1 Space Complexity

Last week, we ended by defining the space complexity and nondeterministic space complexity of a language. This week, we begin by proving the following inclusions:

**Theorem 1.** \( \text{dtime}(S(n)) \subseteq \text{space}(S(n)) \subseteq \text{nspace}(S(n)) \subseteq \text{dtime}(2^{O(S(n))}). \)

**Proof** A Turing machine that runs in time \( t \) can access at most \( t \) cells of its work tape, so the first inclusion is trivial. The second inclusion follows directly from the definitions. In order to prove the third inclusion, we’ll need the idea of a configuration graph.

**Definition 2.** The configuration of a Turing machine is data including the contents of the tapes, the position of the heads, and the state of the Turing machine.

Given a Turing machine \( M \) running in space \( S(n) \), and input \( x \) with \( |x| = n \), the number of configurations is at most \( 2^{O(S(n))} \), and each configuration can be represented by a string of bits of length at most \( O(S(n)) \). Given a Turing machine \( M \) and input \( x \), we will denote by \( G_{M,x} \) the configuration graph of \( M \) on \( x \).

**Definition 3.** The configuration graph is a directed graph with vertices corresponding to the configurations of \( M \) and having an edge from \( c \) to \( c' \) if and only if \( M \) can go from \( c \) to \( c' \) in one step.

A configuration graph has two distinguished vertices \( c_{\text{start}} \) corresponding to the starting state with \( x \) written on the input tape, and \( c_{\text{accept}} \) corresponding to \( M \) having a blank work tape and being in the accept state; we force \( M \) to erase its work tape after reaching the accept state. For a deterministic Turing machine, each vertex of \( G_{M,x} \) has out degree at most 1. For a nondeterministic Turing machine, each vertex has out degree at most 2. \( M \) accepts on input \( x \) if and only if there is a directed path in \( G_{M,x} \) from \( c_{\text{start}} \) to \( c_{\text{accept}} \).

For a Turing machine \( M \) that runs in space \( S(n) \) on input \( x \), \( G_{M,x} \) has the following properties:

- \( G_{M,x} \) has at most \( 2^{O(S(n))} \) vertices,
- each vertex can be described by a string of length at most \( S(n) \),
- \( G_{M,x} \) can be constructed in time \( 2^{O(S(n))} \).

In order to construct \( G_{M,x} \), we simply iterate over all possible states of \( M \) and compute the next states. By these properties, we can decide if \( M(x) = 1 \) in time \( 2^{O(S(n))} \) using a breadth first search from \( c_{\text{start}} \). Thus, \( \text{nspace}(S(n)) \subseteq \text{dtime}(2^{O(S(n))}) \). ♦

In addition to the above three properties, we know that there exists a formula \( \varphi \) of size \( O(S(n)) = O(|c| + |c'|) \) such that \( \varphi(c, c') = 1 \) if and only if \( (c, c') \in G_{M,x} \). The proof of this fact follows from ideas very similar to those used in the proof of the Cook-Levin theorem.

There is a space hierarchy theorem, similar to the time hierarchy theorem.

**Theorem 4 (Space hierarchy theorem).** If \( f, g \) are space constructible functions such that \( f(n) = o(g(n)) \), then \( \text{SPACE}(f(n)) \subsetneq \text{SPACE}(g(n)) \).

The proof of this theorem follows from similar ideas to those used in the proof of the time hierarchy theorem. The following are the most basic bounded space complexity classes.
Definition 5.

\[\text{PSPACE} = \bigcup_{c \geq 0} \text{SPACE}(n^c)\]
\[\text{NPSPACE} = \bigcup_{c \geq 0} \text{SPACE}(n^c)\]
\[L = \text{SPACE}(\log n)\]
\[\text{NL} = \text{NSPACE}(\log n)\]

We have the following inclusions among the complexity classes introduced so far:

**Theorem 6.**

\[L \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXP-TIME},\]

and

\[\text{P} \subseteq \text{coNP} \subseteq \text{PSPACE}.\]

With the exception of \(\text{NP} \subseteq \text{PSPACE}\) and \(\text{coNP} \subseteq \text{PSPACE}\), which will be shown below, all of the above inclusions follow directly from arguments already presented.

## 2 P Versus PSPACE

The big question in this section is: Does \(\text{P}\) equal \(\text{PSPACE}\)? In order to address this question, we will define \(\text{PSPACE}\)-hard so that, if a problem that is \(\text{PSPACE}\)-hard could be shown to be in \(\text{P}\), then it would follow that \(\text{P} = \text{PSPACE}\).

**Definition 7.** A language \(L'\) is \(\text{PSPACE}\)-hard if, for all \(L \in \text{PSPACE}\),

\[L \leq_{P} L'.\]

\(L'\) is \(\text{PSPACE}\)-complete if \(L' \in \text{PSPACE}\) and \(L'\) is \(\text{PSPACE}\)-hard.

Recall that \(\leq_{P}\) means, “reduces in polynomial time”.

Our first example of a \(\text{PSPACE}\)-complete language is clearly \(\text{PSPACE}\)-complete, though it’s a bit artificial.

**Definition 8.**

\[\text{SPACE-TM-SAT} = \{ <M,x,1^t> : \text{such that } M \text{ accepts } x \text{ in space } t \}\].

The following more natural \(\text{PSPACE}\)-complete language is a generalization of \text{SAT}.

**Definition 9.** \(\text{QBF}\), or quantified boolean formula, is an expression of the form:

\[Q_1x_1,Q_2x_2,\ldots,Q_nx_n\varphi(x_1,\ldots,x_n),\]

where each \(Q_i\) is either \(\forall\) or \(\exists\), each \(x_i\) are boolean variables, and \(\varphi\) is an unquantified boolean formula.

The form \(Q_1x_1,Q_2x_2,\ldots,Q_nx_n\varphi(x_1,\ldots,x_n)\) is called prenex-normal form; it is not hard to see that every quantified boolean formula can be put in this form in polynomial time.

A \text{QBF} always evaluates to either 0 (false) or 1 (true).

**Definition 10.** \(\text{TQBF}\) is the language of true \text{QBFs}.

**Theorem 11** (Stockmeyer, Meyer, 1973). \(\text{TQBF}\) is \(\text{PSPACE}\)-complete.
First, we'll show that **TQBF** is in **PSPACE**.

**Proof**  Let $\psi$ be an arbitrary QBF on $n$ variables with size $m$. Let $Q_1$ be the first quantifier of $\psi$; either $Q_1 = \forall$ or $Q_1 = \exists$. For a fixed QBF $\varphi$, denote by $\varphi|_{x_1=b}$ the QBF derived from $\varphi$ by fixing each occurrence of $x_1$ to $b$.

Consider the following recursive algorithm, $A$. If $Q_1 = \exists$, output 1 if and only if at least one of $A(\psi|_{x_1=0}) = 1$ or $A(\psi|_{x_1=1}) = 1$. If $Q_1 = \forall$, output 1 if and only if both $A(\psi|_{x_1=0}) = 1$ and $A(\psi|_{x_1=1}) = 1$.

Let $S(n, m)$ denote the space needed by $A$ on input $\psi$ of size $m$ with $n$ variables. Notice that $A$ can use the same space for each recursive call; the only necessary overhead is to keep track of the current formula. Hence,

$$S(n, m) = S(n - 1, m) + \text{space needed to store current formula},$$

$$= S(n - 1, m) + O(m).$$

Solving the recurrence, we get $S(n, m) = O(m^2)$. Therefore, **TQBF** ∈ **PSPACE**. $\blacksquare$

In order to complete the proof that **TQBF** is **PSPACE**-complete, it remains to be shown that **TQBF** is **PSPACE**-hard.

**Proof**  Let $L \in \text{PSPACE}$. We wish to show that $L \leq_p \text{TQBF}$. Since $L \in \text{PSPACE}$, there exists a Turing machine $M$ deciding $L$ in space $s(n)$, where $s = O(n^k)$ for some $k \geq 0$. We know that $x$ is accepted by $M$ if and only if there exists a path from $c_{\text{start}}$ to $c_{\text{accept}}$ in $G_{M,x}$. Recall that $G_{M,x}$ is the configuration graph of $M$ on input $x$. The vertices of $G_{M,x}$ are the possible configurations of $M$, and each configuration has a description using $O(s(n))$ bits. Let $t$ denote the number of bits in the description of a configuration in $G_{M,x}$. Then, the number of vertices in $G_{M,x}$ is at most $2^t$.

We wish to encode the statement “There exists a path from $c_{\text{start}}$ to $c_{\text{accept}}$ in $G_{M,x}$” into an equivalent QBF in polynomial time. One try is to say that such a path exists if $\exists c_1 \exists c_2 \cdots \exists c_k \phi(\cdots)$, where $\phi$ is a boolean expression that says $c_1$ is reachable from $c_{\text{start}}$ in one step, $c_2$ is reachable from $c_1$ in one step, etc. Unfortunately, the path may be of length up to $2^t$, so this expression may have exponential length.

Now, for $c, c' \in G_{M,x}$, let $\psi_0(c, c') = 1$ if $c$ and $c'$ are valid configurations and if there exists a path from $c$ to $c'$ of length at most 1 (and it equals 0 otherwise). Recall that there exists a small formula $\varphi$ that tests this (from Cook-Levin construction). Inductively, define $\psi_i(c, c')$ to be 1 if and only if there exists a path in $G_{M,x}$ from $c$ to $c'$ of length at most $2^i$. Notice that

$$\psi_1(c, c') \equiv \exists c'' \psi_{i-1}(c, c'') \land \psi_{i-1}(c'', c'),$$

since a path from $c$ to $c'$ of length at most $2^i$ must pass through some $c''$ such that there is a path from $c$ to $c''$ of length at most $2^{i-1}$ and there is a path from $c''$ to $c'$ of length at most $2^{i-1}$. Unfortunately, if we expand this construction out, we obtain that $\psi_1$ is exponential in size.

The problem before was that there are two copies of $\psi_{i-1}$ on the right side of the expression for $\psi_i$. Notice, though, that

$$\psi_i \equiv \exists c'' \forall D_1 \forall D_2 ((D_1 = c \land D_2 = c'') \lor (D_1 = c'' \land D_2 = c')) \Rightarrow \psi_{i-1}(D_1, D_2).$$

We can write $\Rightarrow$ and $\lor$ in terms of $\land$, $\forall$, and $\exists$ with a negligible space blow-up. Now, we only have one copy of $\psi_{i-1}$ on the right side, so at each step we add $O(t)$ information. So, we obtain $|\psi_i| \leq |\psi_{i-1}| + O(t)$, so $|\psi_i| \leq O(t^2)$. Since $t^2$ is a polynomial in $t$, we can simply output $\psi_t(c_{\text{start}}, c_{\text{accept}})$, since that has size $O(t^2)$. Hence, we have reduced $L$ to **TQBF** in polynomial time, as required. (Note: technically we need to convert our output into prenex normal form, but this can also be done in polynomial time.) $\blacksquare$

Observe that the preceding proof never uses the fact that the out-degree of each vertex in $G_{M,x}$ is at most 1. Hence, the same proof actually shows that for all $L \in \text{NPSPACE}$, $L \leq_p \text{TQBF}$. This immediately implies that **NPSPACE** = **PSPACE**.

What follows is a simpler argument that **NPSPACE** = **PSPACE**.

3
Theorem 12 (Savitch, 1970). Let \( s : \mathbb{N} \to \mathbb{N} \) be a space-computable function such that \( s(n) \geq \log n \). Then, \( \text{NSPACE}(s(n)) \subseteq \text{SPACE}\left(\left(s(n)^2\right)\right) \).

**Proof**  Let \( M \) be a nondeterministic Turing machine that runs in space \( s(n) \) on input \( x \). We wish to simulate \( M \) using a (deterministic) Turing machine \( M' \) in space \( O(s(n)^2) \). Equivalently, we want \( M' \) to decide if there exists a path from \( c_{\text{start}} \) to \( c_{\text{accept}} \) in \( G_{M,x} \). Again, we will say that each configuration has description of length at most \( t = O(s(n)) \), so we can restrict our search to paths of length at most \( 2^t \).

Let \( s_i \) be the amount of space needed to decide if there exists a path of length at most \( 2^i \) between some two given configurations \( c \) and \( c' \). The Turing machine \( M' \) will enumerate over all intermediate vertices \( v \) and check if there exists a path of length at most \( 2^{i-1} \) from \( c \) to \( v \) and a path of length at most \( 2^{i-1} \) from \( v \) to \( c' \). We see that \( s_i = O(t) = s_{i-1} \) since we need \( O(t) \) space to store the name of vertex \( v \), and we need \( s_{i-1} \) space to check each shorter path, but we can reuse this space to save a factor of 2.

This recurrence implies that \( s_i = O(t^2) = O\left(s(n)^2\right) \), so in space \( O\left(s(n)^2\right) \), \( M' \) can decide whether there exists a path of length at most \( 2^i \) between \( c_{\text{start}} \) and \( c_{\text{accept}} \), as required. \( \blacksquare \)

Note that we do not know how important the square is in Theorem 12. For example, we know that \( L \subseteq \text{NL} \subseteq \left(\log^2 \text{ space}\right) \) by Theorem 12, but it is possible that \( L = \text{NL} \).

## 3 L Versus NL

The big question in this section: Does \( L \) equal \( \text{NL} \)? Recall that we know that \( L \subseteq \text{NL} \subseteq P \), though we don’t know anything about the strength of those containments.

### 3.1 NL-Completeness

To try to understand our main question, we need to define what it means for a language \( \mathcal{L} \) to be \( \text{NL-hard} \).

**First Try.** A language \( \mathcal{L}' \) is \( \text{NL-hard} \) if for all \( \mathcal{L} \in \text{NL} \), \( \mathcal{L} \leq_p \mathcal{L}' \).

This definition will not be useful. Since \( \text{NL} \subseteq P \), every language in \( \text{NL} \) would be \( \text{NL-hard} \). So, this definition adds no meaning to the term “\( \text{NL-hard} \)”. Instead, we need the following definition:

**Definition 13.** A language \( \mathcal{L}' \) is \( \text{NL-hard} \) if for all \( \mathcal{L} \in \text{NL} \), \( \mathcal{L} \leq_L \mathcal{L}' \), where \( \leq_L \) denotes a log-space reduction (which will be defined shortly).

This leads to the following related definition:

**Definition 14.** A language \( \mathcal{L} \) is \( \text{NL-complete} \) if \( \mathcal{L} \in \text{NL} \) and \( \mathcal{L} \) is \( \text{NL-hard} \).

### 3.2 Log-Space Reductions

**Definition 15.** For a function \( f : \{0,1\}^* \to \{0,1\}^* \), we say that \( f \) is (implicitly) log-space-computable if \( f \) is polynomially bounded (that is, there exists \( c \) such that \( |f(x)| \leq |x|^c \) for all \( x \)) and if there exists a log-space Turing machine \( M \) that on input \( (x,i) \) outputs \( f(x)_i \) (the \( i^{\text{th}} \) bit of \( f(x) \)).

Note that the polynomial boundedness of \( f \) is actually implied by the second condition, as we at least need to be able to keep track of which bit we need to return. Equivalently, we could have defined an implicitly log-space-computable function as one that can be computed on a log-space Turing machine \( M \) that has a separate “write-only” output tape where the head only writes bits and can never move to the left. (The head only moves to the right or stays in the same position.)

We can now make the following definition:
**Definition 16.** Let $A$ and $B$ be languages. We say that $A \leq_L B$ if there exists an implicitly log-space-computable function $f$ such that $x \in A$ if and only if $f(x) \in B$.

To reiterate our earlier definitions, a language $L'$ is $\text{NL}$-hard if for all $L \in \text{NL}$, $L \leq_L L'$, and $L'$ is $\text{NL}$-complete if it is, in fact, in $\text{NL}$. We would like to be able to say the following:

**Proposition 17.** Suppose $L$ is $\text{NL}$-hard and $M$ is a log-space Turing machine deciding $L$. Then, $\text{NL} = \text{L}$.

To prove this, we will need the following lemmas:

**Lemma 18.** Let $A$, $B$, and $C$ be languages, and suppose $A \leq_L B$ and $B \leq_L C$. Then $A \leq_L C$.

**Proof.** Since $A \leq_L B$, there exists a function $f$ implicitly log-space-computable by Turing machine $M_f$ such that $x \in A$ if and only if $f(x) \in B$. Since $B \leq_L C$, there exists a function $g$ implicitly log-space-computable by Turing machine $M_g$ such that $y \in B$ if and only if $g(y) \in C$. Notice that $x \in A$ if and only if $g(f(x)) \in C$. So, it would suffice to show that we can compute $g(f(x))$ in log space. Observe that $M_g$ only uses one bit of $f(x)$ at a time. Hence, we can simply run $M_g$, and, whenever it needs a bit of $f(x)$, we can compute that bit by simulating $M_f$, obtain that bit, and then erase the computation of $M_f$ (so we can reuse that space for future runs of $M_f$). The total space usage is the space required by $M_g$ plus the space required by $M_f$. Since both are log-space Turing machines, the result will be a log-space Turing machine that computes $g(f(x))$, as required.

**Lemma 19.** Let $A$ and $B$ be languages such that $A \leq_L B$. If $B \in \text{L}$, then $A \in \text{L}$.

**Proof.** Since $A \leq_L B$, there exists a function $f$ implicitly log-space-computable by Turing machine $M_f$ such that $x \in A$ if and only if $f(x) \in B$. Since $B \in \text{L}$, there exists a log-space Turing machine $M$ deciding it. Let $g$ be the function computed by $M$. By the same procedure used in the proof of Lemma 18, we can compute $g(f(x))$ in logarithmic space. But, the output of $g(f(x))$ is 1 precisely when $x \in A$ (and 0 otherwise). Therefore, $A \in \text{L}$, as required.

We see immediately that Lemma 19 implies Proposition 17, as required.