**Spacereq**(M)

spacereq (M) =

If M is a **deterministic** Turing machine that halts on all inputs, then:

\[ \text{spacereq}(M) = \ f(n) = \text{the maximum number of tape squares that } M \text{ reads on any input of length } n. \]

If M is a **nondeterministic** Turing machine all of whose computational paths halt on all inputs, then:

\[ \text{spacereq}(M) = \ f(n) = \text{the maximum number of tape squares that } M \text{ reads on any path that it executes on any input of length } n. \]
CONNECTED

CONNECTED = \{<G> : G is an undirected graph and G is connected\} can be decided by connected, which starts at G’s first vertex and follows edges, marking vertices as they are visited. If every vertex is eventually marked, then G is connected; otherwise it isn’t. In addition to representing G, connected uses space for:

- Storing the marks on the vertices.
- Maintaining the list L of vertices that have been marked but whose successors have not yet been examined.
- The number marked-vertices-counter, which can be stored in binary in \log(|<G>|) bits.

So \text{spacereq}(\text{connected}) is \mathcal{O}(|<G>|).
SAT

decideSATdeterministically(<w>) =

1. Lexicographically enumerate the rows of the truth table for \( w \). For each row do:
   1. Evaluate \( w \) (by replacing the variables with their values and applying the operators to those values, as described above).
   2. If \( w \) evaluates to \( True \), accept.
2. If no row of the truth table caused \( w \) to be \( True \), reject.

If we use a stack, \( spacereq \) is:

Alternative:
SAT

decideSATdeterministically(<w>) =

1. Consider the rows of the truth table for \( w \). For each row do:
   1. Evaluate \( w \) (by replacing the variables with their values and applying the operators to those values, as described above).
   2. If \( w \) evaluates to \( True \), accept.
2. If no row of the truth table caused \( w \) to be \( True \), reject.

Represent rows of the truth table as binary numbers:

000, 001, 010, …
TSP-DECIDE

• Deciding it nondeterministically:
  • Time:
  • Space:

• Deciding it deterministically:
  • Time:
  • Space:
TSP-DECIDE

decideTSPdeterministically(<G, cost>) =
1. Set circuit to contain just vertex 1.
2. If explore(G, 0, circuit) returns True then accept, else reject.

explore(<G, cost, circuit>) =
1. If circuit is complete with cost is less than cost, return True.
2. If circuit is complete with cost not less than cost, return False.
3. If circuit is not complete then do:
4. For each edge e that is incident on the last vertex of circuit, or until a return statement is executed, do:
   1. If the other vertex of e is not already part of circuit or if it would complete circuit then:
      Call explore(<G, cost + cost of e, circuit with e added>). If the value returned is True then return True.
5. No alternative returned True. So return False.
Relating Time and Space Complexity

**Theorem:** Given a Turing machine $M = (K, \Sigma, \Gamma, \delta, s, H)$ and assuming that $\text{spacereq}(M) \geq n$, the following relationships hold between $M'$ s time and space requirements:

$$\text{spacereq}(M) \leq \text{timereq}(M) \in \mathcal{O}(c^{\text{spacereq}(M)}).$$

**Proof:** $\text{spacereq}(M)$ is bounded by $\text{timereq}(M)$ since $M$ must use at least one time step for every tape square it visits.

Since $M$ halts, the number of steps that it can execute is bounded by $\text{MaxConfigs}(M)$, the number of distinct configurations that it can enter.
Relating Time and Space Complexity

\[ \text{MaxConfigs}(M) = |K| \cdot |\Gamma|^{\text{spacereq}(M)} \cdot \text{spacereq}(M). \]

Let \( c \) be a constant such that \( c > |\Gamma| \). Then:

\[ \text{MaxConfigs}(M) \in \Theta(c^{\text{spacereq}(M)}). \]

Since \( \text{MaxConfigs}(M) \in \Theta(c^{\text{spacereq}(M)}) \) and \( \text{timereq}(M) \leq \text{MaxConfigs}(M) \):

\[ \text{timereq}(M) \in \Theta(c^{\text{spacereq}(M)}). \]
PSPACE and NPSPACE

• **The Class PSPACE:** $L \in \text{PSPACE}$ iff there exists some **deterministic** Turing machine $M$ that decides $L$ and $\text{spacereq}(M) \in \mathcal{O}(n^k)$ for some constant $k$.

• **The Class NPSPACE:** $L \in \text{NPSPACE}$ iff there exists some **nondeterministic** Turing machine $M$ that decides $L$ and $\text{spacereq}(M) \in \mathcal{O}(n^k)$ for some constant $k$.

$\text{PSPACE} = \text{NPSPACE}$
Savitch’s Theorem

**Theorem:** If $L$ can be decided by some nondeterministic Turing machine $M$ and $\text{space}(M) \geq n$, then there exists a deterministic Turing machine $M'$ that also decides $L$ and $\text{space}(M') \in O(\text{space}(M)^2)$.

**Proof:** The proof is by construction of a DTM $M'$ that searches the tree of computations performed by $M$.

**Idea #1:** Depth-first search through the tree. How deep can the stack get?
Proof of Savitch’s Theorem

Idea #2: Divide-and-conquer. Recursively do half the work. Then do the other half, reusing the space.

Each stack entry takes $\text{spacereq}(M)$ space.

The depth of the stack is $\leq \log_2(\text{MaxConfigs}(M))$.

$$\text{MaxConfigs}(M) \in \mathcal{O}(c^{\text{spacereq}(M)}).$$
$$\log_2(\text{MaxConfigs}(M)) \in \mathcal{O}(\text{spacereq}(M)).$$

So the depth of the stack is $\mathcal{O}(\text{spacereq}(M))$ and the total space required is $\mathcal{O}(\text{spacereq}(M)^2)$. 
Proof of Savitch’s Theorem

canreach(\(T, w, c_1, c_2, t\)) =

1. If \(c_1 = c_2\) then return \(True\).
2. If \(t = 1\) then:
   1. If \(c_1 |-_T c_2\) then return \(True\).
   2. Else return \(False\).
3. If \(t > 1\), then let \(Confs\) be the set of all of \(T\)’s configurations whose tape is no longer than \(\text{spacereq}(T)\) applied to \(|w|\). For each configuration \(middle\) in \(Confs\) do:
   If \(\text{canreach}(T, w, c_1, middle, \lfloor t/2 \rfloor)\) and \(\text{canreach}(T, w, middle, c_2, \lfloor t/2 \rfloor)\) then return \(True\).
4. Return \(False\).
Proof of Savitch’s Theorem

1. From $M_{blank}$, build $M_{blank}^\prime$.
2. From $M_{blank}^\prime$, build $M'$, which operates as follows:
   - If $canreach(M_{blank}^\prime, w, c_{start}, c_{accept}, max-on-w)$ accept.
   - Else reject.
PSPACE = NPSPACE

Theorem: PSPACE = NPSPACE.

Proof: We will prove:

• If \( L \) is in PSPACE then it is NPSPACE. (Trivial)

• If \( L \) is in NPSPACE then it is in PSPACE:
PSPACE = NPSPACE

If $L$ is in NPSPACE then there is some NDTM $M$ such that $M$ decides $L$ and $\text{spacereq}(M) \in \mathcal{O}(n^k)$ for some $k$.

If $k \geq 1$, then, by Savitch’s Theorem, there exists a DTM $M'$ such that $M'$ decides $L$ and $\text{spacereq}(M') \in \mathcal{O}(n^{2k})$.

If $k < 1$ then, using the same construction that we used in the proof of Savitch’s Theorem, we can show that there exists a DTM $M'$ such that $M'$ decides $L$ and $\text{spacereq}(M') \in \mathcal{O}(n^2)$.
Theorem: $P \subseteq NP \subseteq PSPACE$.

Proof: We have already shown that $P \subseteq NP$.

Proof that $NP \subseteq PSPACE$: If $L$ is in $NP$, then it is decided by some NDTM $M$ in polynomial time. In polynomial time, $M$ cannot use more than polynomial space since it takes at least one time step to visit a tape square. Since $M$ is an NDTM that decides $L$ in polynomial space, $L$ is in $NPSPACE$. But, by Savitch’s Theorem, $PSPACE = NPSPACE$. So $L$ is also in $PSPACE$. 
PSPACE-Completeness

A language $L$ might have these properties:

1. $L$ is in PSPACE.
2. Every language in PSPACE is deterministic, polynomial-time reducible to $L$.

- $L$ is **PSPACE-hard** iff it possesses property 2.

- $L$ is **PSPACE-complete** iff it possesses both property 1 and property 2.
PSPACE-Completeness, P, and NP

All PSPACE-complete languages can be viewed as being equivalently hard in the sense that all of them can be decided in polynomial space and:

- If any PSPACE-complete language is also in NP, then all of them are and NP = PSPACE.

- If any PSPACE-complete language is also in P, then all of them are and P = NP = PSPACE.
A First PSPACE-Complete Language

• SAT won’t work because it is NP-complete and we suspect that there are PSPACE languages that are not in NP.

• FOL theorem won’t work because it isn’t decidable.

We need a logical language with power in between Boolean logic and full FOL.
Quantified Boolean Expressions

Propositional symbols take on one of two values: True or False.

• The base case: all wffs are QBEs.
• Adding quantifiers: if \( w \) is a QBE that contains the unbound variable \( A \), then the expressions \( \exists A \ (w) \) and \( \forall A \ (w) \) are QBEs.

\[
\begin{align*}
(P \land \neg R) \to S \\
\exists P \ ((P \land \neg R) \to S) \\
\forall R \ (\exists P \ ((P \land \neg R) \to S)) \\
\forall S \ (\forall R \ (\exists P \ ((P \land \neg R) \to S)))
\end{align*}
\]
The Language QBF

A *quantified Boolean formula* is a QBE that is also a sentence (i.e., all of its variables are bound).

- $QBF = \{ \langle w \rangle : w$ is a true quantified Boolean formula $\}$.  
  - $\exists P (\exists R (P \land \neg R)) \in QBF$.  
  - $\exists P (\forall R (P \land \neg R)) \notin QBF$. 

Every element of QBF is in prenex normal form.
QBF is in PSPACE

\[ \text{QBFcheck}(<w>) = \]

1. If \( w \) contains no quantifiers, evaluate it by applying its Boolean operators to its constant values. The result will be either \textbf{True} or \textbf{False}. Return it.

2. If \( w \) is \( \forall v (w') \), where \( w' \) is some quantified Boolean formula, then:
   1. Substitute \textbf{True} for every occurrence of \( v \) in \( w' \) and invoke \textbf{QBFcheck} on the result.
   2. Substitute \textbf{False} for every occurrence of \( v \) in \( w' \) and invoke \textbf{QBFcheck} on the result.
   3. If both of these branches accept, then \( w' \) is true for all values of \( v \). So accept; else reject.

3. If \( w \) is \( \exists v (w') \), where \( w' \) is some quantified Boolean formula, then:
   1. Substitute \textbf{True} for every occurrence of \( v \) in \( w' \) and invoke \textbf{QBFcheck} on the result.
   2. Substitute \textbf{False} for every occurrence of \( v \) in \( w' \) and invoke \textbf{QBFcheck} on the result.
   3. If at least one of these branches accepts, then \( w' \) is true for some value of \( v \). So accept; else reject.
QBF is in PSPACE

\[ \text{QBF\textit{decide}}(\langle w \rangle) = \]

1. Invoke \textit{QBF\textit{check}}(\langle w \rangle).
2. If it returns \textit{True}, accept; else reject.

Analyzing \textit{spacereq}(\textit{QBF\textit{decide}}):

- The depth of \textit{QBF\textit{check}}’s stack:
- The space required for each stack entry:

Thus the total space used by \textit{QBF\textit{decide}} is \( O(|w|) \).
QBF is PSPACE-Complete

- **Idea # 1:** Use the same technique we used to prove the Cook-Levin Theorem.
- Use $\exists$ to capture satisfiability.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>$q_0/\square$</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$q_1/a$</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>$q_1/b$</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>$q_1/b$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>b</td>
<td>$q_2/\square$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>b</td>
<td></td>
<td>$y/\square$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>b</td>
<td></td>
<td></td>
<td>$y/\square$</td>
<td></td>
</tr>
</tbody>
</table>

How many rows in this table?
QBF is PSPACE-Complete

• **Idea # 2:** Use the divide-and-conquer technique we used to prove Savitch’s Theorem and use quantifiers to group multiple assertions into a single one.

Let $f(c_1, c_2, t)$ be true iff $M$ can get from configuration $c_1$ to configuration $c_2$ in at most $t$ steps. Then either:

• $t = 0$.
• $t = 1$.
• $t > 1$. Then there is some configuration we’ll call *middle* with the property that $M$ can get from $c_1$ to *middle* within $\lceil t/2 \rceil$ steps and from *middle* to $c_2$ within another $\lceil t/2 \rceil$ steps. We build $f(c_1, c_2, t)$ using an existential quantifier to assert that *middle* exists.
QBF is PSPACE-Complete

So we have:

\[ f(c_1, c_2, t) = \exists m_1 (\exists m_2 \ldots (f(c_1, \text{middle}, \lfloor t/2 \rfloor) \land f(\text{middle}, c_2, \lfloor t/2 \rfloor)) \ldots ). \]

At each recursive step, we cut the number of computation steps in half but we replace one formula by the conjunction of two. So there’s no net saving of space.

• **Idea # 3:** Use \( \forall \) to group the two subformulas.

So we have:

\[ f(c_1, c_2, t) = \exists \text{middle} (\forall(c_3, c_4) \in \{(c_1, \text{middle}), (\text{middle}, c_2)\} (f(c_3, c_4, \lfloor t/2 \rfloor))). \]
More PSPACE-Hard Problems

- Two-Person Games
- Languages and Automata
Two-Person Games

Consider:

$$\exists X (\forall Y (\exists Z (\forall W (P))))$$
Two-Person Games

• If the length of a game is bounded by some polynomial function of the size of the game, then the game is likely to be PSPACE-complete.

• If the length of the game may grow exponentially with the size of the game, then the game is likely not be solvable in polynomial space. But it is likely to be solvable in exponential time and thus to be EXPTIME-complete.
Languages and Automata

- $\text{NeqNDFSMs} = \{<M_1, M_2> : M_1 \text{ and } M_2 \text{ are nondeterministic FSMs and } L(M_1) \neq L(M_2)\}$

- $\text{NeqREGEX} = \{<E_1, E_2> : E_1 \text{ and } E_2 \text{ are regular expressions and } L(E_1) \neq L(E_2)\}$

Both are PSPACE-COMPLETE.
Languages and Automata

• 2FSMs-INTERSECT = \{<M_1, M_2> : M_1 \text{ and } M_2 \text{ are DFSMs and } L(M_1) \cap L(M_2) \neq \emptyset\}

is in P.

• FSMs-INTERSECT = \{<M_1, M_2, \ldots , M_n> : M_1 \text{ through } M_n \text{ are DFSMs and there exists some string accepted by all of them}\}

is PSPACE-complete.
Languages and Automata

• NOT-SIGMA-STAR = \{<E> : E is a regular expression and \(L(E) \neq \Sigma_E^*\}\}

is PSPACE-complete.

**Definition:** If \(\alpha\) is a regular expression with squaring, then so is \(\alpha^2\). \(L(\alpha^2) = L(\alpha)L(\alpha)\).

• NOT-SIGMA-STAR-SQUARING = \{<E> : E is a regular expression with squaring and \(L(E) \neq \Sigma_E^*\}\}

is not in PSPACE.
Languages and Automata

• CONTEXT-SENSITIVE-MEMBERSHIP =
  \{<G, w> : w \in L(G)\}

is PSPACE-complete.
Sublinear Space Complexity

- Sublinear time complexity doesn’t allow for reading the entire input string.
- But sublinear space complexity makes sense if we consider working space.

So consider Turing machines with two tapes:

- A read-only input tape, and
- A read-write working tape.

Then let $\text{spacereq}(M)$ be the number of visited cells of the read-write (working) tape.
The Classes L and NL

- **The Class L**: $L\# \in L$ iff:
  - there exists some **deterministic** TM $M$ that decides $L\#$, and
  - $\text{spacereq}(M) \in \mathcal{O}(\log n)$.

- **The Class NL**: $L\# \in \text{NL}$ iff:
  - there exists some **nondeterministic** TM $M$ that decides $L\#$, and
  - $\text{spacereq}(M) \in \mathcal{O}(\log n)$.
Why $O(\log n)$?

- Many useful problems can be solved in $O(\log n)$ space. For example:
  - It is enough to remember the length of an input.
  - It is enough to remember a constant number of pointers into the input.
  - It is enough to remember a logarithmic number of Boolean values.
- It is unaffected by some reasonable changes in the way inputs are encoded. For example, it continues not to matter what base, greater than 1, is used for representing numbers.
- Savitch’s Theorem can be extended to cases where $\text{spacereq}(M) \geq \log n$. 
Balanced Parentheses

\[ \text{Bal} = \{ w \in \{ (, ) \} : \text{the parentheses are balanced} \} . \]

Bal is in \( \text{NL} \).

A Turing machine \( M \) that decides it uses its working tape to hold a binary counter. How does it work?
USTCON

USTCON = \{<G, s, t> : G is an undirected graph and there exists an undirected path in G from s to t}\}.

A simple deterministic marking algorithm:
- takes polynomial time
- takes linear space
USTCON

USTCON = \{<G, s, t> : G is an undirected graph and there exists an undirected path in G from s to t}\}.

- Define an NDTM $M$ that searches for a path from $s$ to $t$ but only remembers the most recent vertex on the path. $M$ begins by counting the vertices in $G$ and recording the count (in binary) on its working tape. Then it starts at $s$ and looks for a path. At each step, it nondeterministically chooses an edge from the most recent vertex it has visited to some new vertex. It stores on its working tape the index (in binary) of the new vertex. And it decrements its count by 1. If it ever selects vertex $t$, it halts and accepts. If, on the other hand, its count reaches 0, it halts and rejects. If there is a path from $s$ to $t$, there must be one whose length is no more than the total number of vertices in $G$. So $M$ will find it. So USTCON is in NL.

- It is also possible to show that USTCON is in L.
L and NL

$L \subseteq NL \subseteq PSPACE.$

We know of no languages that are in NL and that can be proven not to be in L. But neither can we prove that $L = NL$. 
NL-Completeness

$L_1$ is log-space reducible to $L_2$ iff there is a deterministic two-tape Turing machine that reduces $L_1$ to $L_2$ and whose working tape uses no more than $O(\log n)$ space.

**The Class NL-hard:** $L#$ is NL-hard iff every language in NL is log-space reducible it.

**The Class NL-complete:** $L#$ is NL-complete iff it is NL-hard and it is in NL.
STCON

STCON = \{<G, s, t> : G is a directed graph and there exists a directed path in G from s to t}\}.

STCON is in NL because it can be decided by almost the same nondeterministic, log-space Turing machine that we described as a way to decide USTCON. The only difference is that now we must consider the direction of the edges that we follow.

We know of no algorithm that shows that STCON is in L. But it is possible to prove that STCON is NL-complete.
The Relationship between $L$ and $P$

**Theorem:** $L \subseteq P$.

**Proof:** Any language in $L$ can be decided by a deterministic Turing machine $M$, where $\text{space}(M) \in O(\log n)$. On an input of length $n$, the maximum number of distinct configurations of $M$ is:

$$n \cdot |\Gamma| \cdot \text{space}(M) \cdot \text{space}(M) \cdot k.$$ 

Since $M$ is deciding a language in $L$, $\text{space}(M) \in O(\log n)$. The number $k$ is independent of $n$. So the maximum number of distinct configurations of $M$ is $O(n \cdot 2 \log n \cdot \log n)$ or $O(n^2 \cdot \log n)$.

Thus $\text{time}(M)$ is also $O(n^2 \cdot \log n)$ and thus $O(n^3)$. 
A Stronger Claim

**Theorem:** NL ⊆ P.

**Proof:** \( \text{STCON} = \{<G, s, t> : G \text{ is a directed graph and there exists a directed path in } G \text{ from } s \text{ to } t \} \) is in P. STCON is also NL-complete, which means that any other language in NL can be reduced to it in deterministic logarithmic space. But any deterministic log-space Turing machine also runs in polynomial time because the number of distinct configurations that it can enter is bounded by a polynomial, as we saw above in the proof that L ⊆ P. So any language in NL can be decided by the composition of two deterministic, polynomial-time Turing machines and thus is in P.
Summarizing

$L \subseteq NL \subseteq P \subseteq PSPACE$
Space Families

- \textit{dspace}(f(n)) = the set of languages that can be decided by some deterministic TM \( M \), where \( \text{spacereq}(M) \in \mathcal{O}(f(n)) \).

- \textit{ndspace}(f(n)) = the set of languages that can be decided by some nondeterministic TM \( M \), where \( \text{spacereq}(M) \in \mathcal{O}(f(n)) \).

- \textit{co-dspace}(f(n)) = the set of languages whose complements can be decided by some deterministic TM \( M \), where \( \text{spacereq}(M) \in \mathcal{O}(f(n)) \).

- \textit{co-ndspace}(f(n)) = the set of languages whose complements can be decided by some nondeterministic TM \( M \), where \( \text{spacereq}(M) \in \mathcal{O}(f(n)) \).
Closure under Complement

**Theorem:** For every function $f(n)$:

$$dspace(f(n)) = co-dspace(f(n)).$$

**Proof:**
Closure under Complement –
the Immerman-Szelepcsényi Theorem

**Theorem:** For every function $f(n) \geq \log n$:

$$\text{ndspace}(f(n)) = \text{co-ndspace}(f(n)).$$

**Proof:** The proof of this claim, that the nondeterministic space complexity classes are closed under complement, was given independently in [Immerman 1988] and [Szelepcsényi 1988].
Closure of the Context-Sensitive Languages under Complement

**Theorem:** The context-sensitive languages are closed under complement.

**Proof:** A language is context-sensitive iff it is accepted by some LBA. So the class of context-sensitive languages is exactly $ndspace(n)$. By the Immerman-Szelepceşényi Theorem:

$$ndspace(n) = co-ndspace(n)$$

So the complement of every context-sensitive language can also be decided by a NDTM that uses linear space (i.e., an LBA). Thus it too is context-sensitive.
Space Hierarchy Theorems

A function $s(n)$ from the positive integers to the positive integers is \textit{space-constructible} iff:

- $s(n) \geq \log n$, and
- the function that maps the unary representation of $n$ (i.e., $1n$) to the binary representation of $s(n)$ can be computed in $O(s(n))$ space.
Deterministic Space Hierarchy Theorem

**Theorem:** For any space-constructible function $s(n)$, there exists a language $L_{s(n)\text{hard}}$ that is decidable in $O(s(n))$ space but that is not decidable in $o(s(n))$ space.

**Proof:** The proof is by diagonalization and is similar to the proof we gave for the Deterministic Time Hierarchy Theorem. The tighter bound in this theorem comes from the fact that it is possible to describe an efficient space-bounded simulator.