CSC 448 Lecture Notes

William He

Contents

1 Preliminaries

This course is about what can be computed, examples of computational problems are

- Suppose a function $f : D \mapsto R$, given $x \in D$, how do we find $f(x)$.
- Given $y \in R$, find $x \in D$ s.t. $f(x) = y$.
  Note that this is not a well-defined function. Hence, an alternative approach is:
- Given $g : R \mapsto P(D)$, and $y \in R$, find $f^{-1}(y)$.
- Search Problem: given the relation $S \subseteq D \times R$, given $x \in D$, find any $s \in R$ such that $(x, y) \in S$.
- Randomisation: Some distribution μ on a set S, we want a sample from μ

Notation 1.0.1. *Throughout the course, we define*

- $\mathbb{N} = \{0, 1, 2, \dots\}$
- Σ *is the finite set of letters, e.g.* $\{0\}, \{0, 1\}$
- $\Sigma^* = \{\epsilon\} \cup \Sigma \cup \Sigma^2 \dots$ *is the set of all strings made out of letters from* Σ.

Definition 1.0.2. *Given two sets A, B, they are said to have the same cardinality if $\exists f : A \mapsto B$ that is a bijection (one-to-one + onto)*

A is $\leq B$ in size if there is an one-to-one map $f : A \mapsto B$.

Theorem 1.0.3 (Cantor Bernstein)**.** *If $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$.*

Proposition 1.0.4.

- $|\mathbb{N}| = |$*even numbers*$|$
- $|\mathbb{N}| = |\Sigma^*|$ *(by seeing the bijection)*
- $|\mathbb{N}| = |\mathbb{Q}|$
- $|\mathbb{N}| = |\mathbb{N}^2|$

Theorem 1.0.5 (Cantor)**.** $|\mathbb{N}| < |\mathbb{R}|$. *Specifically* $|\mathbb{N}| < |(0, 1)|$.

Proof. We show $|\mathbb{N}| \ngeq |\mathbb{R}|$ (there is no surjective function that maps from $\mathbb{N}$ to $\mathbb{R}$). In fact, we can show there is no surjective function that maps from $\mathbb{N}$ to $(0, 1)$.

Suppose there is an onto map from $\mathbb{N}$ to $(0, 1)$ say f.

Find x such that jth digit of x is different than jth digit of $g(j)$. Then, if $g(i) = x$, then ith digit of x is different than ith digit of $g(i) = x$. $\square$

Theorem 1.0.6 (Cantor)**.** *For all sets A, $|A| < |P(A)|$.*

Proof. Again we show $|A| \ngeq |P(A)|$.

If there exists a surjective function $f : A \mapsto P(A)$, consider

$$B = \{x \in A : x \notin f(x)\}$$

If $f(b) = B$ then either $b \in B \to b \notin f(b) = B$ or $b \notin B \to b \notin f(b) \to b \in B$. $\square$

Definition 1.0.7. *A set is countable if there is an injection from this set to* $\mathbb{N}$.

A set is uncountable if it is not countable.

Proposition 1.0.8. *The following sets are countable:*

$\mathbb{N}$, *any finite set,* $\mathbb{Z}, \mathbb{Q}, \mathbb{N} \times \mathbb{N}, \mathbb{N}^*$, *the set of all functions map from* $\{1, 2\}$ *to* $\mathbb{N}$.

The following sets are not countable:

$\mathbb{R}, P(\mathbb{N})$, *the set of all functions map from* $\mathbb{N}$ *to* $\{1, 2\}$, *the set of all functions map from* $\mathbb{N}$ *to* $\mathbb{N}$.

In general, the set of all functions map from A to B is equivalent to B^A.

Notation 1.0.9. *We define $|\mathbb{N}| = \aleph_0$ and $\mathbb{R} = |P(\mathbb{N})| = c$(continuum)*

Theorem 1.0.10 (Continuum Hypothesis, Cantor)**.** *All sets either have size $\leq \aleph_0$ or $\geq c$.*

Definition 1.0.11. *An alphabet is a finite set Σ, where elements are called letters/symbols.*

A word/string over an alphabet Σ is a finite sequence over Σ. The empty word is denoted ϵ.

A language over Σ is a subset L of Σ^*.

For a language L, the decision problem for L is the computational problem, given $x \in \Sigma^*$, determine whether $x \in L$.

Theorem 1.0.12. *There exists some language L whose decision problem cannot be solved by a C-program.*

Proof. Each C-program solves the decision problem of at most 1 language.

$|P(\Sigma^*)| = |\text{set of languages}| > |\Sigma^*| > |\text{set of C programs}|$ □

Definition 1.0.13. *A register machine is a program that has access to x (a changable number) registers.*

- *Each register holds an element of* $\mathbb{N}$.
- *We can increment, decrement, conditional jump on* 0*, halt and accept, halt and reject.*
- *Input goes in* $R0$*, with all other registers set to* 0.
- *Output accept/reject/not halt.*

Note that by constructing a register machine, it specifies a function $f : \mathbb{N} \mapsto \{\textit{Accept, Reject, Not Halt}\}$ *where the input is the input to the register* $R0$*. Alternatively, we can write* $\mathbb{N}$ *as* $\{0,1\}^*$*, the binary encoding.*

2 Finite Automata

2.1 DFAs

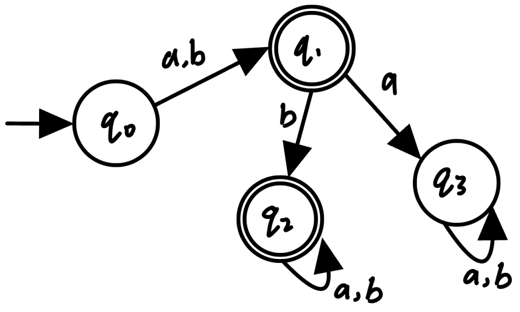
Definition 2.1.1 (Deterministic Finite Automata)**.** *It is a one-pass computer with fixed finite memory. Given by*

- Q*: set of finite states*
- $q_0 \in Q$*: start state*
- $F \subseteq Q$*: Accepting final states*
- *A transition function* $\delta : Q \times \Sigma \mapsto Q$

It operates on a string $a_1 a_2 \ldots a_n$ *by*

1. *Initiate* q_0
2. *For* $i = 1$ *to* n*:*
   - $q_i = \delta(q_{i-1}, a_i)$
3. *If* $q_n \in F$*, accept. Otherwise, reject.*

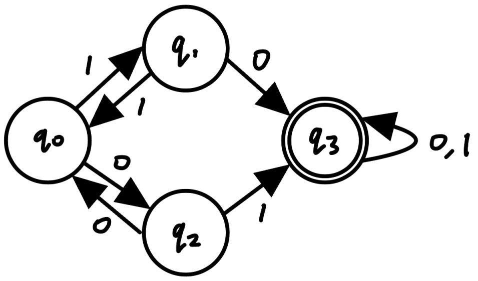
Example 2.1.2. *The following example is a DFA*



there are four states, q_1, q_2 are accepting. There are also arrows between states. The transition q_1 to q_2 through the arrow b means that means that we were in state q_1, and the next character of the input is b, then we move to q_2.

One fixed DFA A specifies a language, denoted as $L(A) = \{x \in \Sigma^* : A \text{ on input } x \text{ accepts}\}$

Example 2.1.3. *We have the following DFA A*



On input 1011011*, the state goes as $q_0 q_2 q_3 \dots q_3$, we accept.*

On input 110110*, $q_0 q_2 q_0 q_1 q_3 \dots q_3$, we accept.*

$L(A) = (00 \cup 11)^*(01 \cup 10)(0 \cup 1)^*$

Notation 2.1.4 (Operations on languages)**.** *Let $L, L_1, L_2 \subseteq \Sigma^*$, we have the following operations on languages*

- *Union:* $L_1 \cup L_2$
- *Intersection:* $L_1 \cap L_2$
- *Complement:* L^c
- *Concatenation:* $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$
- $L^* = \{\epsilon\} \cup L \cup L \cdot L \cup L \cdot L \cdot L \dots$

Example 2.1.5. 1. $\{0, 11, \epsilon\} \cdot \{1, 00, \epsilon\} = \{01, 000, 0, 111, 1100, 11, 1, 00, \epsilon\}$

2. $\{0, 11, \epsilon\}^* = \{\epsilon\} \cup \{0, 11, \epsilon\} \cup \{00, 011, 0, 110, 1111, 0, 11, \epsilon\} \dots$

3. $L = \emptyset$ *then* $L^* = \{\epsilon\}$

4. $L = \{\epsilon\}$ *then* $L^* = \{\epsilon\}$

5. $L^* \cdot L^* = L^*$

6. $(L_1 \cdot L_2)^* \neq L_1^* \cdot L_2^*$

Definition 2.1.6. *A language L is regular if there exists some DFA A with $L = L(A)$.*

Theorem 2.1.7. *If $L, L_1, L_2 \subseteq \Sigma^*$ are regular languages, then so are all the languages obtained by the operations on Notation 2.1.4.*

Proof. L^c: Swap the accepting states and the non-accepting states.

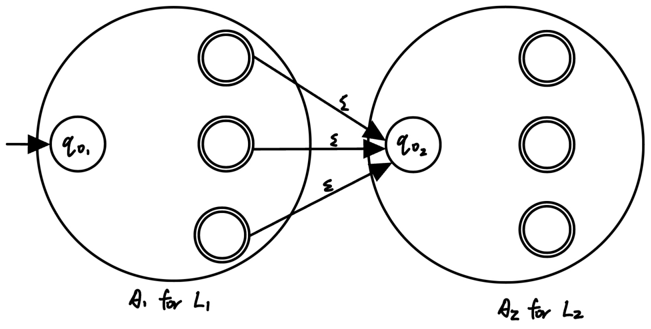
$L_1 \cup L_2$: Construct the "product automata", where $Q = Q_1 \times Q_2$ with $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$. The DFA starts at $q_0 = (q_{0_1}, q_{0_2})$. And the accepting states are all the (q_1, q_2) if $q_1 \in F$ or $q_2 \in F$.

$L_1 \cap L_2$: Same as $L_1 \cup L_2$ except the accept state becomes "and".

$L_1 \cdot L_2$: Note that $z = z_1 \dots z_n \in L_1 \cdot L_2 \iff \exists i$ such that $z_1 \dots z_i \in L_1$ and $z_{i+1} \dots z_n \in L_2$.

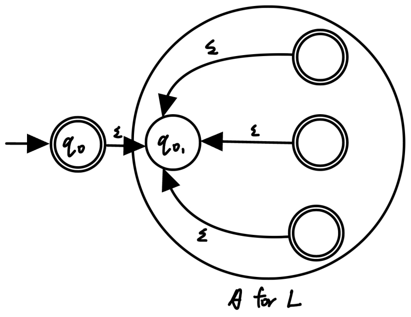
To do this, since L_1, L_2 are regular languages, there exists some DFAs A_1, A_2 such that $L(A_1) = L_1, L(A_2) = L_2$.

we construct an NFA for $L_1 \cdot L_2$, with ϵ-transition between them as follows



Then, by Theorem 2.2.4, this is regular.

L^*: Suppose A is the DFA for L.



We make an NFA as follows, we construct ϵ-transition from accept states to the start state. Then, we construct a new start state with an ϵ-transition to the original old start state. We also make the new start state an accept state, this is to make sure we accept ϵ. □

One thing to note is that DFA have limited power, it cannot decide/recognise many "simple" languages.

2.2 NFAs

Definition 2.2.1. *An NFA is an automata with a multiple transition options at any given state.*

A string is accepted by the NFA if there is some way of taking the options that leads to an accept state.

Formally, an NFA consists of

- *Q: set of finite states*
- *$q_0 \in Q$: start state*

- *$F \subseteq Q$: Accepting final states*
- *A transition function $\delta : Q \times \Sigma_\epsilon \mapsto P(Q)$*

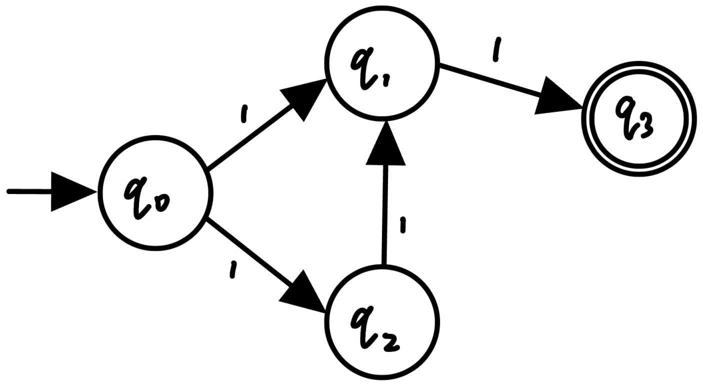
A string x is accepted by this NFA if $\exists y_1 y_2 \dots y_m \in \Sigma_\epsilon$ and $q_1, \dots, q_m$ such that

1. *$y_1 y_2 \dots y_m = x$*
2. *$q_i \in \delta(q_{i-1}, y_i)$ for all i*
3. *$q_m \in F$*

Note that it is possible that there is no arrow coming out of a state given a character. In this case, we immediately reject.

Definition 2.2.2. *For an NFA A, we define $L(A) = \{$strings accepted by $A\}$*

Example 2.2.3. *Consider the following NFA A*



We have $L(A) = \{11, 111\}$

Theorem 2.2.4. *For all NFA A, $L(A)$ is regular.*

Proof. we construct a DFA A', which consists

- Q' $= P(Q)$
- $q_0' = B(q_0) =$ all states reachable using any number of $\epsilon-$transitions from q_0
- $\delta'(S, b) = \bigcup_{q \in S} \bigcup_{p \in \delta(q,b)} B(p)$
- $F' = \{S \in P(Q) : S \cap F \neq \emptyset\}$

By induction, the state S that Q' is in after reading $x_1 \dots x_i$ equals the set of all states that Q could have reached after reading $x_1 \dots x_i$ with any number of ϵ's transitions. □

Theorem 2.2.5 (Kleene's Theorem). *Any regular languages can be made out of the languages*

- $\emptyset$
- $\{\epsilon\}$
- *Singleton $\{a\} : a \in \Sigma$*
- *Union, concatenation or $*$.*

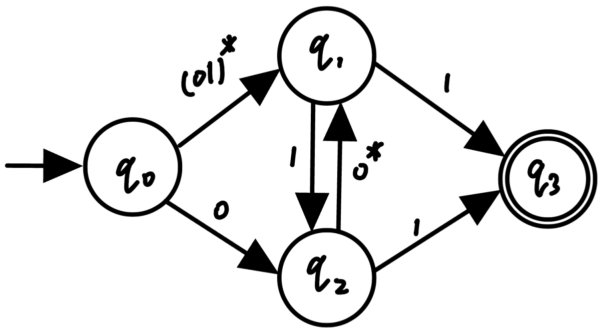
A language expressed using these operations are called regular expressions

Definition 2.2.6. *A generalised NFA is an NFA with*

1. *On the arrows, we have regular expressions*
2. *Unique start state which has arrows going into it*
3. *Unique accept state with no arrows going out of it*

A string x is accepted by the GNFA if $\exists$ strings $y_1, y_2, \dots, y_m$ such that $y_1 y_2 \dots y_m = x$ and states $q_1 q_2 \dots q_m$ such that y_i is the language written on the arrow $q_{i-1} \to q_i$. Similar definition as the regular NFA just characters becomes strings.

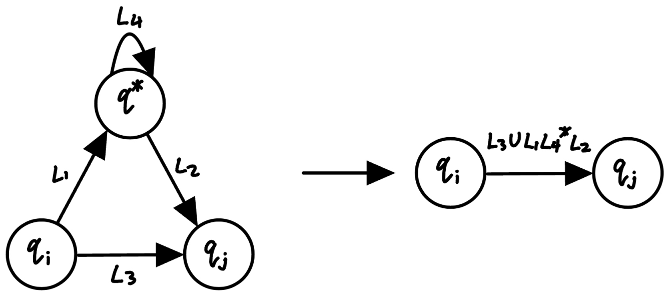
Example 2.2.7. *The following is an example of a GNFA*



On input $0101011, 01, 00001001$*, such GNFA accepts the string.*

Theorem 2.2.8. *Every GNFA accepts a regular language*

Proof. Pick some state q^* that is not start or accept state. To remove q^*, we concatenate the strings along the path with the state that want to remove, and combine with the string on the path without the deleted state to form a union.



□

2.3 Non-regular languages

Remark 2.3.1. *Since there are countably many regular languages $\subseteq \Sigma^*$ while there are uncountably many languages $\subseteq \Sigma^*$. By diagonalisation proof, this shows that non-regular languages exists. However, this does not gives us an explicit description of a non-regular languages. Hence, we are going to find out ways that we can get those examples of non-regular languages.*

Lemma 2.3.2 (Pumping Lemma). *Let $L \subseteq \Sigma^*$ be a regular language. Then there exists some $p \in \mathbb{N}$ such that for all strings $w \in L, |w| > p$, w can be broken into $x, y, z \in \Sigma^*$ such that*

1. $xyz = w$
2. $|xy| \leq p, |y| > 0$
3. $\forall i \geq 0, xy^iz \in L$

Proof. Since L is regular, we have a DFA A for L

Let we take p be the number of states in $A + 1$. Take a string with length greater than p, say $w_1 \dots w_n$ where $n > p$.

Let $q_0q_1 \dots q_p$ be the states visited while reading $w_1w_2 \dots w_p$.

Since $p >$ number of states in A, there is one $0 \leq i < j \leq p$ such that $q_i = q_j$. Then $(w_{i+1} \dots w_j)$ takes q_i to $q_j = q_i$.

Let $x = w_1 \dots w_i, y = w_{i+1} \dots w_j, z = w_{j+1} \dots w_n$, then xy^iz is accepted by A. □

Example 2.3.3. *Here are some examples of non-regular languages*

$$L_1 = \{0^n1^n : n \in \mathbb{N}\}$$

$$L_2 = \{\textit{strings } \in \{0,1\}^* \textit{ with equal number of } 0\textit{'s and } 1\textit{'s}\} \qquad L_3 = \{1^{2^n} : n \in \mathbb{N}\}$$

Note that this might seems similar to L_2 but it is regular.

$$L_4 = \{\textit{strings } \in \{0,1\}^* \textit{ with equal number of } 01\textit{'s and } 10\textit{'s}\}$$

The idea is that if you have a 01 and if another 01 exists, there must be a 10 before that.

Proof. L_1 is not regular:

Suppose L_1 is regular, then by Pumping Lemma, there exists some p (pumping length) such that all the conditions of the pumping lemma holds.

Consider $w = 0^{2p}1^{2p} \in L_1$. By Pumping Lemma, $w = xyz$ where $|xy| \leq p, |y| > 0$. And $xy^iz \in L_1$.

However, xy only consists of 0's, where y must have at least one 0. Then, xy^2z would have more 0's than 1's, contradiction.

L_3 is not regular:

If it is regular, there exists a pumping length p. Take $w = 1^{2^p}$, we can write $w = xyz$ where $|xy| < p, |y| > 0$

take $w = 1^{2^p}$, we can write $w = xyz$, where $|xy| \leq p, |y| > 0$. Then, $|xy^2z| \in (2^p+1, 2^p+p) < 2^{p+1}$. So $xy^2z \notin L_4$. □

Example 2.3.4. *Primes* $= \{1^p : p \textit{ is a prime}\}$

Proof. Assume Primes is regular, then there is a pumping length k.

Take $w = 1^q$ where q is the first prime greater than k. Then, we can write $w = xyz$ where $|y| > 0, |xy| \leq k$.

Take $xy^{q+1}z = 1^{q(|y|+1)} \notin$ Primes. □

Definition 2.3.5. *Let L, L' be two languages, we define $L/L' = \{a \in \Sigma^* : \exists b \in L', ab \in L\}$*

Theorem 2.3.6. *If L is regular, L' is arbitrary, then L/L' is regular.*

Question 2.3.1. *Let Σ', Σ be finite sets. Let $f : \Sigma' \to \Sigma$ be some map. Take some $L' \subseteq (\Sigma')^*$ which is regular. Consider $f(L') = \{f(x_1)f(x_2) \dots f(x_n) : x_1 \dots x_n \in L'\}$. Is $f(L')$ regular?*

Answer: Yes

Example 2.3.7. $R = \{ww : w \in \{0,1\}^*\}$ *is not regular.*

$Perfect_{sq} = \{0^{k^2} : k \in \mathbb{N}\}$ *is not regular.*

3 Context Free Languages (CFLs)

A language defined by Context Free Grammar

3.1 Context Free Grammars (CFGs)

A grammar G is a collection of nonterminals $A_0, A_1, \ldots$ and rules $A_i \to$ some strings in $\{\Sigma \cup (A_0, A_1, \ldots\})^*$.

Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a variable. The string consists of variables and other symbols called terminals. One variable is designated as the start variable, usually occurs on the left-hand side of the topmost rule.

We say x is derivable from G if there exists a sequence of strings in $(\Sigma \cup \{A_0, A_1, \ldots\})^*$ starting with A_0 and ending with x such that each successive string is obtained from the previous using one of the substitution rules

Example 3.1.1. *Suppose the grammar G is the following*

1. *Sentence → Noun Verb Noun*
2. *Noun → Noun "and" Noun*
3. *Noun → "I" | "You" | "Bananas" | "Apples"*
4. *Verb → "Eat" | "Drink" | "Kick"*

A sentence is made out of a noun, a verb and a noun, where a noun can be made out of a noun and another noun connecting with the word "and"

Examples of strings derivable from this grammar including:

- *Sentence → Noun Verb Noun → I Eat Bananas*
- *Sentence → Noun Verb Noun → I Kick Bananas*
- *Sentence → Noun Verb Noun → I Eat Noun and Noun → I Eat Bananas and Noun and Noun → I Eat Bananas and Apples and Bananas*

Example 3.1.2. *We have the following grammar G*

1. $s \to A|B, s$ *: the start symbol*
2. $A \to 0|1A$
3. $B \to 101$

The following strings are derivable from G

- *110: Since* $s \to A \to 1A \to 11A \to 110$
- *0: Since* $s \to A \to 0$
- *101: Since* $s \to B \to 101$

This language can be expressed as the regular language $L(G) = (101) \cup (1^*0)$

Definition 3.1.3. *For a grammar* $G, L(G) = \{$*strings* $x \in \Sigma^*$ *for which there is a derivation of* x *from* $G\}$

Example 3.1.4. *We have the following grammar G*

1. $s \to 0s1|\epsilon$

We have $L(G) = \{0^n 1^n : n \in \mathbb{N}\}$

Example 3.1.5. *The Palindrome $\{wbw^R : w \in \{0, 1\{^*, b \in \{0, 1, \epsilon\}\}$ is generated by the grammar*

1. $S \to 0S0|1S1|0|1|\epsilon$

Theorem 3.1.6. *Regular languages are CFLs (Context Free Languages)*

Proof Idea. Method 1. Introduce 1 nonterminal rule per state

Method 2. Show that the class of CFLs is closed under regular operations

Suppose L_1, L_2 are CFL, where G_1, G_2 are their CFGs, with distinct nonterminals, S_1, S_2 are their start symbols. Then $S \to S_1 \cdot S_2$ along with G_1, G_2 is the CFG for $L_1 \cdot L_2$.

$S \to S_1|S_2$ along with G_1, G_2 is the CFG for $L_1 \cup L_2$

$S \to \epsilon|S_1S$ along with G_1, is the CFG for L_1^*. □

Example 3.1.7. *Let $L = \{w : w$ has an equal number of 0's and 1's$\}$, then the context free grammar that generate this language is*

$s \to \epsilon|ss|0S1|1S0$

Definition 3.1.8. *A context free grammars are given by*

- *Σ: terminals*
- *V: Nonterminals/variables*
- *R: Rules (consists of a variable, and a string composed of variables and terminals*
- *S: Start symbol*

For a CFG G, there is $L(G)$

3.2 Chomsky Normal Form

Definition 3.2.1 (Chomsky Normal Form). *A context-free frammar is in Chomsky Normal Form (CNF) if*

1. *No rules of the form $N \to \epsilon$ unders $N = S$*
2. *All rules are either $N \to N_1N_2$ where N_1, N_2 are nonterminals and not equal to S, or $N \to a$, where a is the terminal*

Theorem 3.2.2. *Every CFG G can be converted into a G' in CNF such that $L(G) = L(G')$*

Proof. First, we add a new start variable S_0 and the rule $S_0 \to S$. This guarantees that the start variable doesn't occur on the right-hand side of a rule.

Second, we take care of all ϵ-rules. For each $A \to \epsilon$ where $A \neq S$, we remove it. Then for each occurrence of an A on the right-hand side of a rule, we add a new rule with that occurrence deleted. That is, if $R \to uAv$, we add $R \to uv$. If $R \to uAvAw$, we add $R \to uvAw, R \to uAvw, R \to uvw$. If $R \to A$, we add $R \to \epsilon$.

Third, we handle all unit rules. For any $A \to B$, we remove it. If there is $B \to u$, we add $A \to u$.

Finally, we convert all remaining rules in the form of $A \to u_1 \dots u_k$ where $k \geq 3, u_i$ could be terminals or variables. We replace it with $A \to u_1A_1, A_1 \to u_2A_2, \dots, A_{k-2} \to u_{k-1}u_k$. And we replace any terminals u_i with $U_i \to u_i$. □

Question 3.2.1. *Given $x = x_1 \dots x_n \in \Sigma^*$, and a CFG G, can we decide if $x \in L(G)$ and if so give a derivation of x in G?*

Remark 3.2.3. *The natural algorithm is based on the dynamic programming. But first simplify the grammar*

Theorem 3.2.4 (Parsing Algorithm (CYK))**.** *Given string $x_1, \dots x_n$, define*

$$T[i,j] = \{\textit{nonterminals } N : N \to \textit{(derives) } x_i x_{i+1} \dots x_j\}$$

Note that $S \in T[1,n] \iff x \in L(G)$.
The algorithm works as follows, given input $x \in \Sigma^*, G$ in CNF

1. *Compute $T[i,i] = \{$all nonterminals N with the rule $N \to x_i\}$ for all $i \in [n]$*
2. *For $k = 2$ to n,*

```
    For $i = 1$ to $n - k + 1$
        Define $j = i + k - 1$
        set $T[i,j] = \emptyset$
        For $l = i$ to $j - 1$
            For each $N_1$ in $T[i,l]$ and $N_2$ in $T[l+1,j]$
                If $M \to N_1 N_2$ is a rule of $G$, add $M$ to $T[i,j]$
return $S \in T[1,n]$
```

Note that we have $T[i,j] = \bigcup_{l=i}^{j-1} \{N : N \to N_1 N_2$ is a rule of $G \wedge N_1 \in T[i,l] \wedge N_2 \in T[l+1,j]\}$

Theorem 3.2.5. *Parsing a string of length n under G can be down in time $\mathcal{O}_G(n^3)$.*

Remark 3.2.6. *The derivation helps us "understand" the mathematical expression + evaluate it.*

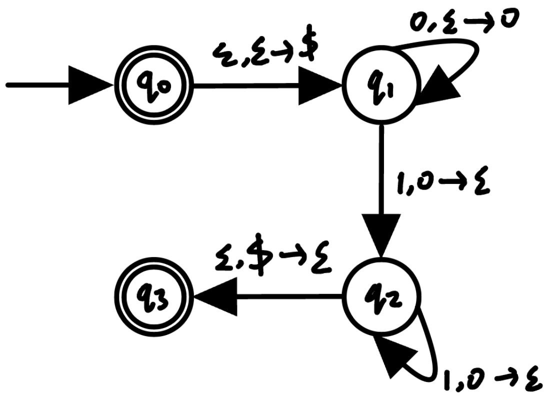
3.3 Push-Down Automata

Definition 3.3.1. *A push-down automata is a NFA with access to an infinite stack with the following*

- *Q: the set of states*
- *Σ: the input alphabet*
- *Γ: The stack alphabet*
- *$q_0 \in Q$*
- *$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$*
- *$F \subseteq Q$*

Notation 3.3.2. *We write $(q', c) \in \delta(q, a, b)$ if the current state is q, next input character is a, b is on the top of the stack, then change the state to q', replace the b with c on the top of the stack.*
If $b = \epsilon$, then we add c on top of the stack. If $c = \epsilon$, we remove b on the top of the stack.
Graphically, this can be represented as

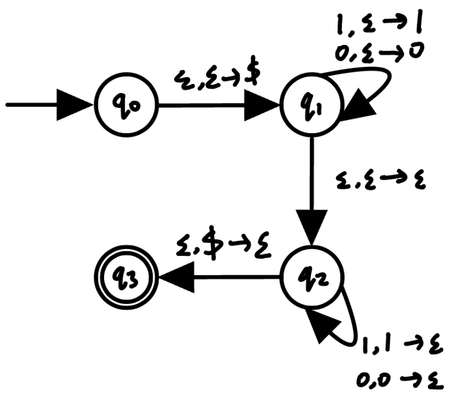
Example 3.3.3. *For the language $L = \{0^n 1^n : n \in \mathbb{N}$, the push down automata is (there are many ways)*



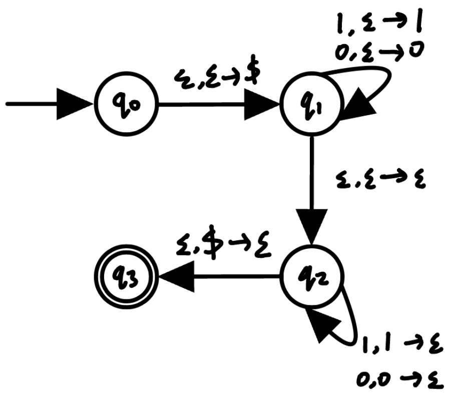
Definition 3.3.4. *$x_1 \ldots x_n \in L(A)$ for some push down automata A if there eixsts $y_1 \ldots y_m \in \Sigma_\epsilon, q_0, q_1 \ldots q_m \in Q, s_0 \ldots s_m \in \Gamma^*$ such that*

1. *$y_1 \ldots y_m = x_1 \ldots x_n$*
2. *$q_i, b \in \delta(q_{i-1}, y_i, a)$ for all $i = 1, \ldots m$ where $s_{i-1} = aw, s_i = bw$ for all $i = 1, \ldots n-1, w \in \Gamma^*$*
3. *$s_0 = s_m = \epsilon, q_m \in F$*

Example 3.3.5. *For the language $L = \{ww^{Rev} : w \in \Sigma^*\}$, the push down automata is*



Example 3.3.6. *For the language $L = \{w : w \text{ is a palindrome}\}$, the push down automata is*



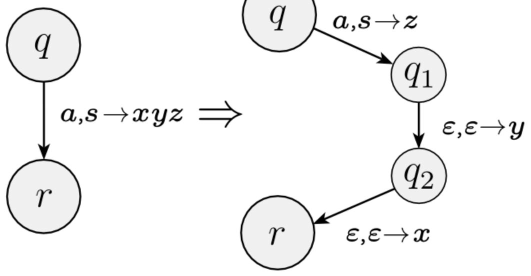
Theorem 3.3.7. *For every CFG, G, there is a PDA, A that $L(G) = L(A)$*

Proof. We build the pushdown automata $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

Let q, r be the states of P and $a \in \Sigma_\epsilon, s \in \Gamma_\epsilon$. If we want the PDA to go from q from r when it reads a and pops s. Furthermore, we want it to push the entire string $u = u_1 \dots u_l$ at the same time. We can implement this action by introducing new states $q_1, \dots q_{l-1}$ and setting the transition as follows

$$\begin{aligned}(q_1, u_l) &\in \delta(q, a, s)\\ \delta(q_1, \epsilon, \epsilon) &= \{q_2, u_{l-1}\}\\ \delta(q_2, \epsilon, \epsilon) &= \{q_3, u_{l-2}\}\\ &\dots\\ \delta(q_{l-1}, \epsilon, \epsilon) &= \{r, u_1\}\end{aligned}$$

We will use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r. The following figure shows this implementation



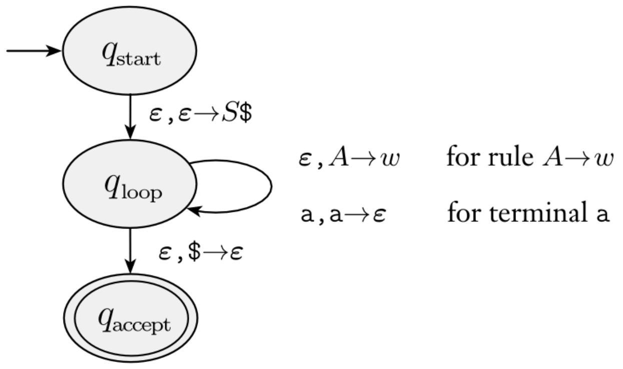
We have $Q = \{q_0, q_{accept}, q_{loop}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initialising the stack to contain the symbols \$ and S, in the informal description: $\delta(q_0, \epsilon, \epsilon) = \{(q_{loop}, S\$)\}$.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, w) : \text{where } A \to w \text{ is a rule in } R\}$

Second, we handle the case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{loop}, a, a) = \{q_{loop}, \epsilon\}$ (If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.).

Finally, we handle case (c) wherein the empty stack marker \$ is on the top of the stack $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$

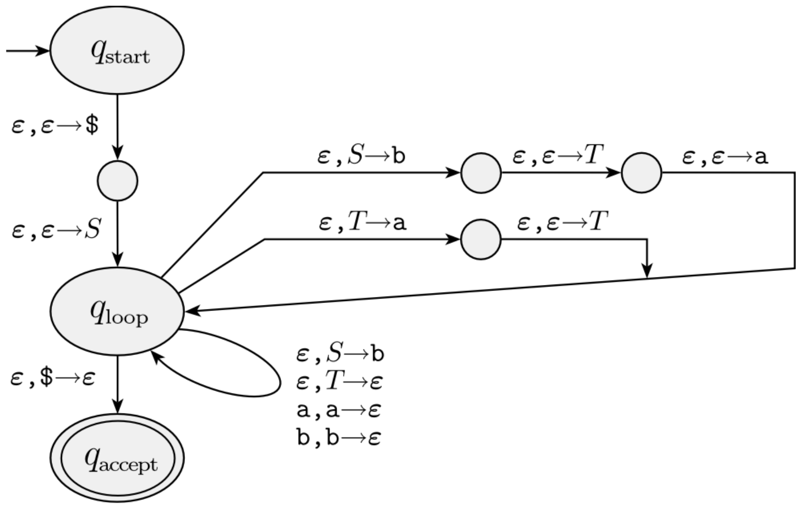


q_{start}
$\varepsilon, \varepsilon \to S\$$
q_{loop}
$\varepsilon, A \to w$ for rule $A \to w$
a, a→ε for terminal a
$\varepsilon, \$ \to \varepsilon$
q_{accept}

□

Example 3.3.8. *Take the following CFG G*

1. $S \to aTb|b$
2. $T \to Ta|\epsilon$



q_{start}
$\varepsilon, \varepsilon \to \$$
$\varepsilon, \varepsilon \to S$
$\varepsilon, S \to \text{b}$
$\varepsilon, \varepsilon \to T$
$\varepsilon, \varepsilon \to \text{a}$
$\varepsilon, T \to \text{a}$
$\varepsilon, \varepsilon \to T$
q_{loop}
$\varepsilon, S \to \text{b}$
$\varepsilon, T \to \varepsilon$
a, a→ε
b, b→ε
$\varepsilon, \$ \to \varepsilon$
q_{accept}

Theorem 3.3.9. *Given a pushdown automata A, there exists some CFG G where $L(A) = L(G)$.*

Proof. Let the set of variables be $\{A_{pq} : p, q \in Q\}$. With the start variable being $A_{q_0, q_{accept}}$.

1. For each $p, q, r, s \in Q, u \in \Gamma$ and $a, b \in \Sigma_\epsilon$ if $(r, u) \in \delta(p, a, \epsilon), (q, \epsilon) \in \delta(s, b, u)$, put the rule $A_{pq} \to aA_{rs}b$ in G.
2. For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.
3. Finally, for each $p \in Q$, put the rule $A_{pp} \to \epsilon$ in G.

We have if A_{pq} generates x, then x can bring p from p with empty stack to g with empty stack. And if x can bring P from p with empty stack to g with empty stack, A_{pq} generates x.

See the textbook for details. □

3.4 Non Context Free Languages

Theorem 3.4.1 (Pumping Lemma for Context Free Languages)**.** *If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions*

1. *for each $i \geq 0, uv^ixy^iz \in A$*
2. *$|vy| > 0$*
3. *$|vxy| \leq p$*

Proof. See the textbook for details. □

Example 3.4.2. *$L_1 = \{ww^{Rev} : w \in \Sigma^*\}$ is a CFL.*
$L_2 = \{ww : w \in \Sigma^*\}$ is not a CFL.

Proof. Suppose L_2 is a CFL, with a pumping length p. Take $w = 0^p1^p0^p1^p \in L_2$.

First, we show that the substring vxy must straddle the midpoint of w. Otherwise, if the substring occurs only in the first half of w, pumping s to uv^0xy^0z results that the first half contains of at least one of the 0s and 1s less than p, and so it cannot be of the form ww. Similarly, if vxy occurs in the second half of s, pumping s to uv^0xy^0z results that the second half contains of at least one of the 0s and 1s less than p.

But if the substring vxy straddles the midpoint of s, when we try to pump s down to uxz it has the form $0^p1^i0^j1p$, where at least one of the i and j cannot be p. This string is not of the form ww. Contradiction. □

Example 3.4.3. *$L = \{0^n1^n0^n : n \in \mathbb{N}\}$ is not a CFL*

Proof. Suppose it is. Let p be the pumping length and consider $0^p1^p0^p$. For all the decomposition of $w = uvxyz$,

- If v and y contain only one type of alphabet symbol, uv^2xy^2z cannot contain equal numbers.
- If either v, y contain more than one character, then uv^2xy^2z would not be in the correct order.

□

Example 3.4.4. *Let $\Sigma = \{0, 1\}, L = \{w_1 \dots w_n \in \Sigma^* : w_j = 1 \iff j$ is prime$\}$. Then L is not a CFL.*

Proof. Suppose p be the pumping length, let $s \in L$ where $|s| = q \geq p$ and q is a prime. This means, the last character of s is 1. Take $s = uvxyz$. Take $uv^{q+1}xy^{q+1}z$, we have $|uv^{q+1}xy^{q+1}z| = q + q|vy|$ which is not a prime, but the last character is 1. □

Theorem 3.4.5. *The union of two CFLs is a CFL.*

Theorem 3.4.6. *The intersection of two CFLs need not be a CFL*

Proof. Take $\{0^k1^n2^n : k, n \in \mathbb{N}\} \cap \{0^n1^n2^k : k, n \in \mathbb{N}\} = \{0^n1^n2^n : k, n \in \mathbb{N}\}$.
The previous two is a CFL, while the intersection of them is not. □

Theorem 3.4.7. *The complement of a CFL need not be a CFL.*

Proof. If the complement of a CFL is a CRL, then $A \cap B = (A^c \cup B^c)^c$
showing $A \cap B$ is also a CFL, but CFL is not closed under complement, contradiction. □

Theorem 3.4.8. *The intersection of a CFL and a regular language is a CFL.*

4 Computability Theory

A computing machine which has finite memory of its own and the ability to read and write onto a tape

4.1 Turing Machine

Definition 4.1.1. *A Turing Machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{Accept}, q_{Reject})$ where*

- Q : *set of states*
- Σ : *input alphabet where* $\sqcup \notin \Sigma$
- Γ : *Tape alphabet where* $\Sigma \subseteq \Gamma, \sqcup \in \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L \times R\}$

Definition 4.1.2. *The configuration of a Turing Machine at some point in time tells*

- *The contents of the tape*
- *The location of the "tape head"*
- *Current state in Q*

Example 4.1.3. *If the current tape is written $s_1 \dots s_m$ where the tape head is at s_i, the state is at q, the configuration is $s_1 \dots s_{i-1} q s_i \dots s_m$.*

Definition 4.1.4. *There are some rules with how the Turing Machine can run. If the current configuration is $uaqbv$ where $u, v \in \Sigma^*, a, b \in \Sigma$.*
Suppose $\delta(q, b) = (q', b', D)$ where D is either L or R, then the next configuration is

- *If $D = L$ then the configuration is $uq'ab'v$*
- *If $D = R$ then the configuration is $uab'q'v$*
- *If $D = L$ and the current configuration is qbv then the next configuration is $q'b'v$*

If the state is on q_{Accept} we accept and stop, if q_{Reject} we reject and stop.

Definition 4.1.5. *We say $w \in \Sigma^*$ is accepted by a Turing Machine M if starting with w on the tape and the Turing Machine accepts.*
The language accepted by / recognised by M is written as

$$L(M) = \{w \in \Sigma^* \text{ s.t. } M \text{ accepts } w\}$$

Definition 4.1.6. *A Turing Machine M decides $L \subseteq \Sigma^*$ if $\forall w \in L, M$ accepts w. And $\forall w \notin L, M$ rejects w.*

Remark 4.1.7. *If L is decided by M, then L is recognised by M.*

Definition 4.1.8. *A language is Turing Recognisable if there exists a Turing Machine M such that M recognises L.*

A language is Turing decidable if there exists a Turing Machine M such that M decides it.

Example 4.1.9. *$L = \{0^n 1^n : n \in \mathbb{N}\}$ is Turing Recognisable.*

Take $\Gamma = \{\sqcup, 0, 1, \cancel{0}, \cancel{1}\}$. When the tape head sees a 0, put a scratch on it, go to the very end of the tape when it sees a blank or a 1 with a scratch, put a scratch at the very last 1 without a bar.

q_0 : so far an equal number of 0s and 1s have been given a bar and I am at the leftmost character without a bar.

$\delta(q_0, 0) = (q_1, \bar{0}, R), \delta(q_0, 1) = (q_{Reject}, \ldots), \delta(q_0, \bar{1}) = (q_{Accept}, \ldots)$

Example 4.1.10. *$L = \{0^n 1^n 2^n : n \in \mathbb{N}\}$ is Turing Recognisable (it is also decidable)*

$L = \{0^n 1^m : n|m\}$ is Turing Recognisable (it is also decidable)

For every 0, scratch one and scratch a corresponding 1. After one iteration, unscratch all the 0s and repeat. Until all the 1s have been scratched, in this case all 0 must be scratched as well and accept.

Remark 4.1.11. *If L is a language recognised by M. M' is M with q_{Accept}, q_{Reject} switched. Then M' recognises some language that is a subset of L^c.*

Example 4.1.12. *$L = \{0^n 1^{n^2} : n \in \mathbb{N}\}$ is Turing Recognisable and decidable.*

For every 0, scratch one and scratch a corresponding 1, add a bar on the first 0 without a bar. After one iteration, unscratch all the 0s and repeat, add a bar on the next 0 without a bar. Until all the 1s have been scratched, in this case all 0 must be scratched and contains a bar, then we accept.

$L = \{0^n 1^{2^n} : n \in \mathbb{N}\}$ is Turing Recognisable and decidable.

Scratch off the first 1, then go to the first 0 that has not been scratched off, scratch off it, then go to the first 1 that has been scratched off, for every 1 that has been scratched off, cratch off the next 1 that has not been scratched off (essentially doubling it). Then, repeat this process.

$L = \{0^{2^{2^{2^n}}} : n \in \mathbb{N}\}$

4.2 Variations of Turing Machines

Definition 4.2.1. *A multitape Turing Machines is a k-tape Turing Machine with $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$.*

Start with w on the first tape, and remaining tapes blank.

Theorem 4.2.2. *For every Multitape Turing Machine M, there is some single tape Turing Machine M' such that $\forall w$, the behaviour of M on w (accept/reject/loop) is the same as the behaviour of M' on w.*

Proof. Write all the contents on the tape into a single tape, separated by some other symbol #. Write a bar on one element in each section corresponding to a tape to indicate the tape head. □

Definition 4.2.3. *A nondeterministic Turing Machine has a $\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$.*

An NTM accept w if there exists a choice of valid steps to take leading to q_{Accept}.

The configuration of an NTM under input w can be represented as a tree.

Theorem 4.2.4. *For all NTM M, there exists a TM M' such that $L(M) = L(M')$ (although it is illogical to talk about the behaviour of M as it can choose different options to behave differently)*

Proof. Consider the tree of possible configurations. Trying to explore the full tree looking for one accept. Keep track of a string $s_1 s_2 \ldots s_l$ where $s_i \in [|Q \times \Gamma \times \{L, R\}|]$, this tells us where in the tree we are. Explore the tree in BFS way to look for an accepting configuration. □

4.3 Recognisable and Decidable Languages

Recall that for a TM M

- L is recognised by M if $L = \{x \in \Sigma^* : M \text{ accepts } w\}$
- L is decided by M if $L = \{w \in \Sigma^* : m \text{ accepts } w$ and M halts on all inputs

Theorem 4.3.1. *There are languages recognisable but not decidable, since there is countably many Turing Machines.*

Definition 4.3.2. $A_{TM} = \{\langle M, w\rangle : M \textit{ is a TM}, w \in \Sigma^*, M \textit{ accepts } w\}$

Theorem 4.3.3. A_{TM} *is recognisable, but not decidable*

Proof. Simulate M on w and accepts if M accepts w, and rejects if M rejects w.

If M loops, A_{TM} simply never returns (hence rejects) as it nevers fall into the case where it can accept or reject.

We can simulate this with a 2-tape machine. Keep $\langle M, w\rangle$ on the first tape. The second tape will store the current configuration.

For A_{TM} being not decidable. This can be done by a diagonalisation argument.

Let B be a decider for A_{TM}, and the Turing Machines can be listed as $M_1, M_2, \ldots$. Then D is a Turing Machine that runs as follows

```
On input w
Find i s.t. w_i = w
Run B(<M_i, w_i>)
If it accepts, reject. Else, reject
```

If $D \neq M_i$ since $D(w_i) \neq M_i(w_i)$. But D is some M_i. Contradiction. □

From undecidability of the A_{TM} we can deduce undecidability of many other TM-related languages.

Definition 4.3.4. $Halt_{TM} = \{\langle M, w\rangle : M \textit{ halts on } w\}$
$E_{TM} = \{\langle M\rangle : L(M) = \emptyset\}$
$All_{TM} = \{\langle M\rangle : L(M) = \Sigma^*\}$

Theorem 4.3.5. E_{TM} *is not decidable.*

Proof. Suppose it is, let B be a decider for E.

Here is a TM C deciding A_{TM}

```
On input <M, w>
Create a TM M' that on input w', ignores w', prints w on the tape, runs M
Run B on <M'>, if it accepts, reject. Else accept
```

□

Theorem 4.3.6. $\overline{E_{TM}}$ *is recognisable.*

Proof.
```
For i = 1 to infty:
    For j = 1 to i:
        Run M on w_j for i steps
        If accepts, accept
Reject
```

□

Theorem 4.3.7. *If L and $\overline{L}$ are both recognisable then L is decidable.*

Proof. On input w, run recogniser for L, L^c in parallel. One must halt with the answer. If L accepts then accept, if L^c accepts then reject. □

Theorem 4.3.8. *All_{TM} and All_{TM}^c are both not recognisable.*

4.4 Enumerators

A TM like machine that prints out list of strings on an additional write-only tape

Definition 4.4.1. *An enumerator for a language L is a TM that prints out a sequence $w_1, w_2, \ldots$ s.t. $L = \{w_1, w_2, \ldots\}$*

Theorem 4.4.2. *A language is recognisable if and only if it is enumerable*

Proof. $(\Rightarrow)$:

```
For i = 1 to infty:
    For j = 1 to i:
        Run M on w_j for i steps
        If accept, print w_j
```

$(\Leftarrow)$:
On input w, just run E and accept if we see w □

Theorem 4.4.3. *A language is decidable if and only if it is enumerable in lexicographical order*

Proof. $(\Leftarrow)$
Given a lexicographical order enumerator E for L. We construct a decider M.

```
On input w
Run E
While E prints a string
Compare the string > w
If True, then reject. Else accept
```

$(\Rightarrow)$
Just run M on all strings in lexicographical order, and prints if accept. □

4.5 Closure under operations

Definition 4.5.1. *Let $L \subseteq \Sigma^* \times \Sigma^*$, then*

$$\pi_1 L = \{w \text{ s.t. } \exists v \in \Sigma^* \text{ s.t. } (w, v) \in L\}$$

Example 4.5.2. $\pi_1(A_{TM}) = \overline{E_{TM}}$

Theorem 4.5.3.

	Recognisable	*Decidable*
$\cup$	✓	✓
$\cap$	✓	✓
complement	×	✓
$\cdot$	✓	✓
$*$	✓	✓
projection	✓	×

Proof. For $\cdot, *$ there are finitely many ways to split the string, just check for each string.

For projection under decidable case, let $BA_{TM} = \{(\langle M, w\rangle, i) : M \text{ accepts } w \text{ within } i \text{ steps}\}$. Then $\pi_1(BA_{TM}) = A_{TM}$ □

4.6 Reduction

Definition 4.6.1. *A function $f : \Sigma^* \to \Sigma^*$ is computable $\iff$ there exists a TM M, such that for all $w \in \Sigma^*$, halts with $f(w)$ on the tape.*

Example 4.6.2.

Definition 4.6.3. *A is mapping reducible to B (denoting $A \leq_m B$) if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for all $w \in \Sigma^*, w \in A \iff f(w) \in B$.*

In this case, f is the reduction of A to B

Example 4.6.4. *f can be some transformation of Turing Machines from some $w = \langle M\rangle$ to some $\langle M'\rangle$.*

Proposition 4.6.5. *If $A \leq_m B$, and B is decidable, then A is decidable. We can also channge decidability to Turing recognisability.*

Proof. We consturct a decider N for A, suppose M is a decider for B, on input w:

1. Compute $f(w)$
2. Run $f(w)$ on M and output whatever M outpus

$w \in A \implies f(w) \in B \implies M$ accepts $\implies N$ accepts.
$w \notin A \implies f(w) \notin B \implies M$ rejects $\implies N$ rejects. □

Proposition 4.6.6. *If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$*

Proof. There exists a function f such that $w \in A \iff f(w) \in B \implies w \notin A \iff f(w) \notin B \implies w \in \overline{A} \iff f(w) \in \overline{B}$. □

Proposition 4.6.7. *If A is Turing recognisable and $A \leq_m \overline{A}$ then A is decidable.*

Proof. Since $A \leq_m \overline{A}$, $\overline{A} \leq_m A$. And since A is Turing recognisable then $\overline{A}$ is also Turing recognisable. Hence, A is co-Turing recognisable, thus is decidable. □

Proposition 4.6.8. *(1) A is Turing recognisable $\iff A \leq_m A_{TM}$ (2) A is decidable $\iff A \leq_m 0^*1^*$*

Proof. (1) $(\Rightarrow)$: we show $A \leq_m A_{TM}$. We construct the computable function as follows, on input x, output $\langle M, x\rangle$ and we have $x \in A \iff \langle M, x\rangle \in A_{TM}$

$(\Leftarrow)$: A_{TM} is Turing recognisable hence A is Turing recognisable.

(2) $(\Rightarrow)$: we show $A \leq_m 0^*1^*$. Since A is decidable there is a decider M that decides A. We construct the following computable function, on input x, run M on x. If M accepts, output 01, otherwise, output 10.

$(\Leftarrow)$: Since 0^*1^* is decidable hence A is decidable. □

Proposition 4.6.9. *$HALT_{TM} = \{\langle M, w\rangle : M \text{ is a TM } \wedge M \text{ acceots } w\}$ is undecidable.*

Proof. We show $A_{TM} \leq_m HALT_{TM}$ by constructing a computable function F. On input $\langle M, w\rangle$:

1. Construct a TM $\langle M'\rangle$ where M' satisfies on input x
   (a) Run M on x
   (b) If M accepts, M' accepts

   (c) If M rejects, M' infinite loops

2. Output $\langle M', w\rangle$

We have $\langle M,w\rangle \in A_{TM} \implies M$ accepts $w \implies M'$ accepts $w \implies M'$ halts on $w \implies \langle M',w\rangle \in HALT_{TM}$.
Similarly, $\langle M,w\rangle \notin A_{TM} \implies M$ loops or rejects on $w \implies M'$ loops on $w \implies M'$ doesn't halt on $w \implies \langle M',w\rangle \notin HALT_{TM}$ □

Proposition 4.6.10. *$\{\langle M_1, M_2\rangle : M_1, M_2$ are TMs and $L(M_1) = L(M_2)\} = EQ_{TM} \geq_m E_{TM} = \{\langle M\rangle : M$ is a TM and $L(M) = \emptyset\}$*

Proof. We construct a computable function F, on input $\langle M\rangle$

1. Construct M_1 so that on input x, reject.
2. Output $\langle M, M_1\rangle$

We have $\langle M\rangle \in E_{TM} \implies L(M) = \emptyset \implies L(M) = L(M_1) \implies \langle M, M_1\rangle \in EQ_{TM}$.
We have $\langle M\rangle \notin E_{TM} \implies L(M) \neq \emptyset \implies L(M) \neq L(M_1) \implies \langle M, M_1\rangle \notin EQ_{TM}$. □

Proposition 4.6.11. $A_{TM} \leq_m \overline{E_{TM}}$

Proof. We construct a computable function F, on input $\langle M,w\rangle$

1. Construct M' such that on input x
   (a) If $x \neq w$ reject
   (b) Otherwise, run M on w, accept if accept, reject if reject
2. Output $\langle M'\rangle$

We have $\langle M,w\rangle \in A_{TM} \implies w \in L(M') \implies \langle M'\rangle\overline{E_{TM}}$
Similarly, $\langle M,w\rangle \notin A_{TM} \implies L(M') = \emptyset \implies \langle M'\rangle E_{TM}$ □

Definition 4.6.12. *A language A is co-Turing recognisable if $\overline{A}$ is Turing recognisable.*

Proposition 4.6.13. *EQ_{TM} is not Turing recognisable and not co-Turing Recognisable.*

Proof. For not co-Turing recognisability, we show $A_{TM} \leq_m EQ_{TM}$ where A_{TM} is not co-Turing recognisable. We construct the following computable function F, on input $\langle M,w\rangle$

1. Construct M_1 where on input x, accepts
2. Construct M_2 where on input x, run M on w and accept if accepts. If M rejects, we reject.
3. Output $\langle M_1, M_2\rangle$

We have $\langle M,w\rangle \in A_{TM} \implies L(M_1) = \Sigma^* = L(M_2) \implies \langle M_1, M_2\rangle \in EQ_{TM}$
Similarly, $\langle M,w\rangle \notin A_{TM} \implies L(M_1) = \Sigma^*$ but $w \notin L(M_2) \implies \langle M_1, M_2\rangle \notin EQ_{TM}$
We also show $\overline{A_{TM}} \leq_m EQ_{TM}$ where $\overline{A_{TM}}$ is not Turing recognisable. We construct the following computable function F, on input $\langle M,w\rangle$

1. Construct M_1 where on input x, rejects
2. Construct M_2 where on input x, run M on w and accept if accepts. Otherwise, we reject.
3. Output $\langle M_1, M_2\rangle$

We have $\langle M,w\rangle \in \overline{A_{TM}} \implies L(M_1) = \emptyset = L(M_2) \implies \langle M_1, M_2\rangle \in EQ_{TM}$
Similarly, $\langle M,w\rangle \notin \overline{A_{TM}} \implies L(M_1) = \emptyset$ but $w \in L(M_2) \implies \langle M_1, M_2\rangle \notin EQ_{TM}$ □

Proposition 4.6.14. *$T = \{\langle M\rangle : M$ is a TM that accepts $w^{\mathcal{R}}$ whenever it accepts $w\}$ is undecidable*

Proof. We will show $A_{TM} \leq_m T$, we construct the computable function F, where on input $\langle M, w\rangle$

1. Construct M' where on input x
   (a) If $x \neq 10, 01$, reject
   (b) If $x = 01$, accept
   (c) If $x = 10$, run M on w, accept if accept
2. Output $\langle M'\rangle$

In this case, we have $\langle M, w\rangle \in A_{TM} \implies L(M') = \{01, 10\} \implies \langle M'\rangle \in T$.
Similarly, $\langle M, w\rangle \notin A_{TM} \implies L(M') = \{01\} \implies \langle M'\rangle \notin T$. □

Proposition 4.6.15. *$B = \{\langle M, w\rangle : M$ is a 2-tape TM and in its computation on w, it writes a non-blank symbol on tape 2$\}$ is undecidable*

Proof. We show $A_{TM} \leq_m B$, we construct a computable function F where on input $\langle M, w\rangle$

1. Construct M' such that on input x
   (a) Run M on w using tape 1
   (b) If it accepts, write 0 on tape 2
2. Output $\langle M', w\rangle$

This gives $\langle M, w\rangle \in A_{TM} \iff M'$ writes a non-blank symbol on tape 2. □

Proposition 4.6.16. *$S = \{\langle M\rangle : M$ is a TM and $L(M) = \{\langle M\rangle\}\}$ is not Turing recognisable and not co-Turing recognisable*

Proof. We first show $A_{TM} \leq_m \overline{S}$, the computable function is constructed as follows, on input $\langle M, w\rangle$

1. Construct M_w such that on input x
   (a) If $x = \langle M_w\rangle$ accept
   (b) Run M on w, accept if accept
2. Return M_w

We have $\langle M, w\rangle \in A_{TM} \iff L(M_w) \neq \{\langle M_w\}$.
We also have $A_{TM} \leq_m S$. □

Proposition 4.6.17. *$J = \{w :$ either $w = 0x$ for some $x \in A_{TM}$ or $w = 1y$ for some $y \notin A_{TM}\}$ is not Turing recognisable and not co-Turing recognisable*

Proof. We first show $A_{TM} \leq_m J$, we construct the computable function as follows, on input $\langle M, w\rangle$, output $0\langle M, w\rangle$.
It is obvious that $\langle M, w\rangle \in A_{TM} \iff f(\langle M, w\rangle) = 0x$ where $x \in A_{TM}$
For $\overline{A_{TM}} \leq_m J$, we construct the computable function as follows, on input $\langle M, w\rangle$, output $1\langle M, w\rangle$.
It is obvious that $\langle M, w\rangle \in \overline{A_{TM}} \iff f(\langle M, w\rangle) = 1x$ where $x \notin A_{TM}$ □

4.7 Reductions via computation histories

Definition 4.7.1. *Let M be a Turing machine and w an input string. An accepting computation history for M on w is a sequence of configurations, $C_1, C_2, \ldots, C_l$, where*

1. *C_1 is the start configuration of M on w*
2. *C_l is an accepting configuration of M*
3. *Each C_i legally follows from C_{i-1} according to the rules of M.*

A rejecting computation history for M on w is defined similarly, except that C_l is a rejecting configuration.

Definition 4.7.2. *A linear bounded automaton is a restricted type of Turing machine wherein the tape head isn't permitted to move off the portion of the tape containing the input. If the machine tries to move its head off either end of the input, the head stays where it is in the same way that the head will not move off the left-hand end of an ordinary Turing machine's tape.*

Definition 4.7.3. $A_{LBA} = \{\langle M, w\rangle : M \text{ is a LBA that accepts } w\}$

Proposition 4.7.4. *A_{LBA} is decidable.*

Proof. We construct a Turing Machine N, on input $\langle M, w\rangle$

1. Simulate M on w for $|Q| \cdot n \cdot g^n + 1$ steps, where n is the length of the tape, $|\Gamma| = g$.
2. Accept if accepted, and reject otherwise.

The total number of configurations is $|Q| \cdot n \cdot g^n$. □

Proposition 4.7.5. *E_{LBA} is undecidable*

Proof. We show $A_{TM} \leq_m \overline{E_{LBA}}$, construct a computable function F such that on input $\langle M, w\rangle$:

1. Construct LBA B such that on input x
   (a) Check if x is an accepting computation history for M on w
2. Output B

We have $\langle M, w\rangle \in A_{TM} \implies \langle B\rangle \notin E_{LBA} \implies \langle B\rangle \neq \emptyset$.
Similarly, $\langle M, w\rangle \notin A_{TM} \implies \langle B\rangle \in E_{LBA} \implies \langle B\rangle = \emptyset$ □

5 Advanced Topics in Computability Theory

5.1 Recursion Theorem

TMs that can inspect their own code (self pointing programs)

Question 5.1.1. 1. *Can you write a program that prints itself.*

2. *Can you write a program such that do the following*
   Print out this sentence
3. *Can you write a program such that do the following*
   Print out the following sentence twice, the second time in quotes:

Lemma 5.1.1. *There is a computable function $q : \Sigma^* \to \Sigma^*$ where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and then stops.*

Proof. We construct Q that represents the computable function q, on input string w

1. Construct the Turing Machine P_w where on any input
   (a) Erase input
   (b) Write w on the tape
   (c) Halt
2. Output $\langle P_w \rangle$

□

Proposition 5.1.2. *There is a Turing machine SELF that ignores its input and prints out a copy of its own description.*

Proof. Suppose $SELF = AB$, we would like to print out $\langle SELF \rangle = \langle AB \rangle$.

1. $A = P_{\langle B \rangle}$
2. $B =$ On input $\langle M \rangle$, where M is a portion of a TM:
   (a) Compute $q(\langle M \rangle)$
   (b) Combine the result with $\langle M \rangle$ to make a complete TM.
   (c) Print the description of this TM and halt.

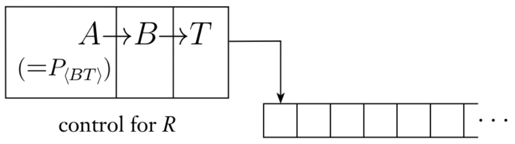
If we now run SELF,

1. First A runs. It prints $\langle B \rangle$ on the tape.
2. B starts. It looks at the tape and finds its input, $\langle B \rangle$.
3. B calculates $q(\langle B \rangle) = \langle A \rangle$ and combines that with $\langle B \rangle$ into a TM description, $\langle SELF \rangle$
4. B prints this description and halts.

□

Theorem 5.1.3 (Recursion Theorem)**.** *Suppse we have a computable map $t : \Sigma^* \times \Sigma^* \to \Sigma^*$ then there is a TM R such that for all $w, R(w) = t(\langle R \rangle, w)$*

Proof. The proof is similar to the construction of SELF. We construct a TM R in three parts, A, B and T, where T is given by the statement of the theorem; a schematic diagram is presented in the following figure.



□

Theorem 5.1.4. *A_{TM} is undecidable*

Proof. If A_{TM} was decidable, let N be a TM that decides it.
Consider $t : \Sigma^* \times \Sigma^* \to \Sigma^*$ as follows. On input $\langle M, w \rangle$:

1. Run N on $\langle M, w \rangle$

2. If N accepts, reject. If N rejects, accept.

By the recursion Theorem, there exists a TM R such that for all w

$$R(w) = t(\langle R \rangle, w) = \begin{cases} REJECT & \text{if } R \text{ accepts} \\ ACCEPT & \text{if } R \text{ rejects} \end{cases}$$

Contradiction. □

Theorem 5.1.5. *$MIN_{TM} = \{\langle M \rangle :$ for every M' with $L(M) = L(M')$ we have $|\langle M' \rangle| \geq |\langle M \rangle|\}$ is not recognisable*

Proof. Suppose MIN_{TM} is recognisable, then there is a enumerator E.
We construct the following TM C, on input w

1. Obtain $\langle C \rangle$ using the Recursion Theorem. Run E until it produces a string $\langle S \rangle$ with $|\langle S \rangle| > |\langle C \rangle|$.
2. Run $S(w)$.

Because MIN_{TM} is infinite, E must contain a TM with a longer description than C. Therefore, step 2 of C eventually terminates with some TM D that is longer than C. Then C simulates D and so is equivalent to it. Because C is shorter than D and is equivalent to it, D cannot be minimal. But $D \in MIN_{TM}$. Thus, we have a contradiction. □

Theorem 5.1.6 (Fixed Point Theorem)**.** *Let $t : \Sigma^* \to \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.*

Proof. Let F be the following Turing machine, on input w:

1. Obtain, via the recursion theorem, own description $\langle F \rangle$
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G
3. Simulate G on w.

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G. □

5.2 Logic

Definition 5.2.1. *In mathematical statements*

- *The symbols $\wedge, \vee, \neg$ are called Boolean operations*
- *$(,)$ are the parentheses*
- *$\forall, \exists$ are called quantifiers*
- *Symbols $x, y, z, \ldots$ are used to denote variables*
- *$R_1, \ldots, R_k$ are called relations.*

A formula is a well-formed string over this alphabet. Particularly, ϕ is a formula if

- *is an atomic formula (in the form of $R_i(x_1, \ldots, x_k)$)*
- *has the form $\phi_1 \wedge \phi_2$ or $\phi_1 \vee \phi_2$ or $\neg\phi_1$, where ϕ_1 and ϕ_2 are smaller formulas*
- *has the form $\exists x, [\phi_1]$ or $\forall x, [\phi_1]$, where ϕ_1 is a smaller formula.*

A quantifier may appear anywhere in a mathematical statement. Its scope is the fragment of the statement appearing within the matched pair of parentheses or brackets following the quantified variable.

We assume that all formulas are in prenex normal form, where all quantifiers appear in the front of the formula. A variable that isn't bound within the scope of a quantifier is called a free variable. A formula with no free variables is called a sentence or statement.

Example 5.2.2. *Among the following examples of formulas, only the last one is a sentence.*

1. $R_1(x_1) \wedge R_2(x_1, x_2, x_3)$
2. $\forall x_1, R_1(x_1) \wedge R_2(x_1, x_2, x_3)$
3. $\forall x_1 \exists x_2 \exists x_3, R_1(x_1) \wedge R_2(x_1, x_2, x_3)$

Definition 5.2.3. *A universe together with an assignment of relations to relation symbols is called a model.*

Formally, we say that a model M is a tuple $(U, P_1, \ldots, P_k)$, where U is the universe and P_1 through P_k are the relations assigned to symbols R_1 through R_k.

If ϕ is a sentence in the language of a model, it is either true or false in that model. If ϕ is true in a model M, we say that M is a model of ϕ.

If M is a model, we let the theory of M, written $Th(M)$, be the collection of true sentences in the language of that model.

Example 5.2.4. *Take $U = \mathbb{N}, P_1 = \leq, R_1$ takes two inputs.*

The sentence, $\forall x, \forall y, R_1(x, y) \vee R_1(y, x)$ is true in the model $M = (\mathbb{N}, \leq) = (U, P_1)$

We will focus only on the universe of the natural numbers.

Definition 5.2.5 (Peano Arithmetic). *Denote $S(x)$ to be the successor of x, namely $x + 1$.*

- $P_1 : \forall x, x \neq 0 \implies \exists y, S(y) = x$
- $P_2 : \forall x, x + 0 = x$
- $P_3 : \forall x, \forall y, s + S(y) = S(x + y)$
- $P_4 : \forall x, x \cdot 0 = 0$
- $P_5 : \forall x, \forall y, x \cdot S(y) = (x \cdot y) + x$
- $P_6 : \forall x, \forall y, S(x) = S(y) \implies x + y \to S$

We also have the induction scheme

$$Ind(A(x)) : \forall y_1, \cdots, \forall y_k [A(0) \wedge \forall x (A(x) \implies A(S(x)) \implies \forall x, A(x)]$$

where A is any formula whose free variables are among $x, y_1, \ldots, y_k$.

We denote $\Gamma_{PA} = \{P_1, P_2, P_3, P_4, P_5, P_6, Induction\ Axioms\}$ as all the axioms for the Peano Arithemetic.

The Peano Arithmetic are all the statements that can be proved from Γ_{PA}. Those are the collections of first order sentences that we observe that $\mathbb{N}$ has.

Theorem 5.2.6. *$Th(N, +)$ is decidable*

Theorem 5.2.7. *$Th(N, +, \times)$ is not decidable.*

Proof. We give a mapping reduction from A_{TM} to $Th(N, +, \times)$. The reduction constructs the formula $\phi_{M,w}$ from the input $\langle M, w \rangle$ by Lemma 5.2.8. Then it outputs the sentence $\exists x \phi_{M,w}$. □

Lemma 5.2.8. *Let M be a Turing machine and w a string. We can construct from M and w a formula $\phi_{M,w}$ in the language of $(N, +, \times)$ that contains a single free variable x, whereby the sentence $\exists x \phi_{M,w}$ is true iff M accepts w.*

Proof. Take x to be the suitable accepting history of M on w. □

Now, we will look at the Godel's Incompleteness Theorem, we need to assume the following two conditions

1. The correctness of a proof of a statement can be checked by machine. Formally, $\{\langle \phi, \pi \rangle | \pi \text{ is a proof of } \phi\}$ is decidable.
2. The system of proofs is sound. That is, if a statement is provable, it is true. We will see that the converse may not hold.

Theorem 5.2.9. *The collection of provable statements in $Th(N, +, \times)$ is Turing-recognizable.*

Proof. We construct the algorithm P, on input ϕ

1. Iterate through each string as a candidate for a proof of ϕ
2. Use the proof checker to check if this is a proof. Accept if yes, go to the next string otherwise.

□

Theorem 5.2.10. *There exists some unprovable true statements in $Th(N, +, \times)$*

Proof. For the sake of contradiction that all true statements are provable.
Consider the following algorithm, on input ϕ

1. Running the algorithm P in on inputs $\phi, \neg\phi$ in parallel. One of these two statements must be true and thus is provable.
2. If P halt on ϕ, return true. If P halt on $\neg\phi$ return false.

This however, contradicts Theorem 5.2.7. □

Theorem 5.2.11. *Let S be a TM that operates as follows, on any input*

1. *Obtain own description $\langle S \rangle$ via the recursion theorem.*
2. *Construct the sentence $\psi = \neg\exists c, \phi_{S,0}$*
3. *Run algorithm P on input ψ.*
4. *If stage 3 accepts, accept.*

We have ψ in stage 2 is unprovable.

Proof. Note that ψ is true iff S doesn't accept 0.
For the sake of contradiction that S finds a proof of ψ, S accepts 0, and ψ would thus be false. A false sentence cannot be provable. Hence, ψ is not provable. Contradiction.
This leaves us the only possibility that S fails to find a proof of ϕ. This means S doesn't accept 0 and ψ is true. Showing that ψ is true but unprovable. □

Question 5.2.1. *Is there an finite set of axioms that characterise $\mathbb{N}$. That is, given a finite set of axioms, we are able to prove all the true results in $\mathbb{N}$.*

Solution. No, we already have infinite number of axioms for the Peano Arithmetic. □

Question 5.2.2. *Is there an infinite set of axioms that characterise $\mathbb{N}$*

Proof. Yes, take the set of all the true statements. It is obviously infinite. □

5.3 Turing Reducibility

Definition 5.3.1. *An oracle for a language B is an external device that is capable of reporting whether any string w is a member of B.*

An oracle Turing machine is a modified Turing machine that has the additional capability of querying an oracle.

We write M^B to describe an oracle Turing machine that has an oracle for language B.

Example 5.3.2. *We define an oracle Turing Machine that decides E_{TM} called $M^{A_{TM}}$, on input $\langle M' \rangle$:*

1. *Construct the following TM N, such that on any input*
   (a) *Run M' in parallel on all strings in Σ^**
   (b) *If M' accepts any of these strings, accept*
2. *Query the oracle to determine whether $\langle N, 0 \rangle \in A_{TM}$*
3. *If the oracle answers no, accept. If yes, reject.*

If M's language isn't empty, N will accept every input, including 0. Hence the oracle will answer yes, and M^{A_TM} will reject. Conversely, if M's language is empty, M^{A_TM} will accept. Thus M^{A_TM} decides E_{TM}. We say that E_{TM} is decidable relative to A_{TM}. That brings us to the definition of Turing reducibility.

Definition 5.3.3. *Language A is Turing reducible to language B, written $A \leq_T B$, if A is decidable relative to B (you can define an oracle Turing Machine on language B that decides A)*

Theorem 5.3.4. *If $A \leq_T B$ and B is decidable, then A is decidable.*

Proof. If B is decidable, then we may replace the oracle for B by an actual Turing Machine that decides B. Thus, we may replace the oracle Turing machine that decides A by an ordinary Turing machine that decides A. □

Theorem 5.3.5. *If $A \leq_m B$ then $A \leq_T B$.*

Proof. We define the following M^B, on input x:

1. Run the reduction to find $f(x)$.
2. Query the oracle to determine whether $f(x) \in B$
3. If the oracle says yes, accept. Otherwise, reject.

If $x \in A, f(x) \in B$ then the oracle says yes and we accept. If $x \notin A, f(x) \notin B$ then the oracle says no and we reject. □

5.4 Kolmogorov complexity

Definition 5.4.1. *$K(x)$ = length of the shortest $\langle M, w \rangle$ that on input w and halts with x on the tape. This is called the Kolmogorov complexity of x. We also called such minimal description as $d(x)$. Conventionally, the strings are in bits.*

Theorem 5.4.2. *There exists a c such that $\forall x K(x) \leq |x| + c$.*

Proof. Let M be a Turing machine that halts as soon as it is started.

We have $\langle M, x \rangle$ is a description of x. Thus, we have $K(x) \leq \langle M, x \rangle = |x| + \langle M \rangle$, where $c = \langle M \rangle$. □

Theorem 5.4.3. *There exists a c such that $\forall x, K(xx) \leq K(x) + c$.*

Proof. Consider the following Turing machine, which on input $\langle N, w\rangle$

1. Run N on w until it halts and produces an output string s
2. Output the string ss

Note that under input $d(x)$, M would halt and have xx on the tape. Hence, the length of description for xx is $|\langle M\rangle| + |d(x)| = |\langle M\rangle| + K(x)$ where we can take $c = |\langle M\rangle|$. □

Definition 5.4.4. *We say p is a description language if this is a computable function $p : \Sigma^* \to \Sigma^*$.*

The minimal description of x with respect to p, written as $d_p(x)$ is some shortest string s such that $p(s) = x$.

Theorem 5.4.5. *For any description language p, a fixed constant c exists that depends only on p where $K(x) \leq K_p(x) + c$ for all x. $K_p(x)$ is the complexity with respective to p.*

Proof. Take any p and consider the following machine M, on input w

1. Output $p(w)$

Then, $\langle\langle M\rangle, d_p(x)\rangle$ is a description of x is at most a fixed constant greater than $K_p(x)$, where we take $c = \langle M\rangle$. □

Definition 5.4.6. *Let x be a string. Say that x is c-compressible if $K(x) \leq |x| - c$.*

If x is not c-compressible, we say that x is incompressible by c.

If x is incompressible by 1, we say that x is incompressible.

Theorem 5.4.7. *Incompressible strings of every length exist.*

Proof. The number of binary strings of length n is 2^n. The number of descriptions of length less than n is at most $\sum_{0\leq i\leq n-1} 2^i = 1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$.

The number of short descriptions is less than the number of strings of length n. Therefore, at least one string of length n is incompressible. □

Corollary 5.4.8. *At least $2^n - 2^{n-c+1} + 1$ strings of length n are incompressible by c.*

Hence, $Pr(K(x) < |x| - c) = \frac{|\{x\in\{0,1\}^n : K(x)<n-c\}|}{|\{0,1\}^n|} < 2^{-c}$

Proof. Every c-compressible string has a description of length at most $n - c - 1$. This gives no more than $2^{n-c} - 1$ such descriptions. Hence, at most $2^{n-c} - 1$ of the strings of length n may have such descriptions. This gives at least $2^n - 2^{n-c} + 1$, are incompressible by c.

This gives $Pr(K(x) < |x| - c) = \frac{2^{n-c}-1}{2^n} < \frac{2^{n-c}}{2^n} = 2^{-c}$. □

Remark 5.4.9. *Random strings are very likely to have high Kolmogorov Complexity.*

Proposition 5.4.10. *There is no computable function that given n, it outputs a string of length n with Kolmogorov complexity $\geq \frac{n}{2}$.*

Proof. Suppose for a contradiction that there is a program A that on input n we are going to give a string with Kolmogorov complexity $\geq \frac{n}{2}$.

Given n, the program outputs some string s_n, where $|s_n| = n$. We also know that $\langle A, n\rangle$ is a description of that string. This shows that $\frac{n}{2} \leq K(s_n) \leq |\langle A, n\rangle| = |\langle A\rangle| + \log n$

Since $|\langle A\rangle|$ is constant, we can take n to be extremely large and obtain a contradiction. □

Corollary 5.4.11. *There is no TM that takes x as input, and outpus $K(x)$. Otherwise, Proposition 5.4.10 is true.*

Theorem 5.4.12 (Existence of Unprovable Statements)**.** *For every binary string x and integer k, we can construct a statement $S_{x,k}$ represents the statement $K(x) \geq k$.*

For every formalisation of mathematics as described above, there is a threshold value t, such that all statements of the form $S_{x,k}$ with $k > t$ are unprovable

Proof. Consider the following algorithm, on input k

```
1. $m = 1$
2. while true
   (a) For all strings $x$ of length at most $m$
       i. For all strings $P$ of length at most m
       ii. If $P$ is a valid proof of $S_{x,k}$, output $x$ and halt
   (b) $m = m + 1$
```

If M is the Turing Machine that implements the above algorithm and $S_{x,k}$ is provable for some x, k, then M would able to find the proof. In this case

$$k \leq K(M(k)) \leq \langle M, k \rangle = \langle M \rangle + \log k$$

This can hold only for finitely many small k. □