# CS 483 - Review guide

Michael Olson and Ryan Stutsman

December 10, 2006

Version 2.2

*Condense soup, not books! – Anonymous*

## Contents

## Preface



### Disclaimer

**This review guide is neither guaranteed nor warranted to be accurate.**

The authors of this review guide explicitly disclaim responsibility for any damage (academic or otherwise) that may result from use of the content contained in this guide.

The contents of this review guide are original (except for the Theorems and Lemmas) and are not known to infringe on any copyright or patent claims. The authors have used ideas from both the course textbook (written by Sipser) and the lecture notes and homework (written by Frederickson).

To the best of our knowledge, this review guide is not endorsed by any Purdue University faculty member.

### Availability

The source code for this review guide is available at

`http://www.mwolson.org/notes/manual/cs483review.tex`

The latest version of this document is available in PDF form at

`http://www.mwolson.org/notes/manual/cs483review.pdf`

Contributions of effort and corrections are welcome, but not mandatory. If you wish to receive updates via email when changes are made, email `mwolson@member.fsf.org`.

### Notation

We use formatting like this: $L, w, f(w)$ when describing any language, string, or function that is used as a "parameter".

We use formatting like this: PSPACE, $\text{HALT}_{TM}$ when describing a class of languages or a particular language.

# 1 Glossary

## 1.1 Uncountable numbers

**diagonalization** Used to determine $|A| = |B|$.

A set is countably infinite if there exists a mapping of each element to $\mathbb{N}$. To demonstrate uncountability, we "enumerate" the set and demonstrate that the enumeration is not complete.

Diagonalization achieves this by creating such an enumeration and then showing an element is missing from the enumeration by showing there is an element in the original set that differs from every other item in the enumeration by at least one of its components.

**Real numbers ($\mathbb{R}$)** An uncountably infinite set.

This is demonstrated by the fact that if a set of real numbers is mapped to $\mathbb{N}$ there always exists some number $\in \mathbb{R}$ for which the first digit does not match the first digit of the first element in the enumeration, the second digit does not match the second digit of the second element in the enumeration, and so on.

## 1.2 Language classes

$\text{NC}^1 \subseteq \text{L} \subseteq \text{NL} \subseteq \text{NC}^2 \subseteq \text{NC} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

None of the above relations have been proven to be strict.

### 1.2.1 Not recognizable

Examples:

- co-ATM ($\overline{ATM}$)

### 1.2.2 Undecidable

**Theorem 1** *Language is decidable $\Leftrightarrow$ it is Turing-recognizable and co-Turing-recognizable.*

Examples:

- $\text{A}_{TM} = \{<M, w> \mid M \text{ is a T.m. and } M \text{ accepts } w\}$.
- $\text{HALT}_{TM} = \{<M, w> \mid M \text{ is a T.m. and } M \text{ halts on input } w\}$.
- $\text{E}_{TM} = \{<M> \mid M \text{ is a T.m. and } \text{L}(M) = \emptyset\}$.
- $\text{EQ}_{TM} = \{<M_1, M_2> \mid M_1 \text{ and } M_2 \text{ are T.m.'s and } \text{L}(M_1) = \text{L}(M_2)\}$
- PCP: Determine whether a collection of dominoes may be arranged such that the string formed by the top of the row of dominoes matches the string formed by the bottom.

### 1.2.3 P

P is the class of languages for which a polynomial time T.m. are known to exist.

**Theorem 2** *Every context-free language is a member of P.*

**Theorem 3** *If $A \leq_p B$ and $B \in P$, then $A \in P$.*

Examples:

- $\text{PATH} = \{<G, s, t> \mid G \text{ is a directed graph that has a directed path from vertex } s \text{ to vertex } t\}$.

### 1.2.4 NP

NP is defined as the class of languages for which a polynomial-time verifier exists. It is more conveniently described as the class of languages for which a non-deterministic polynomial time T.m. are known to exist.

Examples:

- HAMPATH = {<$G$, $s$, $t$> | $G$ is a directed graph with a Hamiltonian path from $s$ to $t$}. A Hamiltonian path includes every node in the graph exactly once, with no repeats. In this case, it begins with node $s$ and ends with node $t$.
- CLIQUE = {<$G$, $k$> | $G$ is an undirected graph with a $k$-clique}. A clique is a component of the graph where every node is connected with every other node. A $k$-clique is a clique of size $k$.
- SUBSET-SUM = {<$S$, $t$> | $S$ is a set of numbers that may be partitioned into two sets that have the same sum, and that sum is $t$}.
- SAT = {<$\phi$> | $\phi$ is a satisfiable Boolean formula}. That is, we can find an assignment of **true** or **false** to each of the variables in $\phi$ such that the formula yields **true**.
- 3SAT = {<$\phi$> | $\phi$ is a satisfiable Boolean formula which is in 3-cnf form}. That is, the formula is grouped into parenthesized groups of three logical OR's, connected by AND's.

**NP-complete**

NP-complete languages are those for which if one particular language in the class was known to be in P, the remainder of the languages in the class would also be in P.

Additionally, if one such language were found, we would know that P = NP.

**Cook-Levin theorem** SAT $\in$ P $\Leftrightarrow$ P = NP.

The proof that SAT is NP-complete uses a tableau to reduce any language $A$ in NP to SAT in polynomial time. If any row of the tableau is an accepting configuration for $A$, $A$ will accept the given input. The top row of the tableau is the starting configuration for $A$.

Examples:

- SAT
- 3SAT
- CLIQUE
- VERTEX-COVER = {<$G$, $k$> | $G$ is an undirected graph that has a $k$-node vertex cover}. That is, there is a some $k$-node subset of the vertices of $G$ for which every node of $G$ touches one of these nodes.
- HAMPATH
- UHAMPATH is a variant of HAMPATH that has an undirected graph, and an undirected path through the graph.
- SUBSET-SUM

### 1.2.5 PSPACE

This is the class of languages that take polynomial (according to the length of the input) amounts of space.

**Savitch's theorem**

$\forall f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geq n$, NSPACE$(f(n)) \subseteq$ SPACE$((f(n))^2)$.

This is proven by constructing an algorithm called CANYIELD. This algorithm takes a nondeterministic T.m. $N$ (implicitly), and the parameters $c_1$, $c_2$, and $t$. It accepts if $N$ can go from configuration $c_1$ to $c_2$ (along any nondeterministic path of $N$) in $t$ or less steps.

The result of this theorem is that PSPACE = NPSPACE.

**PSPACE-complete**

This is the class of languages that is both in PSPACE and for which every language $A$ in PSPACE is polynomial time reducible to it.

Examples:

- TQBF = $\{<\phi> \mid \phi$ is a true fully quantified Boolean formula$\}$. "Fully quantified" means that $\phi$ has all of its quantifiers in front, and its variables and operators following. The formula is "true" iff there is no assignment of values to variables that can make it false.
- FORMULA-game: This is the same as TQBF, but written as a graph-traversal game.
- GG (generalized geography): The generic form of a graph-traversal problem, for which we are given a graph $G$ and a starting node $b$.

### 1.2.6 EXPTIME

This is the class of languages for which an exponential time T.m. exists.

### 1.2.7 L

L is the class of languages that are decidable in log space on a deterministic TM. That is, L = SPACE$(\log n)$.

Log space is a machine that has a read-only input tape, and a read/write work tape where no more than $O(\log n)$ space is used on the work tape.

### 1.2.8 NL

Same as L except L = NSPACE$(\log n)$ (the machine is equipped with non-determinism). Note that space is defined as the maximum amount of space used on any branch of computation.

**NL-complete**

$B$ is NL-complete iff

1. $B \in$ NL, and
2. $\forall A \in$ NL, $A \leq_L B$

   That is, $A$ is log space transducible to $B$.

Log space transduction is performed by some TM with a read-only input tape, a write-only output tape, and only uses $O(\log n)$ space on its read/write work tape.

In actual proof situations, we show $B \in$ NL, then demonstrate $A \leq_L B, A \in$ NL, and then claim since $A \in$ NL that $B \in$ NL as well by the transitivity provided by $\leq_L$. That is, any problem in NL can be solved by $B$ with a log space transduction since $A$ can be solved by $B$ using a log space transduction and any $C \in$ NL could be transduced to $A$ to begin with. Props to transitivity.

Examples:

- PATH is NL-complete.

#### 1.2.9 NC

$\mathrm{NC}^i$, for $i \geq 1$, is the class of languages which may be decided by a uniform family of circuits with polynomial size and $O((\log n)^i)$ depth. NC is the union of all such classes.

### 1.3 Mechanisms

#### 1.3.1 Deterministic finite automaton

This is abbreviated "DFA".

$$\mathrm{G} = (\mathrm{Q}, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$$

#### 1.3.2 Non-deterministic finite automaton

This is abbreviated "NFA".

$$\mathrm{G} = (\mathrm{Q}, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$$

The meaning of the symbols is the same for that of the DFA.

#### 1.3.3 Push-down automaton

This is abbreviated "PDA".

$$\mathrm{G} = (\mathrm{Q}, \Sigma, \Gamma, \delta, \mathrm{q}_0, \mathrm{F})$$

#### 1.3.4 Grammar

There are two main classes of grammar: context-free and context-sensitive.

$$\mathrm{G} = (\mathrm{V}, \Sigma, \mathrm{R}, \mathrm{S})$$

#### 1.3.5 Turing machine

We abbreviate "Turing machine" by "T.m." in the remainder of the text.

#### 1.3.6 Enumerator

This is a device that prints all of the strings in a particular language. Repeating a string is permitted.

It may be defined in terms of a T.m., but that is not covered here.

#### 1.3.7 Oracle

This is a device that can immediately determine (in constant time) whether a given string is part of a language.

### 1.3.8 Oracle Turing machine

This is a T.m. that uses an oracle. It is denoted $M^B$, where $B$ is the language for which an oracle exists.

### 1.3.9 Verifier

This is an algorithm (or rather, a T.m. written in the form of an algorithm), that given a string and a language, accepts iff. the string (called a "certificate") is part of the language. The language is an implied parameter, not an actual one. We usually refer to a specific verifier as "The verifier for $L$ ", where $L$ is a language.

### 1.3.10 Circuits

- Uniform circuit family
- $A \in \text{TIME}(t(n)) \rightarrow A$ has circuit complexity $O(t(n)^2)$
- size-depth circuit complexity

## 1.4 Techniques

### 1.4.1 Mapping reducibility

This is denoted $L \leq_m M$. To understand what this means, we must first define a *computable function*.

**computable function**

A function $f$ that for some T.m. $T$, given an input $w$ for some T.m. $M$, leaves the input $f(w)$ on the tape. $f(w)$ will be composed of the symbols in the input alphabet for $M$. In short,

- No new symbols are produced.
- Some transformation is done on $w$.

Now we can define $L \leq_m M$ as: $L$ is mapping reducible to $M$ if there is a computable function $f$ such that for every $w$ in $A$, $f(w)$ is in $B$. Also, for every $w'$ not in $A$, $f(w')$ is not in $B$. This may be written more succinctly as follows.

$$w \in L \Leftrightarrow f(w) \in M$$

Whenever the textbook states that "$L$ reduces to $M$" or "show a reduction from $L$", they are referring to mapping reducibility, often with an implicit time or space constraint, as discussed in later techniques.

### 1.4.2 Polynomial-time reducibility

This is denoted $L \leq_p M$, and means that $L$ is mapping reducible to $M$, with the added restriction that this reduction must occur in polynomial time.

This is used to prove NP-completeness of a language, or that a language is in P.

### 1.4.3 Turing reducibility

This is denoted $L \leq_T M$, and means that if an oracle for the language $M$ existed, we can write an oracle T.m. that decides language $L$. In other words, we use a "subroutine" for $M$ in the definition of a T.m for $L$.

We can also say that language $L$ is "Turing reducible" to language $M$. If we relate this to programming, we could even say that a routine for $L$ "depends on" a routine for $M$.

Annoyingly, this notation is backwards compared to the direction for mapping reducibility. In mapping reducibility, we construct a function that takes the input from the left side and transforms it into input for

the right side. In Turing reducibility, however, we construct a T.m. for the left side given a T.m for the right side.

We can use Turing reducibility to determine whether a language is "decidable relative to" another language.

**Theorem**

If $L \leq_T M$, and $M$ is decidable, then $L$ is decidable.

**We also know**

If $L \leq_m M$, then $L \leq_T M$. The reverse is not necessarily true.

### 1.4.4 Legal windows

These are groups of cells in a tableau which can represent a valid transition from one configuration to another.

**Warning** This definition may be inaccurate and in need of revision. The authors of this guide do not particularly like this mechanism, and have not given it much effort.

### 1.4.5 Log space reducibility

This is denoted $LL \leq_L M$. and means that $LL$ is mapping reducible to $M$, with the added restriction that this reduction must occur in log space.

This is used to prove NL-completeness of a language, or that a language is in L.

# 2 Problem-solving

## 2.1 Prove language is not regular

We will use the pumping lemma to prove that a language is not regular. First, we assume that the language is regular, and then use the pumping lemma to arrive at a string that *should* be part the language, but isn't. This is a contradiction, so the language must not have been regular to begin with.

**Lemma 1** *If $A$ is regular then $\exists p$ such that if $s \in A, |s| \geq p$ then $s = xyz$ such that*

1. *$\forall i \geq 0, xy^iz \in A$,*
2. *$|y| > 0$, and*
3. *$|xy| \leq p$.*

Note: Just because you can pump a language does not mean that it must be regular.

## 2.2 Find decider for a language

Not yet written.

## 2.3 Proving language is undecidable

General strategy for some language $B$:

1. Suppose $B$ is decidable by some machine, $M$.
2. Show some problem $A$ that is undecidable is reducible (in any time and space) to $B$ by some machine that employs $M$ (usually in some devious manner).
3. Perform obligatory hand waving.
4. Since $A$ is undecidable and we provided a decider for it that employs $M$ our assumption must be garbage.
5. $B$, therefore, in all its glory, is undecidable as well.

Select $A$ from the sweet set mentioned in 1.2.2
e.g. $\text{HALT}_{TM}$ is reducible to $\text{A}_{TM}$, therefore $\text{HALT}_{TM}$ is undecidable.

## 2.4 Prove language is not context-free

We will use an expanded variant of the pumping lemma.

**Lemma 2** *If $A$ is a CFL then $\exists p$ such that if $s \in A, |s| \geq p$ then $s = uvxyz$ such that*

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

Note: Just because you can pump a language does not mean that it must be a CFL.

## 2.5 Proving language is NP-complete

In order to prove that some language $L$ is NP-complete, we must do the following.

1. Show that $L \in$ NP.

   The most popular way to do this is to use nondeterministic guessing of input, coupled with polynomial time verification of our guesses. We could explicitly give an algorithm, but often a simple paragraph suffices.

2. Show that some language known to be NP-complete is polynomial-time reducible to $L$. We'll call this second language $M$. The notation for this is $M \leq_p L$.

   If we wanted to, instead of doing the rest of this proof, we could show that $L$ is NP-hard (in other words, that any language can be reduced using polynomial time to $L$). This is usually more tedious than doing the following, however.

   - This is done by showing a polynomial-time mapping reduction from $L$ to $MM$. Namely, write out a computable function $f$ that takes an instance of $MM$ (or equivalently, the "input" of $MM$) and produces an instance of $L$ (the "input" of $L$), doing this in polynomial time.
   - We must also prove that when an instance/input of $L$ yields **true**[1], the corresponding instance/input for $M$ also yields **true**, and vice versa.

     Start by assuming the input of $L$ produces a **true** result, and then show that $f$ produces input for $M$ which must yield a **true** result.

     Then show the reverse direction – namely, assume the input of $M$ produces a **true** result, and then show that $f$ produces input for $L$ which must yield a **true** result.

3. Since we have shown $MM \leq_p L$, and we know $A \leq_p MM$ for every language $A \in$ NP, $A \leq_p MM \leq_p L$. This completes our proof, since we have previously shown that $L \in$ NP.

## 2.6 Proving language in PSPACE

In order to prove that some language $L$ is in PSPACE, we must do the following.

- Define one or more machines that together decide the language in polynomial space (that is, $O(n^k)$, $k \in \mathbb{N}$).
- Provide analysis of space usage of these machines so that it is determined to be $O(n^k)$ at worst case, specifying $k$.

## 2.7 Proving language in L

In order to prove that some language $LL$ is in $L$, we must provide a machine that decides $LL$ as follows.

- The machine that we make (call it $M$), is provided with a read-only input tape of length $n$, and a work tape that is $O(\log n)$.
- One common method for reducing our space usage is to use counters (numbers that are incremented and decremented) and store them in binary encoding to our work tape. This encoding only takes up $O(\log t)$ space, as opposed to storing an equivalent sequence of symbols which has length $t$.
- We can use time-wasteful algorithms that we would normally pass over, as long as they only use logarithmic space.

[1] In other words, that a machine for $L$ would accept the input.

## 2.8 Proving language is NL-complete

In order to prove that some language $LL$ is NL-complete, we must do the following.

1. Show that $LL \in$ NL.

   The most popular way to do this is to use nondeterministic guessing of input, coupled with log space verification of our guesses. We could explicitly give an algorithm, but often a simple paragraph suffices.

2. Show that some language known to be NL-complete is log-space reducible to $LL$. We'll call this second language $MM$. The notation for this is $MM \leq_L LL$.

   If we wanted to, instead of doing the rest of this proof, we could show that $LL$ is NL-hard (in other words, that any language can be reduced to $LL$ using log space). This is usually more tedious than doing the following, however.

   - This is done by showing a logarithmic-space mapping reduction from $LL$ to $MM$. Namely, write out a computable function[2] $f$ that takes an instance of $MM$ (or equivalently, the "input" of $MM$) and produces an instance of $LL$ (the "input" of $LL$), using log space to do so.
   - We must also prove that when an instance/input of $LL$ yields **true**[3], the corresponding instance/input for $MM$ also yields **true**, and vice versa.

     Start by assuming the input of $LL$ produces a **true** result, and then show that $f$ produces input for $MM$ which must yield a **true** result.

     Then show the reverse direction – namely, assume the input of $MM$ produces a **true** result, and then show that $f$ produces input for $LL$ which must yield a **true** result.

3. Since we have shown $MM \leq_L LL$, and we know $A \leq_L MM$ for every language $A \in$ NL, $A \leq_L MM \leq_L LL$. This completes our proof, since we have previously shown that $LL \in$ NL.

[2] This is actually called a "log space transducer", which is a very strange name for such a simple device.
[3] In other words, that a machine for $LL$ would accept the input.