called a state table. Each row of the table specifies the transition function δ for a given state and tape symbol. Table 1 gives an example of a state table. This Turing machine accepts all strings over the alphabet $\{0,1\}$ that contain an even number of 1's, including zero as an even number.

State	Input	Write	Move	Next
q_0	0	0	R	q_0
q_0	1	1	R	q_1
q_0	_	_	L	q_a
q_1	0	0	R	q_1
q_1	1	1	R	q_0
q_1	_	_	L	q_f

Table 1: Even parity Turing machine.

Provided we adopt some conventions, the state table can be used to describe the whole Turing machine. First, the initial state is the first one listed in the table. Second, the tape alphabet is given by the distinct symbols in the second column. Finally, the only difference between the input and tape alphabets is the blank symbol.

Note that for every combination of non-terminal state tape alphabet, there is exactly one row in the state table. As we have defined them, Turing machines must have this property. Sometimes we break the rule, allowing any number of rows for a given state and tape symbol, even zero. In that case we call the Turing machine non-deterministic. Turing machines with exactly one row for each combination are then called deterministic.

Decision problems

A Turing machine that always halts, irrespective of the input, is called a decider. A decider makes a binary choice for a given input: accept or reject. The Turing machine M in this case performs a map:

$$f : \Sigma^* \to \{q_a, q_f\}$$

Sometimes this map is called an indicator function, as it indicates which elements of Σ^* are in $L(M)$. The Turing machine is an algorithm that performs the map.

A map from the Kleene star A^* of an alphabet A is called a decision problem. So, a decider performs a decision problem. The alphabet in that case is the input alphabet of the Turing machine.

Complexity

A decider takes a finite number of steps before halting on a given input. Consider all the strings in Σ^* of some specified length n. For each of these strings, the decider might take a different number of steps.

We are typically interested in knowing the maximum number of steps $f(n)$ the decider takes on inputs of length n. We use the number of steps as a proxy for time. It is common to call the number of steps the *time* the Turing machine takes.

Digital computers have a similar idea built-in called the clock. There is a circuit in the computer that creates a regular pulse. On each pulse, the other circuits in the computer perform their next action. The transition from one pulse to the next is often called a clock cycle. Usually each clock cycle takes the same amount of time up a given precision, but not always. A clock cycle is analogous to us looking for the next step in a Turing machine's state table.

We can sometimes express the maximum number of steps a Turing machine takes on an input of length n as an expression $f(n)$. For example, the Turing machine in Table 1 reads each symbol of the input in turn and then a single blank at the end before halting. So, it takes $f(n) = n + 1$ steps on every input of length n.

We are interested in knowing how $f(n)$ grows in relation to n. For this, we use big-O notation. A function $f(n)$ is said to be big-O[10] of a function $g(n)$ if there exist two positive integers c and n_0 such that $cg(n) \geq f(n)$ for all $n \geq n_0$.

[10] $f(n)$ is $O(g(n))$ if there exist c and n_0 such that $cg(n) \geq f(n)$ for all $n \geq n_0$.

In the previous example, $f(n) = 2n + 2$ is $O(n)$ which can be seen by setting $g(n) = n$, $c = 3$ and $n_0 = 2$. The functions $f(n)$ and $cg(n) = cn$ are depicted in Figure 1.

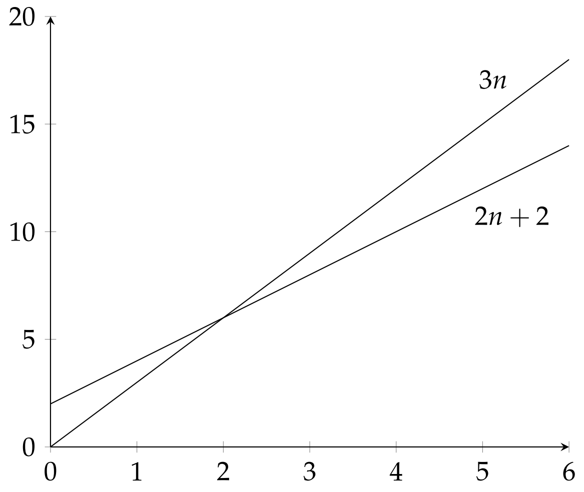

Figure 1: Big-O notation.

Polynomial Time

We define the complexity class[11] TIME$(f(n))$ as the set of all languages for which there is a decider that takes $O(f(n))$ steps. In some sources, notably the Complexity Zoo,[12] TIME is called DTIME. The D stands for deterministic.

[11] A complexity class is a set of languages.

[12] *Complexity Zoo:D - Complexity Zoo*. Dec. 31, 2022. URL: `https://complexityzoo.net/Complexity_Zoo:D#dtime` (visited on 03/09/2023).

Possibly the most famous complexity class is P, the set of languages that are each decidable by some deterministic Turing machine in $O(n^k)$ steps for some $n \in \mathbb{N}_0$. Using the TIME$(f(n))$ definition, P is equivalently defined as the union of TIME(n^k) for all $k \in \mathbb{N}_0$.

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{TIME}(n^k)$$

The P is short for polynomial. We define a polynomial as an expression in a numeric variable n made up of constants, positive integer powers of n, addition, and multiplication.

A key property of a polynomial expression is that the number of additions and multiplications that the expression involves does not depend on n. For example, $n^5 + 2n + 4$ is a polynomial involving five mutliplications and two additions. The expression $2^n + n^2 + 5$ is not a polynomial because the appearance of 2^n means we do not know how many mutliplications we will carry out until we know n.

Non-determinism

We consider now a modification to Turing machines. We will allow a Turing machine to transition to zero or more states simultaneously at each step. That is, in a given state, when reading a given symbol, the machine can follow any number of paths. This requires a modification to the transition function as follows.

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

Here $\mathcal{P}(Q)$ is the powerset of Q. By powerset, we mean the set of all subsets if Q.

How does this affect the state table? Now there may be zero, one, or more rows for each combination of state and tape symbol. In practice, we can view this as the Turing machine being able to fork into two or more branches of computation. Note that every deterministic Turing machine is essentially a non-deterministic Turing machine. You do have to adapt the transition function to map to the set $\{q\}$ rather than the element q, but that's a trivial change.

We say an input is accepted by a non-deterministic Turing machine if any of its branches end in the accept state. It may be that some branches end in the fail state or do not halt. However, we say the machine halts – that is, every branch halts – whenever any branch enters the accept state[13]. The same is not true for the fail state. For an input to be rejected, every branch must enter the fail state. If all branches halt for all inputs, we say the non-deterministic Turing machine decides its language.

[13] In terms of the steps the machine takes, all branches stay in sync. All live branches take a step together, as determined by the state table.

How does non-determinism affect the computations Turing machines can perform? The bad news is that this does not give a Turing machine any extra power. Any language accepted by a non-deterministic Turing machine is also accepted by some deterministic Turing machine. Any language decided by a non-deterministic Turing machine is also decided by some deterministic Turing machine.

Turing proved in his 1936 paper *On computable numbers with an application to the encheidungsproblem* that some languages are not decidable by any Turing machine. There are even languages that cannot be accepted, let alone be decided. Allowing non-determinism does not change the languages that can be accepted or decided.

However, it is not known whether computations can be done more efficiently on non-deterministic Turing machines. Examples abound of polynomial-time non-deterministic Turing machines that accept or decide languages for which no known polynomial-time deterministic Turing machine is known. It is unknown whether such deterministic Turing machines exist. This is the P versus NP problem.

Non-deterministic Polynomial Time

The NP complexity class is the non-deterministic equivalent of the deterministic P complexity class. Define $NTIME(f(n))$ as the set of languages decidable by a non-deterministic Turing machine in $O(f(n))$ steps.

For an accepted input, all branches halt once one of them reaches the accept state. The steps taken by the machine is the number of steps taken in that accepting branch. For a rejected input, all branches must end in the fail states. The number of steps taken is then defined to be the number of steps taken in the longest branch.

Then, NP is the set of all languages decidable by non-deterministic Turing machine in $O(n^k)$ steps for $k \in \mathbb{N}_0$. Given that non-deterministic Turing machines may fork into many paths, how should we count the steps they take? The common way to count this is to consider the longest path taken, irrespective of whether it accepts or rejects. Thus a non-deterministic Turing machine takes $O(f(n))$ steps if the maximum number of steps in its longest path on an input of length n is $O(f(n))$.

Note, because every deterministic Turing machine is also a non-deterministic Turing machine, every language in P is in NP. So, P $\subseteq$ NP. The P versus NP problem is to determine whether P $=$ NP or P $\subset$ NP.

Church-Turing thesis

The ideas surrounding Turing machines arose from Turing's investigations into computability. Alonzo Church was Turing's PhD supervisor and developed an equivalent computational system called lambda calculus. By equivalent, we mean that any computation that can be done on a Turing machine can be done using lambda calculus and vice versa. Lambda calculus is the basis for functional programming languages. The difference between functional and imperative programming languages[14] is mirrored in that between lambda calculus and Turing machines.

[14] Python, C, Java, C++, Perl, Go, Rust are all examples of imperative languages. Imperative languages use statements to manipulate data.

Turing and Church were working on ideas posed by Hilbert and Gödel which concerned the foundations of mathematics. Hilbert asked if mathematics was algorithmic in the sense that true mathematical statements could be deduced by some mechanical procedure from axioms. Gödel had shown that there must be some statements in mathematics that are true but cannot be proved.

Church and Turing proved, while developing their computational systems, that their systems could not compute everything. The question then arises: do Turing machines fully encompass all possible computations? What has become known as the Church-Turing thesis is the idea that what we as humans think of as computation is perfectly encapsulated by Turing machines and so the limits on computation are real — they are not just features of Turing machines. In the next section we show that there is at least one decision problem over $\{0, 1\}$ that no Turing machine can decide.

Decidability

Consider the alphabet $A = \{0, 1\}$ and its Kleene star $A^* = \{\epsilon, 0, 1, 00, \ldots\}$. A language L_1 over A is a subset of A^*: $L_1 \subseteq A^*$. The language L_1 might be recognized by a Turing machine M_a: $L(M_a) = L_1$. A different Turing machine M_b might also recognize L_1: $L(M_a) = L(M_b) = L_1$. There might be more Turing machines that recognize it.

There might also be none. How can we say that with confidence? There are simply too many languages and not enough Turing machines. Note that the set of languages over an alphabet A is infinite. Likewise, the number of Turing machines accepting inputs over A is infinite. So, how can it be that there are more languages than machines?

The answer lies in the notion of correspondence. We will define a correspondence as a bijection between two sets[15]. If there is a bijection between two sets, we will say they have the same size, even when they are infinite.

[15] Sometimes we talk of correspondences related to partial functions, but we will ignore that.

A surprising fact about infinite sets is that some of them correspond to each other and some do not. For instance, the set of natural numbers $\mathbb{N}$ and the set of even natural numbers $2\mathbb{N}$ can be put in a correspondence defined by the following map.

$$f : \mathbb{N} \leftrightarrow 2\mathbb{N} : n \leftrightarrow 2n$$

Each element n of $\mathbb{N}$ is paired with exactly one element of $2\mathbb{N}$. Likewise, each element of $2\mathbb{N}$ is paired with exactly one element of $\mathbb{N}$. So $\mathbb{N}$ and $2\mathbb{N}$ seem to have the same number of elements.

Another set that the same number of elements as $\mathbb{N}$ is the Kleene star of $A = \{0, 1\}$. We need to give a bijection between the sets $\mathbb{N}$ and A^*. Unfortunately, simply reading the elements of A^* as binary numbers does not lead to a bijection. This is because and infinite number of strings in A^* will give the same natural number because of leading zeros. For instance, 10, 010, 0010, and 00010 are all binary representations of the number 2 where the most significant bit us on the left. There are also issues with the number 0. However, if we specify that we prepend a 1 to each element of A^* then we do get one copy of each natural number in binary format.

$$g : \{0,1\}^* \leftrightarrow 1\{0,1\}^* : w \leftrightarrow 1w$$

$$\{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\} \leftrightarrow \{1, 10, 11, 100, 101, 110, 111, \ldots\}$$

Then the correspondence is between the number n written in decimal in $\mathbb{N}$ to the same number $1w$ in $1A^*$ written in binary.

$$f : \mathbb{N} \leftrightarrow \{0,1\}^* : n_{10} \leftrightarrow 1n_2$$

In the above, n_2 means the binary representation of n and n_{10} means the decimal representation. So, A^* is the same size as $\mathbb{N}$.

Now, the set of languages over A is the set of subsets of A^*. The set of all sets of a set S is called the power set of S and is written $\mathcal{P}(S)$. So, the set of all languages over A is the power set of A^*. The mathematician Georg Cantor gave a proof that there can be no correspondence between a set and its power set in 1891. The proof uses what is known as the diagonalization argument.

The proof, adapted for our purposes, is roughly as follows. Suppose f is a correspondence from A^* to its power set $\mathcal{P}(A^*)$. Then $f(w) = \{w_0, w_1, w_2, \ldots\}$ for any w in A^* where w_i is some element of A^*. For example, $f(01)$ might be $\{0,1\}$ and $f(0010)$ might be $\{00, 11, 000, 111, \ldots\}$.

We then use two facts about the natural numbers: they can be listed out, albeit in an infinite fashion, and they can be ordered. We will use the usual way of listing them out in order: $(1, 2, 3, 4, \ldots)$. Under our correspondence, this gives the ordering $(\epsilon, 0, 1, 00, \ldots)$ of A^*.

Then we create the following table. There are an infinite number of rows and an infinite number of columns. The columns are labeled with the elements of A^* according to our ordering of it. Likewise, the rows are labelled with the elements of A^* in the same order. Now, each entry in the table is either 0 or 1. The entry is 1 if the column label is an element of the set in $\mathcal{P}(A^*)$ to which the row label corresponds to. Table 2 depicts the first few entries.

A^*	ϵ	0	1	00	01	11	000	…
ϵ	1	1	0	0	0	0	0	…
0	0	1	1	0	0	1	1	…
1	0	0	0	1	0	0	0	…
00	0	0	1	0	0	1	0	…
01	0	1	1	0	0	1	0	…
11	0	1	0	0	0	1	0	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Table 2: Our supposed correspondence between $\{0,1\}^*$ and $\mathcal{P}(\{0,1\}^*)$.

In the table, no two rows can be the same. The idea of the correspondence is that each element of A^* (the row labels) corresponds to a unique subset of A^*. The rows depict what are called indicator functions. Suppose w is a row label. The 1's indicate which elements of A^* are in the set corresponding to w. Essentially each row depicts a decision problem.

We have assumed that a correspondence f exists. Now we can show that there is a problem with that assumption – we can show that it leads to a contradiction. The correspondence is supposed to map a unique element of A^* to each element of $\mathcal{P}(A^*)$. However, we can find a set D in $\mathcal{P}(A^*)$ that cannot be mapped to from anything in A^* in the correspondence.

That subset D of A^* is as follows. The idea is to look at each of the entries in the main diagonal of the table above. If it is a 0, take the column label as an element of D. If it is a 1, do not include the column label in D. This ensures that D is different from the subset of A^* indicated by row i of the table in the inclusion of the i^{th} column label. If the element is in the subset indicated by row i, it is not included in D. If it is not in the subset, it is included in D.

So, D is different from the subset indicated by every row of the table. However, it is a subset of A^*. We have found a subset of A^* that is not involved in the correspondence. This contradicts our assumption that f is a correspondence. We conclude that no such correspondence can exist. That is, there is at least one more element, in some sense, in the power set than in the original set. Note that we depicted a specific supposed f in Table 2, but this argument works no matter what the entries in the table are.

So, there are more languages over A than there are strings over A. How does this show that some languages are not decided by any Turing machine. The key is to show that every Turing machine can systematically be converted into a string over A.

Turing gives a comprehensive encoding of Turing machines as strings in his paper. We will just note that the state tables in this document are all contained in computer files. Computer files are strings of 0's and 1's. Thus, every state table that we can depict is a string of 0's and 1's. Therefore, there at most as many Turing machines as there are strings over $\{0, 1\}$. So, at least one language over $\{0, 1\}$ cannot be decided by any Turing machine. The decision problem it represents is undecidable.

Reductions

Ultimately the output of a Turing machine is a binary accept/reject decision. However, Turing machines also have another type of output: what is left on the tape once the machine halts. We might care about this, for instance, if we are interested in running two Turing machines sequentially on the one initial input. The first Turing machine might manipulate the input in order that the second Turing machine receives that input altered in some way.

As an example, consider the language EVENPARITY containing all strings over the alphabet $A = \{0, 1\}$ that contain an even number of 1's, including zero as an even number. Thus, EVENPARITY $= \{\epsilon, 0, 00, 11, 110, 101, 011, \ldots\}$. A simple Turing machine decides this language, as per Table 1.

We can use this Turing machine to decide the language ODDPARITY, containing all strings over A that contain an odd number of 1's, provided we use the machine given in Table 3 on the input first.

In state q_0, this machine scans from left to right across the input and puts a 1 in the blank cell at the end. Then it moves to state q_1

State	Input	Write	Move	Next
q_0	0	0	R	q_0
q_0	1	1	R	q_0
q_0	_	1	L	q_1
q_1	0	0	L	q_1
q_1	1	1	L	q_1
q_1	_	_	R	q_a

Table 3: Append a 1 Turing machine.

where it moves back from right to left, until it overshoots the original starting point by one cell and then moves right one cell to that exact point. Note the machine always accepts, no matter the input. The net effect is that the head is back in the same starting point on the tape and a 1 has been appended to the original input on the tape. Giving this new input to the Turing machine in Table 1 will lead to an accept if the original input contained an odd number of ones and reject otherwise. We call this a reduction from ODDPARITY to EVENPARITY, denoted as ODDPARITY $\leq$ EVENPARITY

How many steps $f(n)$ does the Turing machine in Table 3 take on an input of length n? It scans across the input, then reads the blank at the end and moves left, back to the last symbol of the original input. That takes $n+1$ steps. It then scans to the start of the input and reads the blank just beyond that before accepting, taking another $n+1$ steps. It total, that's $f(n) = 2n+2$ steps — a polynomial.

What we have shown is that the ODDPARITY decision problem can be converted into EVENPARITY decision in polynomial time. By converted into, we mean that anything in ODDPARITY is converted into something in EVENPARITY and anything not in ODDPARITY is converted into something not in EVENPARITY so that the decider for EVENPARITY can decide ODDPARITY. The fact that the conversion happens in polynomial time is significant, as the sum of two polynomials is a polynomial. Thus if EVENPARITY is in P then so is ODDPARITY. This essentially means that ODDPARITY is no more difficult to decide than EVENPARITY.

Note a quirk in this situation: EVENPARITY can be reduced to ODDPARITY using the same set up. The two problems are considered equivalent. This is not always the case — sometimes we can only reduce in one direction.

COMPLETE

In 1971, Stephen Cook proved[16] an interesting result: there is a language to which all of the languages in NP can be reduced in polynomial time. Since then, a number of languages have been shown to have this property. They are called NP-HARD because they are at least as hard as each of the languages in NP. The language that Cook used is itself in NP. Languages that are both NP-HARD and in NP themselves are called NP-COMPLETE.

[16] Stephen A. Cook. "The Complexity of Theorem-proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: http://doi.acm.org/10.1145/800157.805047.

The specific language that Cook demonstrated was NP-COMPLETE is called SAT, which is short for *Boolean satisfiability problem*. The language is made up of well-formed Boolean formulas. These are made up of a finite number of Boolean variables that can take either of the values TRUE or FALSE, the two binary operators AND and OR, and the unary operator NOT. The AND operator is denoted $\wedge$ where, for two Boolean variables A and B, $A \wedge B$ is TRUE if and only if both A and B are. The OR operator is denoted $\vee$ where $A \vee B$ is TRUE if any of A and B are, including if both are. Finally, NOT is denoted $\neg$ and $\neg A$ is TRUE when A is FALSE and vice versa. We can define well-formed formulas by saying that variables and the above three operators as described are well-formed formulas, and then allowing the substitution of well-formed formulas for the A's and B's in the above using brackets as necessary to show precedence. A formula is said to be satisfiable if there is some setting of the variables within the formula that makes it TRUE. The language SAT is then the set of all satisfiable well-formed formulas[17].

[17] There is a slight technicality here in that the number of variables in the language is possibly infinite. Alphabets must be finite, so the variables themselves cannot be in the alphabet. We can get around this by encoding the variables using a finite alphabet.