511 Notes

Olek Gierczak, Emmett Neyman

Chapter 1

Definition: A *deterministic finite automaton*, or DFA is a 5-tuple $D = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. Σ is a finite set called the alphabet
3. $\delta : Q \times \Sigma \to Q$ is the transition function
4. q_0 is the start state
5. $F \subseteq Q$ is the accepting states

Definition: For $a \in \Sigma$ and $b \in \Sigma^*$, $\delta^*(q, ab) = \delta^*(\delta(q, a), b)$, and $\delta^*(q, a) = \delta(q, a)$.
Definition: $L(D)$ denotes the *language* of D, namely

$$L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

Definition: A language $L \subseteq \Sigma^*$ is regular iff there exists some DFA D such that $L = L(D)$.
Definition: Let A and B be regular languages. Then the union

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

concatenation

$$AB = \{xy \mid x \in A \wedge y \in B\}$$

and star

$$A^* = \{x_1 \dots x_n \mid n \geq 0 \wedge x_i \in A,\ \forall i \in [n]\}$$

are all regular languages.

Definition: A *nondeterministic finite automaton*, or NFA is a 5-tuple $N = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states,
2. Σ is a finite set called the alphabet
3. $\delta : Q \times \Sigma \to P(Q)$ is the transition function
4. q_0 is the start state
5. $F \subseteq Q$ is the accepting states

Definition: For $a \in \Sigma$ and $b \in \Sigma^*$,

$$\delta^*(q, ab) = \bigcup_{p \in \delta(q,a)} \delta^*(p, b)$$

and $\delta^*(q, a) = \delta(q, a)$.
Definition: $L(D)$ denotes the *language* of D, namely

$$L(D) = \{w \in \Sigma^* \mid \exists q \in F,\ q \in \delta^*(q_0, w)\}$$

Theorem: For every NFA N there is a DFA D such that $L(N) = L(D)$.
Proof: By taking a state for each subset of states in Q, and constructing a DFA that transitions as the NFA would given that subset of states.

Definition: A regular expression is defined inductively as follows:

- $a \in \Sigma$
- ϵ
- $\emptyset$
- $R_1 \cup R_2$, R_1, R_2 are regular expressions
- $R_1 R_2$, R_1, R_2 are regular expressions
- R^*, R is regular expressions

Theorem: A language is regular iff some regular expression describes it.
Theorem: **Pumping Lemma** If A is a regular language, then $\exists p$ where if $s \in A$ and $|s| \geq p$, then $s = xyz$ where

- $\forall i \geq 0$, $xy^i z \in A$
- $|y| \geq 1$
- $|xy| \leq p$

Proof: Have a DFA D, set p to be number of states of D. Basically by pigeonhole principle, there is some state that is repeated since $|s| \geq p \geq |Q|$.

Theorem: **Myhill-Nerode** Let x, y be strings. We say x, y are indistinguishable by L if $\forall z$, we have $xz \in L$ iff $yz \in L$, written $x \equiv_L y$. $\equiv_L$ is an equivalence relation. L is regular if and only if $\equiv_L$ has a finite number of equivalence classes.
Proof: Basically finite equivalence classes means finite states, and finite states means finite equivalence classes.

Chapter 3

Definition: A *Turing Machine* is a 7-tuple, $Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where,

1. Q is a finite set called the states,
2. Σ is a finite set not containing the blank symbol $\sqcup$, called the input alphabet
3. Γ is a finite set called the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state
6. q_{accept} is the accept state
7. q_{reject} is the reject state, where $q_{\text{accept}} \neq q_{\text{reject}}$

Definition: A *Configuration* of a Turing Machine T is a three tuple (q, u, v), where q is a state, and the tape contents of T are $uv \in \Gamma^*$. It is written uqv to mean that T has tape contents uv and is in state q with the head pointing at the first character in v.

Definition: Let $C_1 = uaq_ibv$ and $C_2 = uq_jacv$. We say C_1 yield C_2 if $\delta(q_i, b) = (q_j, c, L)$. Similarly, let $C_1 = uaq_ibv$ and $C_2 = uacq_jv$. We say C_1 yield C_2 if $\delta(q_i, b) = (q_j, c, R)$.

Definition: The start configuration of a Turing machine T on input w is $C_s = \epsilon q_0 w$. A rejecting configuration is a configuration where for any u, v, we have $uq_{\text{reject}}v$. An accepting configuration is defined similarly for q_{accept}. The language $L(T)$ is the set of strings for which there are configurations $C_1, \ldots C_n$ where $C_1 = C_s$, C_n is an accept configuration, and $\forall i$ C_i yields C_{i+1}.

Definition: We say a language L is Turing Recognizeable (recursively enumerable) if some Turing Machine T recognizes it. That is, $L = L(T)$.

Definition: We say a language L is Turing Decidable (recursive) if there is a Turing Machine T that decides it. That is T recognizes L and when T is run on an input $w \notin L$, it has configurations $C_1, \ldots C_n$ where $C_1 = C_s$, C_n is a reject configuration, and $\forall i$ C_i yields C_{i+1}.

Definition: A *Multitape Turing Machine* is an ordinary Turing Machine with k tapes instead of one. That is,

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

Theorem: Every Multitape Turing Machine has an equivalent single tape Turing Machine.
Proof: Use a new symbol # as a delimiter to separate the multiple tapes on the single tape. For the k tape head locations, mark off with a dot where the head is in each section of the tape. In other words, virtual tapes and heads. The Turing Machine moves each virtual head one at a time. If a head moves onto a #, the turing machine expands the tape by moving everything to the right one cell and then writing the desired character.

Definition: A nondeterministic Turing Machine is a TM where,

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

Theorem: Every nondeterministic Turing Machine has an equivalent Turing Machine.
Proof: Evaluate the input by simulating the non deterministic Turing Machine in a breadth first manner. DFS does not work since a branch could loop forever.

Definition: An enumerator E is a Turing Machine variant with a work tape and a printer. E starts with a blank input on its work tape and prints out strings one by one, in any order with repetitions allowed. $L(E)$, or the language enumerated by E, is the set of strings that E prints out.
Theorem: A language L is Turing Recognizable iff some enumerator enumerates it.
Proof: Let $s_1, s_2, \ldots$ be an enumeration of the strings in Σ^* (which is countable). If T recognizes L, then for $i = 1, 2, \ldots$, run T for i steps on each input $s_1, \ldots s_i$. For each that accept, print s_i. If E enumerates L, then on input w, just wait until E enumerates w, otherwise just keep enumerating.

Chapter 4

Definition:

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Theorem: A_{DFA} is decidable.
Proof: Simulate B directly.

Theorem: A_{NFA} defined similarly as above is decidable.
Proof: Convert B to a DFA and run the TM for A_{DFA}.

Definition:

$$E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

Theorem: E_{DFA} is decidable.
Proof: Mark states starting from start state that have any transitions. If no accept state is marked, accept, otherwise reject.

Theorem:

$$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$$

is decidable.
Proof: Define C such that

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

$L(C)$ is empty if and only if $L(A) = L(B)$, so just use E_{DFA} to test emptiness of C.

Definition:

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$$

Theorem: A_{TM} is undecidable.
Proof: By diagonalization. Assume for sake of contradiction it is decidable by a TM T. Define D so that on input $\langle M\rangle$, run T on $\langle M, \langle M\rangle\rangle$, then output the opposite of what T outputs. Run D on $\langle D\rangle$. If D accepts then D rejects, and if D rejects, then D accepts, which is a contradiction.

Definition: A language L is co recognizable iff $\overline{L}$ is recognizable.
Theorem: A language L is decidable iff its recognizable by T and co recognizable by M.
Proof: If L is decidable, then L and $\overline{L}$ are both decidable by definition. If T and M recognize and co recognize L respectively, then simultaneously run M and T on w and accept if either T accepts, and reject if M accepts. This terminates since T and M recognize and co-recognize respectively.
Corollary: $\overline{A_{\text{TM}}}$ is not Turing-recognizable.
Proof: A_{TM} is recognizable, and not decidable, so apply theorem above.

Chapter 5

Definition: A problem A is *reducible* to a problem B if we can use a TM that solves B to solve A. In other words, A cannot be harder than B, since we can solve A by solving B.
Theorem: If A is reducible to B, and B is decidable, then A is decidable. If A is undecidable, then B is undecidable.

Definition:

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

Theorem: $HALT_{\text{TM}}$ is undecidable.
Proof: Assume $HALT_{\text{TM}}$ is decidable by a TM R. Then we can use R to solve A_{TM} by determining whether the machine halts on the input, and simulating it if it does, which is a contradiction.

Definition:

$$E_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$$

Theorem: E_{TM} is undecidable.
Proof: Assume for sake of contradiction it is decidable by T. Construct a Turing Machine R that on input $\langle M, w\rangle$ constructs a TM S that rejects all inputs but w, and runs M on w, and finally runs T on S and returns the opposite. This TM decides A_{TM}.

Definition:

$$REGULAR_{\text{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$$

Theorem: $REGULAR_{\text{TM}}$ is undecidable.
Proof: Reduce from A_{TM}. Assume for sake of contradiction $REGULAR_{\text{TM}}$ is decidable by R. S will decide A_{TM} as follows. On input $\langle M, w\rangle$, construct M' such that on input $\langle T, x\rangle$ it accepts all inputs of the form 0^n1^n and otherwise runs M on w and accepts if T accepts, rejects if T rejects. Run R on $\langle M'\rangle$, and accept if accept, reject if reject. If M accepts w, then $L(M') = \Sigma^*$, and if M rejects w, then $L(M')$ is not regular.

Definition:

$$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$$

Theorem: EQ_{TM} is undecidable.
Proof: Assume for sake of contradiction EQ_{TM} is decidable by T. Reduce from E_{TM} by comparing input TM to a TM that rejects all inputs.

Definition: A function $f : \Sigma^* \to \Sigma^*$ is a **computable function** if some Turing Machine M, on every input w, halts with just $f(w)$ on its tape.

Definition: A language A is *mapping reducible* to language B, written $A \leq B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \leftrightarrow f(w) \in B$$

We say f is a reduction from A to B.
Theorem: If $A \leq B$ and B is decidable, then A is decidable. If A is undecidable, then B is undecidable. If B is recognizable, then A is recognizable. If A is not recognizable, then B is not recognizable.

Theorem: EQ_{TM} is not recognizable or co recognizable.
Proof: First, we show A_{TM} is reducible to $\overline{EQ_{\text{TM}}}$. F, on input $\langle M, w\rangle$, constructs M_1 so that on any input, it rejects, and M_2 so that on any input, it runs M on w and returns the result. F outputs $\langle M_1, M_2\rangle$. M_1 has an empty language. M_2 has a non empty language if and only if M accepts w.

Next, we show A_{TM} is reducible to EQ_{TM}. G, on input $\langle M, w\rangle$ constructs M_1 to accept on any input, and M_2 on input, run M on w and return the output. G outputs $\langle M_1, M_2\rangle$. M_1 has the language Σ^*. M_2 has the language Σ^* iff M accepts w.

Theorem: **Rice's Theorem** Let P be any non trivial property of the language of a Turing Machine. That is, $\exists M, N$ such that $M \in P$ and $N \notin P$. Additionally, assume P has the property that if M_1 and M_2 are TMs such that $L(M_1) = L(M_2)$, then $M_1 \in P$ iff $M_2 \in P$. Then P is undecidable.

Chapter 6

Definition: The *minimal description* of x, written as $d(x)$, is the smallest length string $\langle M, w\rangle$ where M halts on input w with x on its tape. If this string is not unique, pick the lexicographically first one. The *Descriptive Complexity* or *Kolmogorov Complexity* of x, written $K(x)$, is $K(x) = |d(x)|$.

Theorem: $\exists c, \forall x,\ K(x) \leq |x| + c$.
Proof: Use the identity function Turing Machine which just halts immediately on every input.

Theorem: $\exists c, \forall x,\ K(xx) \leq K(x) + c$.
Proof: Make a Turing Machine that on input $\langle N, w\rangle$ runs N on w to get the output s, then outputs ss.

Theorem: $\exists c, \forall x, y,\ K(xy) \leq 2K(x) + K(y) + c$
Proof: We construct a TM M as follows. On input w, break down the input into two strings. The first string will be $d(x)$ where each bit is doubled and it is terminated by 01. The second string will be $d(y)$. Once both descriptions are obtained, run each to get x and y and output xy.

Chapter 7

Measuring Complexity

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

Theorem: Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

Proof Idea (full proof p. 282)
The main idea is that we can convert any multitape TM to a single tape TM and then simulate our run over it, with additional steps to "move" to the separate sections of the tape as required. This run time bound is simply the analysis of this conversion. ■

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

$$P = \cup_k TIME(n^k)$$

There are a number of examples of common problems in P, that include $PATH$ (the language of graphs with a path from some s to t), $RELPRIME$ (the language of pairs of numbers whose largest common factor is 1), etc. We now move to another class **NP**. We begin by defining a verifier:

Definition: A **verifier** for a language A is an algorithm V, where

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

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w. A language A is polynomially verifiable if it has a polynomial time verifier.

NP is then the class of languages that have polynomial time verifiers, which leads us to:

Theorem: A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

Proof Idea (full proof p. 294)
We can show how to convert a pollynomial time verifier to an equivalent polynomial time NTM and vice versa - the NTM simulates the verifier by guessing the certificate, and the verifier simulates the NTM by using the accepting branch as the certificate ■

Finally, we can formally define NP as follows:
Definition: **NP** is the class of languages that are decidable in polynomial time on a nondeterministic single tape Turing machine. In other words,

$$NP = \cup_k NTIME(n^k)$$

Some famous and common problems in NP include $CLIQUE$ (the language of all graphs with clique of some size), $SUBSET - SUM$ (the language of sets of numbers that have subsets that sum to some number), etc. This leads to the large open question of whether $P = NP$; clearly $P \subseteq NP$, but nothing else is known yet. Next, we discuss reductions which we define as follows:

Definition: Language A is ***polynomial time reducible*** to language B, written $A \leq_P B$, if a polynomial

time computable function exists, where for every w

$$w \in A \iff f(w) \in B$$

The function f is called the ***polynomial time reduction*** of A to B.

Theorem: If $A \leq_p B$ and $B \in P$, then $A \in P$. (full proof p. 301)

And finally, this leads naturally a definition of $NP - complete$:

Definition: Language A is ***NP-complete*** if 1. A is in NP, and 2. every B in NP is polynomial time reducible to A.

Chapter 8

Space Complexity

Definition: Let M be a deterministic TM that halts on all inputs. The *space complexity* of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of tape cells used by M on any input of size n. If M is a non-deterministic TM, then the *space complexity* $f(n)$ of M is defined to by the maximum number of tape cells that M uses across all of its branches on any input of size n.

Similarly to time complexity, we use asymptotic notation to define *space complexity classes*.

$$\text{SPACE}(n) = \{L \mid L \text{ is a language decided by a } O(f(n)) \text{ deterministic TM}\}$$
$$\text{NSPACE}(n) = \{L \mid L \text{ is a language decided by a } O(f(n)) \text{ non-deterministic TM}\}$$

Examples

- $SAT \in \text{SPACE}(n)$
- $ALL_{\text{NFA}} \in \text{NSPACE}(n)$

Savitch's Theorem

For any function $f : \mathbb{N} \to R^*$ where $f(n) \geq n$, $\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f(n)^2)$.

Proof Idea (full proof p. 334)
The main idea of the proof is to simulate a non-deterministic TM deterministically. We do this by leveraging the divide-and-conquer paradigm. We continually divide up the tableau to see if we can reach c_2 from c_1 in t steps where c_1 and c_2 are configurations of the non-deterministic TM. Since we use recursion, we will make $\log(t)$ recursive calls. We let t be $2^{O(f(n))}$ so that $\log(t) = O(f(n))$. Since we make $O(f(n))$ recursive calls and each recursive call needs to store $O(f(n))$ information, the deterministic TM uses $O(f(n)^2)$ space. ■

$$\text{PSPACE} = \bigcup_{k \in \omega} \text{SPACE}(n^k)$$
$$\text{NPSPACE} = \bigcup_{k \in \omega} \text{NSPACE}(n^k)$$

As a corollary to Savitch's Theorem, we have that PSPACE = NPSPACE. We also see that if a halting deterministic TM runs in $f(n)$ space, it runs in upper-bounded $2^{O(f(n))}$ time. So far, we have that

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE}$$

PSPACE-Completeness

A language L is PSPACE-complete if

1. $L \in$ PSPACE.
2. $\forall A \in$ PSPACE, $A \leq_{\mathrm{P}} L$.

We use polynomial time since we want the power of the reduction to be less than the power of the language and P $\subseteq$ PSPACE. Let $TQBF$ be the set of true boolean formulas in which all variables are bound by quantifiers. WLOG, let these sentences be in prenex normal form. We call these sentences true fully quantified boolean formulas.

$$TQBF = \{\langle\varphi\rangle \mid \varphi \text{ is a true fully quantified boolean formula}\}$$

Theorem $TQBF$ is PSPACE-complete.
Proof Idea (full proof p. 340)
To show that $TQBF \in$ PSPACE, we recursively assign values to the variables and check whether the formula is true under that assignment. To show that every language in PSPACE reduces in polynomial time to $TQBF$, we show how we can turn the TM for a language $A \in$ PSPACE into a fully quantified boolean formula. The proof proceeds similarly to the proof of Savitch's Theorem, recursively dividing the tableau of the machine deciding A and reusing the space during each recursive call.

The following two languages are also PSPACE-complete.
Formula Game
There are two players, Player A and Player E, and a fully quantified boolean formula φ. Player A selects the values for all the universally quantified variables in φ and Player E selects the variables for all the existentially quantified variables in φ. The turn order is determined by the order of the quantifiers in prenex normal form. Player E wins if, after all variables have been assigned, φ is true. Player A wins otherwise.

$$FORMULA - GAME = \{\langle\varphi\rangle \mid \text{Player E has a winning strategy for } \varphi\}$$

$FORMULA - GAME$ is PSPACE-complete because the set of sentences for which Player E has a winning strategy is exactly the set of true fully quantified boolean formulas, that is $FORMULA - GAME = TQBF$.

Generalized Geography
Using a directed graph G and a designated node s, players try to "jump" to a neighboring node, creating a simple path. The first player that cannot jump to a new node loses.

$$GG = \{\langle G, s\rangle \mid \text{Player 1 has a winning strategy on } G \text{ starting from } s\}$$

GG is PSPACE-complete. To show this, we provide a polynomial time reduction from $FORMULA - GAME$ to GG.

The Classes L and NL

By definition, the length of the input is linear. So, how could a Turing Machine possibly use less than linear space. We change our model of computation to a 2-tape TM with a read only input tape and a read/write work tape on which the output will be written. We will only concern ourselves with the space used on the work tape. We need at least $\log(n)$ space since we need that much to keep track of where the machine is on the input tape.

$$\mathrm{L} = \mathrm{SPACE}(\log(n))$$
$$\mathrm{NL} = \mathrm{NSPACE}(\log(n))$$

An example of a language that's in L is $\{0^k 1^k \mid k \geq 0\}$. A TM just needs to count how many 1s and how many 0s are in the input and counting only takes logspace. Languages that contain simple arithmetic are also in L. An example of a language that's in NL is $PATH$. Recall that

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a digraph that has an } s \rightsquigarrow t \text{ path}\}$$

$PATH$ is in NL since a TM can non-deterministically choose each node along the path, needing only logspace to keep track of the current node. Savitch's Theorem can be extended to hold for sublinear space bounds down to $f(n) \geq \log(n)$.

Definition A *logspace transducer* is a TM that has a read-only input tape, a read/write work tape, and a write-only output tape. The head on the input tape can only move left and the work tape can contain only $O(\log(n))$ many symbols. A *logspace transducer* computes a function $f : \Sigma^* \to \Sigma^*$ where $f(w)$ is the string on the output tape when the input tape contains w. Language A is *logspace reducible* to language B, denoted $A \leq_{\mathrm{L}} B$, if A is reducible to B via some logspace function f.

A language B is NL-complete if

1. $B \in$ NL.
2. $\forall A \in$ NL, $A \leq_{\mathrm{L}} B$.

Theorem If $A \leq_{\mathrm{L}} B$ and $B \in$ L, then $A \in$ L.

Theorem $PATH$ is NL-complete.
Proof Idea (full proof p. 353)
We have to show that every language A reduces to $PATH$ in logspace. The idea is to consider the computation of the TM that decides A as a graph. Let all the configurations of the TM on input w be the nodes and try to find a path from the starting configuration to the accepting configuration. This theoream gives that

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP}$$

Theorem NL= coNL.
Proof Idea (full proof p. 356)
We can prove this either by showing both $PATH$ and $\overline{PATH}$ are NL-complete or by showing both $ACYCLIC$ and $\overline{ACYCLIC}$ are NL-complete. The book's proof proceeds by showing that $\overline{PATH}$ is in NL. It does this by describing a non-deterministic TM that decides $\overline{PATH}$ in logspace.

Chapter 9

Hierarchy Theorems

Intuition tells us that if we give a Turing machine more time or more space, it should be able to solve more problems. The **hierarchy theorems** formalize this intuition for both space complexity and time complexity.

Definition A function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is at least $O(\log(n))$, is called *space constructible* if the function that maps the string 1^n to the binary representation of $f(n)$ is computable in space $O(f(n))$. In other words, f is space constructible if some $O(f(n))$ space TM exists that always halts with the binary representation of $f(n)$ as its output when its input was 1^n.

All commonly occurring functions that are at least $O(\log(n))$ are space constructible including $\log_2 n, n, n\log_2 n$, and n^2.

Space Hierarchy Theorem For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ space bit not in $o(f(n))$ space. A corollary of this is that, for all $\epsilon > 0$,

$$\mathrm{SPACE}(n) \subsetneq \mathrm{SPACE}(n^{1+\epsilon})$$

Proof Idea (full proof p. 366)
To prove the theorem, we must show that there is a language A that is decidable in $O(f(n))$ space bit not in $o(f(n))$ space. Unlike other languages we've discussed, A is not describable by a non-algorithmic definition. We will define A via an algorithm D that runs in $O(f(n))$ space. The algorithm D will use diagonalization to ensure that A is different from all languages that are decidable in $o(f(n))$ space. In other words, if M is a Turing machine that decides a language in $o(f(n))$ space, D will make sure A differs from $L(M)$.

Another corollary of the space hierarchy theorem is that NL $\subsetneq$ PSPACE.

Definition A function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is at least $O(n\log(n))$, is called *time constructible* if the function that maps the string 1^n to the binary representation of $t(n)$ is computable in time $O(f(n))$. In other words, t is time constructible if some $O(t(n))$ space TM exists that always halts with the binary representation of $t(n)$ as its output when its input was 1^n.

Like space constructibility, all commonly occurring functions that are at least $O(n\log(n))$ are time constructible including $n\log(n), n\sqrt{n}, n^2$, and 2^n. However, the time hierarchy theorem is slightly weaker than its space counterpart.

Time Hierarchy Theorem For any time constructible function $t : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(t(n))$ time but not in time $o(\frac{t(n)}{\log(t(n))})$. A corollary of this is that, for all $\epsilon \geq 1$,

$$\text{TIME}(n) \subsetneq \text{TIME}(n^{1+\epsilon})$$

Proof Idea (full proof p. 369)
The proof is very similar to the proof of the space hierarchy theorem. The difference comes from the fact that it's harder to keep track of how much time has been used than how much space has been used. In the proof of the space hierarchy theorem, a constant factor overhead was needed. But to prove the time hierarchy theorem, a logarithmic factor overhead is needed. This difference is the reason for the $\frac{1}{\log(t(n))}$ factor in the statement of the theorem.

Relitivization

Definition An *oracle* for a language A is a device that is capable of reporting whether a string w is in A. An *oracle Turing machine* M^A is a Turing machine that has the additional capability of querying the oracle for A. M^A has a special tape called an *oracle tape* such that, whenever M^A writes a string w to it and enters a special "query state", M^A immediately is told whether $w \in A$.
Definition P^A is the class of languages decidable with a polynomial time Turing machine with an oracle for A. We define NP^A similarly.

Example Since SAT is NP-complete, we see that NP $\subseteq \text{P}^{SAT}$ since any language in NP can be reduced in polynomial time to an instance of SAT and then the SAT oracle can be queried to decide the original problem. Similarly, since P^{SAT} is closed under complement, we have that coNP $\subseteq \text{P}^{SAT}$.
Example If P $\neq$ NP, we believe that $\text{P}^{SAT} \subsetneq \text{NP}^{SAT}$. We know that $\overline{MIN-FORMULA} \in \text{NP}^{SAT}$ but we don't think it's in P^{SAT}.

Theorem

1. An oracle A exists such that $\text{P}^A \neq \text{NP}^A$.
2. An oracle B exists such that $\text{P}^B = \text{NP}^B$.

This theorem implies that we probably can't resolve the P versus NP debate using a diagonalization technique.

Proof Idea (full proof p. 378)
Showing that B exists is easy since we can choose B to be a PSPACE-complete language such as $TQBF$. $\text{NP}^{TQBF} = \text{P}^{TQBF}$ since

$$\text{NP}^{TQBF} \subseteq \text{NPSPACE} = \text{PSPACE} = \text{P}^{TQBF} \subseteq \text{NP}^{TQBF}$$

So it remains to show that A exists. We do this by construction rather than by description. We design A such that a certain language $L_A \in \text{NP}^A$ provable requires a brute-force algorithm to decide it. So L_A provably cannot be in P^A, showing that $\text{P}^A \neq \text{NP}^A$. Specifically, we set $L_A = \{w \mid \exists x \in A, |x| = |w|\}$.

Circuit Complexity

A boolean circuit, similar to a boolean formula, is a collection of gates and inputs. Cycles aren't permitted and three kinds of gates are allowed: AND, OR, and NOT. Usually, a circuit has one specified output gate.

Definition A *circuit family* C is an infinite list of circuits, $(C_0, C_1, \ldots)$ where C_n is a circuit with n inputs. We say that C decides a language A if, for every string w where $|w| = n$,

$$C_n(w) = 1 \Leftrightarrow w \in A$$

Definition The *circuit complexity* of a language is the size complexity of a minimum circuit family that decides that language. The *circuit depth complexity* of a language is the depth complexity of a minimum circuit family that decides that language. We denote the class of languages with circuit complexity of $O(f(n))$ as $\text{SIZE}(f(n))$.

Example The circuit that computes the parity function (p. 381) has has $O(n)$ gates. So let A be the language that contains all strings with an odd number of 1s. The circuit complexity of A is $O(n)$.

Definition $\text{P}_{/poly} = \bigcup_{k \in \omega} \text{SIZE}(n^k)$, the class of all languages with poly size circuit families.

Theorem $\text{TIME}(f(n)) \subseteq \text{SIZE}(f(n)^2)$.

Corollary $\text{P} \subseteq \text{P}_{/poly}$.

Proposition Every unary language is in $\text{P}_{/poly}$.
Proof Let A be a unary language. If $1^n \in A$, then let C_n compute AND_n with $O(n)$ many gates. Otherwise, design C_n such that it always outputs 0.

Chapter 10

Approximation Algorithms

Optimization problems are a type of problem in which we want to find the best solution among a collection of solutions, usually this means the minimum or maximum solution. Usually we want to optimize an NP-hard problem, meaning we can't do so efficiently. So, we use approximation algorithms to approximate the optimal solution.

Maximization Given a maximization problem, an α-approximation algorithm, is a poly-time algorithm A such that on any input I, it returns a solution of value $\geq \frac{OPT(I)}{\alpha}$.

Minimization Given a minimization problem, an α-approximation algorithm, is a poly-time algorithm A such that on any input I, it returns a solution of value $\leq \alpha OPT(I)$.

2-Approximation Algorithm for Min Vertex Cover
The following is a 2-approximation algorithm for the min vertex cover problem. On the input $G(V, E)$, let $C = \emptyset$ and let $E' = E$. For every edge (u, v) in E', let $C = C \cup u, v$ and remove from E' any edges incident on u or v. When E' is empty, return C as the min vertex cover.

Probabilistic Algorithms

A probabilistic algorithm is an algorithm that can use the power of coin flips to choose the next steps. It may not seem intuitive, but sometimes using randomness can lead to better approximations than using only deterministic processes.

Definition A *probabilistic Turing machine* M is a type of non-deterministic Turing machine in which each non-deterministic step is called a **coin-flip step** and has two legal next moves. We assign a probability to each branch b of M's computation tree on input w as follows.

$$\Pr(b) = 2^{-k}$$

where k is the number of coin-flip steps on branch b. We say

$$\Pr(M \text{ accepts } w) = \sum_{b \in B_A} \Pr(b)$$

where B_A is the set of accepting branches in M on input w. In other words, the probability that M accepts on w is the probability that we would reach an accepting state during M's computation. We define the probability that M rejects on w to be $1-\Pr(M \text{ accepts } w)$.

Definition For $0 \leq \epsilon \leq \frac{1}{2}$, we say that M decides the language A with *error probability* ϵ if

1. $w \in A$ implies $\Pr(M \text{ accepts } w) \geq 1 - \epsilon$.
2. $w \notin A$ implies $\Pr(M \text{ rejects } w) \geq 1 - \epsilon$

Definition **BPP** is the class of languages that are decidable by a probabilistic polynomial Turing machine with error probability $\frac{1}{3}$. That is, for any $w \in A$, M accepts w with probability $\geq \frac{2}{3}$ and for any $w \notin A$, M accepts w with probability $\leq \frac{1}{3}$.

The **amplification lemma** lets us make this error probability arbitrarily small. Formally, if ϵ is strictly between 0 and $\frac{1}{2}$, then for any polynomial $p(n)$ a probabilistic polynomial Turing machine M that has error probability ϵ has an equivalent probabilistic polynomial Turing Machine M' that has error probability $2^{-p(n)}$.

We know that P $\subseteq$ BPP but we don't know anything about the relationship between BPP and NP.

Definition Let **RP** be the class of languages that are decided by a probabilistic polynomial Turing machine where inputs that are in the language are accepted with a probability of at least $\frac{1}{2}$ and inputs that are not in the language are rejected with probability 1. That is, if we the Turing machine accepts, we know the input must be in the language.

We know that RP $\subseteq$ BPP, P $\subseteq$ RP, and RP $\subseteq$ NP.

Interactive Proofs

We can think of these proofs as proofs by conversation. In this model of computation, we have an all-powerful prover and a weaker verifier. We can use the class NP as an example. The all-powerful prover would find the proof of membership and then the weaker verifier that runs in polynomial time would check the prover's proof. In many interactive proof models, the verifier is a probabilistic Turing machine.

Definition We say that a language A is in **IP**, that is it has an interactive proof, if some polynomial time computable function V exists such that for some arbitrary function P and for every arbitrary function $\widetilde{P}$ and for every string w,

1. $w \in A$ implies $\Pr(V \leftrightarrow P \text{ accepts } w) \geq \frac{2}{3}$
2. $w \notin A$ implies $\Pr(V \leftrightarrow \widetilde{P} \text{ accepts } w) \leq \frac{1}{3}$

Graph Nonisomorphism
Let $NONISO$ be the set of strings $\langle G, H \rangle$ such that G and H are non-isomorphic graphs. We don't know whether or not $NONISO \in$ NP (it's in coNP), however we know there is an interactive proof for the language. Suppose we have two graphs, G and H, that the provers wants to show are non-isomorphic. After being presented with the two graphs, the verifier chooses one of them at random then randomly permutes the nodes of that graph. Then, the verifier presents that graph to the prover and asks it tell the verifier whether that graph was G or H. The prover and verifier can repeat this process an arbitrary number of times to reduce the probability of error.

We often think of the verifier as a function V that computes its next message to the prover from three inputs:

1. Input string: the original input that we are testing for language membership.
2. Random input: we use this input instead of giving the verifier the power of randomness.
3. Partial message history: all the messages sent between the prover and verifier so far in the interactive proof.

The verifier's output is either a message to the prover or its final status: either accept or reject.

Theorem IP = PSPACE
The proof spans nine pages in Sipser (pp. 418-427) and I won't reproduce it here `#sorrynotsorry`

Parallel Computation

Although the PRAM model of computation is one of the most popular theoretical models of parallel computation, we will discuss another model: boolean circuits.

We define the *processor complexity* of a boolean circuit to be its size. So, each gate is a processor and we will allow for polynomial many gates.

We define the *parallel time complexity* of a boolean circuit to be its depth. The goal is to have a depth of $\log^c(n)$.

Definition A family of circuits C is *uniform* if some logspace transducer outputs $\langle C_n \rangle$ when its input is 1^n.

Definition A language has *simultaneous size-depth complexity* $(f(n), g(n))$ if there is a logspace uniform circuit family with size complexity $f(n)$ and depth complexity $g(n)$ that decides the language.

Definition For all $i \geq 1$, the $\mathbf{NC}_i$ be the class of languages that can be decided by a uniform circuit family with polynomial size and $O((\log(n))^i)$ depth. Let **NC** be the class of languages that are in NC_i for some i.

Theorem $\mathrm{NC}_1 \subseteq L$.
Proof Idea We backtrack through the circuit recursively as if it were a tree with the output gate at the root and the inputs at the leaves. We use the logarithmic space we have to keep track of where we are in the traversal and what the value is at the current gate.

Theorem $\mathrm{NL} \subseteq \mathrm{NC}_2$.
Proof Idea (full proof p. 431)
Compute the transitive closure of the graph of possible configurations of the NL Turing machine. Output whether or not there is a path from the starting configuration to the accepting configuration.

Theorem $\mathrm{NC} \subseteq \mathrm{P}$.
Proof A polynomial time algorithm can run the logspace transducer to generate circuit C_n and simulate it on an input of length n.

Definition A language B is **P-complete** if both 1) $B \in \mathrm{P}$ and 2) every language $A \in \mathrm{P}$ reduces to B in logspace. These problems are the hardest problems in P and are thought to be very hard to parallelize. If we can show that a P-complete language is in NC, then we will have shown $\mathrm{P} = \mathrm{NC}$.