# Space Complexity

Slides based on *S.Aurora, B.Barak. Complexity Theory: A Modern Approach.*

Ahto Buldas

Ahto.Buldas@ut.ee

# Motivation

We will study the *memory requirements* of computational tasks.

We define space-bounded computation, which has to be performed by the TM using a restricted number of tape cells, the number being a function of the input size.

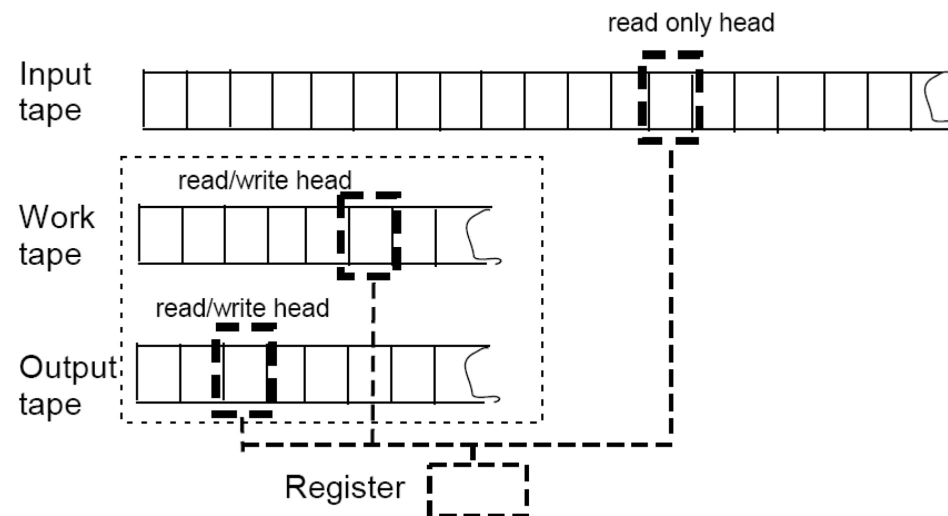
We also study nondeterministic space-bounded TMs.

The goal of introducing a new complexity class is to capture interesting computational phenomena.

One phenomenon that space-bounded computations capture is the computation of *winning strategies* in 2-person games, which seems inherently different from solving $\mathbf{NP}$ problems such as SAT.

# Space-Bounded Computation

*Def.4.1 (Space-bounded computation)*: Let $S\colon \mathbb{N} \to \mathbb{N}$ and $L \subseteq \{0,1\}^*$. We say that $L \in \mathbf{SPACE}(s(n))$ (resp. $L \in \mathbf{NSPACE}(s(n))$) if there is a constant $c$ and TM (resp. NDTM) $M$ deciding $L$ such that on every input $x \in \{0,1\}^*$, the total number of locations that are at some point non-blank during $M$'s execution on $x$ is at most $c \cdot s(|x|)$. (Non-blank locations in the read-only input tape do not count.)



# Some Remarks

Analogous to time complexity, we restrict our attention to space bounds $S : \mathbb{N} \to \mathbb{N}$ that are *space-constructible* functions, i.e. there is a TM that computes $S(n)$ in $O(S(n))$ space when given $1^n$ as input.

If $S$ is space-constructible, then the machine "knows" the space bound it is operating under. For example, $\log n$, $n$ and $2^n$, are space-constructible.

As the work tape is separated from the input tape, it makes sense to consider space-bounded machines that use space $S(n) < n$.

We will assume however that $S(n) > \log n$ since the input tape has length $n$, and we would like the machine to at least be able to remember the index of the cell of the input tape that it is currently reading.

# Relations Between $\mathbf{SPACE}$ and $\mathbf{TIME}$

$\mathbf{DTIME}(S(n)) \subseteq \mathbf{SPACE}(S(n))$ since a TM can access only one tape cell per step.

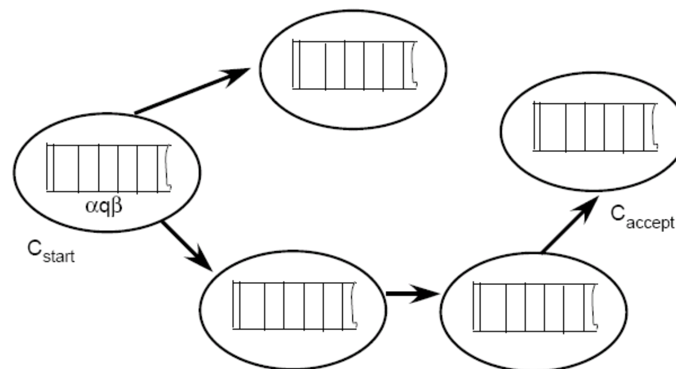
Space can be reused: a cell on the work tape can be overwritten an arbitrary number of times. A space $S(n)$ machine can easily run for as much as $2^{O(S(n))}$ steps—for example a machine that increments a $S(n)$-bit counter.

*Theorem 4.3*: For every space constructible $S : \mathbb{N} \to \mathbb{N}$,

$$\begin{aligned} \mathbf{DTIME}(S(n)) &\subseteq \mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n)) \\ &\subseteq \mathbf{DTIME}\left(2^{O(S(n))}\right) . \end{aligned}$$

# Configuration Graph

A *configuration* of $M$ consists of the contents of all non-blank entries of $M$s working tape, along with its state and head positions on the working and input tapes, at a particular point in its execution. For every $M$ and input $x \in \{0,1\}^*$, the *configuration graph* of $M$ on input $x$, denoted by $G_{M,x}$, is a directed graph whose nodes correspond to possible configurations that $M$ can reach from the starting configuration $C^x_{\mathsf{start}}$. The graph has a directed edge from a configuration $C$ to a configuration $C'$ if $C'$ can be reached from $C$ in one step according to $M$s transition function



# Properties of the Configuration Graph

If $M$ is deterministic then the graph has out-degree one.

If $M$ is non-deterministic then it has an out-degree at most two.

We can assume that $M$'s computation on $x$ does not repeat the same configuration twice (as otherwise it will enter into an infinite loop) and hence that the graph is a directed acyclic graph (DAG).

By modifying $M$ to erase all its work tapes before halting, we can assume that there is only a single configuration $C_{\text{accept}}$ on which $M$ halts and outputs 1. This means that $M$ accepts the input $x$ iff there exists a (directed) path in $G_{M,x}$ from $C_{\text{start}}$ to $C_{\text{accept}}$.

# Properties of the Configuration Graph

*Claim 4.4*: Let $G_{M,x}$ be the configuration graph of a space-$S(n)$ machine $M$ on some input $x$ of length $n$. Then,

- Every vertex in $G_{M,x}$ can be described using $c \cdot S(n)$ bits for some constant $c$ (depending on $M$'s alphabet size and number of tapes) and in particular, $G_{M,x}$ has at most $2^{c \cdot S(n)}$ nodes.
- There is an $O(S(n))$-size CNF formula $\varphi_{M,x}$ such that for every two strings $C, C'$, we have $\varphi_{M,x}(C, C') = 1$ if and only if $C, C'$ encode two neighboring configurations in $G_{M,x}$.

*Proof sketch*: Part 1 follows from observing that a configuration is completely described by giving the contents of all work tapes, the position of the head, and the state that the TM is in. We can encode a configuration by first encoding the snapshot (i.e., state and current symbol read by all

tapes) and then encoding in sequence the non-blank contents of all the work-tape, inserting a special "marker" symbol, to denote the locations of the heads.

Part 2 follows using similar ideas as in the proof of the Cook-Levin theorem. There we showed that deciding whether two configurations are neighboring can be expressed as the AND of many checks, each depending on only a constant number of bits, where such checks can be expressed by constant-sized CNF formulae (by Claim 2.14).

*Proof of Theorem 4.3*: Clearly $\mathbf{SPACE}(S(n)) \subseteq \mathbf{NSPACE}(S(n))$ and so we just need to show $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{DTIME}\left(2^{O(S(n))}\right)$. By enumerating over all possible configurations we can construct the graph $G_{M,x}$ in $2^{O(S(n))}$-time and check whether $C_{\text{start}}$ is connected to $C_{\text{accept}}$ in $G_{M,x}$ using the standard (linear in the size of the graph) breadth-first search algorithm for connectivity.

# Regular Languages

$\mathbf{SPACE}(0)$ – *regular languages* decidable by finite state machines.

*Theorem (Stearns,Hartmanis,Lewis, 1965*)* If $s(n) = o(\log \log n))$ then $\mathbf{SPACE}(s(n))$ contains only regular languages.

*Proof.* We will show that for any $M$ that decides a non-regular language $L_M$ using space $s(n)$, there is $p \in \mathbb{R}$ such that for all $j \in \mathbb{N}$ there is $n_j \in \mathbb{N}$ such that: $s(n_j) \geq j$ and $n_j \leq p^{p^j}$ and hence, there are infinitely many $n_j$ for which

$$s(n_j) \geq j \geq \log_p \log_p n_j = \Omega(\log \log n_j) \ ,$$

and hence we derive a contradiction with $s(n) = o(\log \log n)$.

*J. Hartmanis, P. L. Lewis II, and R. E. Stearns. Hierarchies of memory-limited computations. Proceedings of the 6th Annual IEEE Symposium on Switching Circuit Theory and Logic Design, pp. 179–190. 1965.

We define a *memory configuration* as any combination of state, string of non-blank symbols on the working tape, and the location of the working head on these symbols. Note that the position of the input head is NOT a part of a memory configuration.

If the string of symbols has length strictly less than $j$, we call the memory configuration a *$j$-configuration*. As $|\Gamma|\geq 2$, the number of $j$-configurations is

$$\sum_{i=1}^{j-1} i \cdot |Q| \cdot |\Gamma|^i < j \cdot |Q| \cdot |\Gamma|^j \ .$$

Now we choose $p$ such that for all $j$:

$$p^{p^j} > |\Gamma| \cdot 2^{j \cdot |Q| \cdot |\Gamma|^j} \ .$$

Let $n_j$ be the smallest integer such that $M$ uses at least $j$ working tape cells to process some input word $x = x_1 x_2 \ldots x_{n_j}$ of length $n_j$. The non-regularity of $L_M$ guarantees the existence of such word. Because of $x$, we

must have $s(n_j) \geq j$, and hence, when $M$ finally either accepts or rejects $x$, it is not in a $j$-configuration.

In processing $x$, the machine $M$ may cross and recross many times any particular input cell $x_i$ on its input tape. For each $i \leq n_j$, let $\mathcal{S}_i$ be the set of all memory configurations achieved by $M$ while processing $x$ and while the input head is at the symbol $x_i$.

In order to bound $n_j$, we will first show that $\mathcal{S}_r = \mathcal{S}_s$ for $r < s$ implies that $x_r \neq x_s$. Indeed, assume to the contrary that $\mathcal{S}_r = \mathcal{S}_s$ and $x_r = x_s$ for some $r < s$, and consider what happens if $M$ processes the word

$$x' = x_1 \ldots x_r x_{s+1} \ldots x_{n_j} \ .$$

The length of $x'$ is less than $n_j$ and so all memory configurations that occur in processing $x'$ must be $j$-configurations because of the minimal nature

of $n_j$. We now show that the $j$-configurations for any $x_i$ of $x'$ will be $j$-configurations from the set $\mathcal{S}_i$ (defined for $x$).

Assume to the contrary that that in processing $x'$, the machine $M$ is reading $x_i$ and for the first time enters a $j$-configuration that is not in $\mathcal{S}_i$. This could only occur when the reading head had just crossed from $x_r$ to $x_{s+1}$ or form $x_{s+1}$ to $x_r$. Suppose, the machine is crossing from $x_r$ to $x_{s+1}$. By assumptions, at $x_r$ its configuration is in $\mathcal{S}_r = \mathcal{S}_s$ and hence this configuration could also have occurred at $x_s$ in $x$. Furthermore, since $x_r = x_s$, the machine must go into one of the configurations that occur at $x_s$ in processing $x$.

$$\begin{array}{ccc} x_1 \ldots x_{r-1} & x_r & x_{s+1} \ldots x_{n_j} \\ & \| & \\ x_1 \ldots x_{r-1} & x_s & x_{s+1} \ldots x_{n_j} \end{array}$$

This configuration is by definition in $\mathcal{S}_s$, which proves the assertion that all $j$-configurations occurring from processing $x'$ are $\mathcal{S}_i$-configurations. But this means that $M$ does not stop while processing $x'$ since the only stopping configuration in the processing of $x$ is not a $j$-configuration (because at least $j$ cells are needed and thus the $\mathcal{S}_i$ contain no stopping configuration). Since $M$ must stop for all inputs by definition, this is a contradiction which proves the assertion that $\mathcal{S}_r = \mathcal{S}$ imply $x_r \neq x_s$.

Since $\mathcal{S}_i$ have to be distinct for two occurrences of the same input symbol, the input length $n_j$ must be no greater than the number of subsets of $j$-configurations times the number of input symbols. But this number is less than $|\Gamma| \cdot 2^{j \cdot |Q| \cdot |\Gamma|^j} \leq p^{p^j}$ and hence $n_j \leq p^{p^j}$.

# Some Space Complexity Classes

*Def.4.5*:

$$\begin{aligned} \mathbf{PSPACE} &= \cup_{c>0}\mathbf{SPACE}(n^c) \\ \mathbf{NPSPACE} &= \cup c > 0\mathbf{NSPACE}(n^c) \\ \mathbf{L} &= \mathbf{SPACE}(\log n) \\ \mathbf{NL} &= \mathbf{NSPACE}(logn) \end{aligned}$$

*Example 4.6*: We show how 3SAT $\in \mathbf{PSPACE}$ by describing a TM that decides 3SAT in linear space (that is, $O(n)$ space, where $n$ is the size of the 3SAT instance). The machine just uses the linear space to cycle through all $2^k$ assignments in order, where $k$ is the number of variables. Note that once an assignment has been checked it can be erased from the working tape, and the working tape then reused to check the next assignment. A similar idea of cycling through all potential certificates applies to any $\mathbf{NP}$ language, so in fact $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

*Example 4.7*: Check that the following languages are in $\mathbf{L}$:

$$\begin{aligned} \text{EVEN} &= \{x\colon x \text{ has an even number of 1-s}\} \\ \text{MULT} &= \{(n, m, nm)\colon n, m \in \mathbb{N}\} \ . \end{aligned}$$

*Claim* The following problem PATH is in $\mathbf{NL}$:

PATH $= \{\langle G, s, t\rangle\colon G$ is a directed graph in which there is a path from $s$ to $t\}$ .

*Proof sketch*: A nondeterministic machine can take a nondeterministic walk starting at $s$, always maintaining the index of the vertex it is at, and using nondeterminism to select a neighbor of this vertex to go to next. The machine accepts iff the walk ends at $t$ in at most $n$ steps, where $n$ is the number of nodes. If the nondeterministic walk has run for $n$ steps already

and $t$ has not been encountered, the machine rejects. The work tape only needs to hold $O(\log n)$ bits of information at any step, namely, the number of steps that the walk has run for, and the identity of the current vertex.

Is PATH in $\mathbf{L}$ as well? This is an open problem equivalent to whether or not $\mathbf{L} = \mathbf{NL}$. We will show that PATH is $\mathbf{NL}$-complete. A recent surprising result shows that the restriction of PATH to undirected graphs is in $\mathbf{L}$.

# $\mathbf{PSPACE}$ completeness

As already indicated, we do not know if $\mathbf{P} \neq \mathbf{PSPACE}$, though we strongly believe that the answer is YES. Notice, $\mathbf{P} = \mathbf{PSPACE}$ implies $\mathbf{P} = \mathbf{NP}$. Since complete problems can help capture the essence of a complexity class, we now present some complete problems for $\mathbf{PSPACE}$.

*Def.4.8*: A language $A$ is $\mathbf{PSPACE}$-hard if for every $L \in \mathbf{PSPACE}$, $L \leq_p A$. If in addition $A \in \mathbf{PSPACE}$ then $A$ is $\mathbf{PSPACE}$-complete.

If any $\mathbf{PSPACE}$-complete language is in $\mathbf{P}$ then so is every other language in $\mathbf{PSPACE}$. Viewed contra-positively, if $\mathbf{PSPACE} \neq \mathbf{P}$ then a $\mathbf{PSPACE}$-complete language is not in $\mathbf{P}$. Intuitively, a $\mathbf{PSPACE}$-complete language is the most difficult problem of $\mathbf{PSPACE}$. Just as $\mathbf{NP}$ trivially contains $\mathbf{NP}$-complete problems, so does $\mathbf{PSPACE}$. The following is one:

$$\mathsf{SPACETM} = \{\langle M, w, 1^n \rangle : \text{DTM } M \text{ accepts } w \text{ in space } n\} \ .$$

# True Quantified Boolean Formulae

A quantified boolean formula has the form

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi(x_1, x_2, \ldots, x_n) \ ,$$

where each $Q_i$ is one of the two quantifiers $\forall$ or $\exists$ and $\varphi$ is an (unquantified) boolean formula. Here $\forall$ and $\exists$ quantify over the universe $\{0, 1\}$.

*Example 4.9*: The formula $\forall x \exists y (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is true, because "for every $x \in \{0, 1\}$ there is a $y \in \{0, 1\}$ that is different from it". The formula $\forall x \forall y (x \wedge y) \vee (\overline{x} \wedge \overline{y})$ is false.

*Def.* TQBF is the language that represents all quantified boolean formulae that are true.

# TQBF is $\mathbf{PSPACE}$-complete

First we show that TQBF $\in \mathbf{PSPACE}$. Let

$$\psi = Q_1 x_1 Q_2 x_2 \dots Q_n x_n \varphi(x_1, x_2, \dots, x_n) \tag{1}$$

be a quantified Boolean formula with $n$ variables, where we denote the size of $\varphi$ by $m$. We show a simple recursive algorithm $A$ that can decide the truth of $\psi$ in $O(n+m)$ space. We will solve the slightly more general case where, in addition to variables and their negations, $\varphi$ may also include the constants 0 and 1. If $n = 0$ then the formula contains only constants and can be evaluated in $O(m)$ time and space. Let $n > 0$ and let $\psi$ be as in (1). For $b \in \{0, 1\}$, denote by $\psi|_{x_1=b}$ the modification of $\psi$ where the first quantifier $Q_1$ is dropped and all occurrences of $x_1$ are replaced with the constant $b$. Algorithm $A$ will work as follows: if $Q_1 = \exists$ then output 1 iff at least one of $\psi|_{x_1=0}$ and $\psi|_{x_1=1}$ returns 1. If $Q_1 = \forall$ then output 1 iff both $\psi|_{x_1=0}$ and $\psi|_{x_1=b}$ return 1.

*Space consumption*: Let $s_{n,m}$ denote the space $A$ uses on formulas with $n$ variables and description size $m$. The crucial point is that both recursive computations $\psi|_{x_1=0}$ and $\psi|_{x_1=1}$ can run in the same space. Specifically, after computing $\psi|_{x_1=0}$, the algorithm $A$ needs to retain only the single bit of output from that computation, and can reuse the rest of the space for the computation of $\psi|_{x_1=1}$. Thus, assuming that $A$ uses $O(m)$ space to write $\psi|_{x_1=b}$ for its recursive calls, well get that $s_{n,m} = s_{n-1,m} + O(m)$ yielding $s_{n,m} = O(n \cdot m)$.

This suffices to show TQBF $\in$ $\mathbf{PSPACE}$. Still, we can show that the algorithm runs in $O(m + n)$ space. Note that algorithm always works with restrictions of the same formula $\psi$. So it can keep a global partial assignment array that for each variable $x_i$ will contain either 0, 1 or $q$ (if its quantified and not assigned any value). Algorithm $A$ will use this global space for its operation, where in each call it will find the first quantified

variable, set it to 0 and make the recursive call, then set it to 1 and make the recursive call, and then set it back to $q$. We see that $A$s space usage is given by the equation $s_{n,m} = s_{n-1,m} + O(1)$ and hence it uses $O(n+m)$ space.

**PSPACE**-*hardness*: We show that $L \leq_p$ TQBF for every $L \in \mathbf{PSPACE}$. Let $M$ be a machine that decides $L$ in $S(n)$ space and let $x \in \{0,1\}^n$. We show how to construct a quantified Boolean formula $\psi$ of size $O\left(S(n)^2\right)$ that is true iff $M$ accepts $x$. Recall that (by Claim 4.4), there is a Boolean formula $\varphi_{M,x}$ such that for every two strings $C, C' \in \{0,1\}^m$ (where $m = O(S(n))$ is the number of bits require to encode a configuration of $M$), $\varphi_{M,x}(C, C') = 1$ iff $C$ and $C'$ are valid encodings of two adjacent configurations in the configuration graph $G_{M,x}$. We will use $\varphi_{M,x}$ to come up with a polynomial-sized quantified Boolean formula $\psi'$ that has polynomially many Boolean variables bound by quantifiers and additional $2m$

unquantified Boolean variables $C_1, \ldots, C_m, C'_1, \ldots, C'_m$ (or, equivalently, two variables $C, C'$ over $\{0,1\}^m$) such that for every $C, C' \in \{0,1\}^m$, $\psi(C, C')$ is true iff $C$ has a directed path to $C'$ in $G_{M,x}$. By plugging in the values $C_{\text{start}}$ and $C_{\text{accept}}$ to $\psi'$ we get a quantified Boolean formula $\psi$ that is true iff $M$ accepts $x$.

We define the formula $\psi'$ inductively. We let $\psi_i(C, C')$ be true if and only if there is a path of length at most $2^i$ from $C$ to $C'$ in $G_{M,x}$. Note that $\psi = \psi_m$ and $\psi_0 = \varphi_{M,x}$. The *crucial observation* is that there is a path of length at most $2^i$ from $C$ to $C'$ if and only if there is a configuration $C''$ such that there are paths of length at most $2^{i-1}$ path from $C$ to $C''$ and from $C''$ to $C'$. Thus the following formula suggests itself:

$$\psi_i(C, C') = \exists C'' \colon\; \psi_{i-1}(C, C') \wedge \psi_{i-1}(C'', C) \;.$$

However, this formula is no good. It implies that $\psi_i$ is twice the size of $\psi_{i-1}$, and a simple induction shows that $\psi = \psi_m$ has size about $2^m$, which is too

large. Instead, we use additional quantified variables to save on description size, using the following more succinct definition for $\psi_i(C, C')$:

$$\exists C'' \forall D \forall E : \ (D = C \wedge E = C') \vee (D = C' \wedge E = C'') \ \Rightarrow \ \psi_{i-1}(D, E) \ .$$

Here, $=$ and $\Rightarrow$ are convenient short-hands, and can be replaced by appropriate combinations of the standard Boolean operations $\wedge$ and $\neg$.) Note that $|\psi_i| \leq |\psi_{i-1}| + O(m)$ and hence $|\psi_m| = O(m^2)$. We can convert the final formula to prenex form in polynomial time.

# Savitch's theorem

The reader may notice that because the above proof uses the notion of a configuration graph and does not require this graph to have out-degree one, it actually yields a stronger statement: that TQBF is not just hard for $\mathbf{PSPACE}$ but in fact even for $\mathbf{NPSPACE}$. Since TQBF $\in \mathbf{PSPACE}$, this implies that $\mathbf{PSPACE} = \mathbf{NSPACE}$, which is quite surprising since our intuition is that the corresponding classes for time ($\mathbf{P}$ and $\mathbf{NP}$) are different. In fact, using the ideas of the above proof, one can obtain the following theorem:

*Theorem 4.12 (Savitch 1970)*: For any space-constructible $S: \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$, $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{SPACE}(S(n)^2)$. We remark that the running time of the algorithm obtained from this theorem can be as high as $2^{\Omega(S(n)^2)}$.

*Proof*: (Follows the proof that TQBF is $\mathbf{PSPACE}$-complete. Let $L \in \mathbf{NSPACE}(S(n))$ be a language decided by a TM $M$ such that for every $x \in \{0,1\}^n$, the graph $G_{M,x}$ has at most $N = 2^{O(S(n))}$ vertices. We describe a recursive procedure $\mathsf{REACH}(u,v,i)$ that returns 1 if there is a path from $u$ to $v$ of length at most $2^i$ and 0 otherwise. Note that $\mathsf{REACH}(s,t,\lceil \log N \rceil)$ is 1 iff $t$ is reachable from $s$. Again, the main observation is that there is a path from $u$ to $v$ of length at most $2^i$ iff theres a vertex $z$ with paths from $u$ to $z$ and from $z$ to $v$ of lengths at most $2^{i-1}$. Thus, $\mathsf{REACH}(u,v,i)$ will enumerate over all vertices $z$ (at a cost of $O(\log N)$ space) and output 1 if it finds $z$ such that $\mathsf{REACH}(u,z,i-1) = 1$ and $\mathsf{REACH}(z,v,i-1) = 1$. Once again, the crucial observation is that although the algorithm makes $n$ recursive invocations, it can reuse the space in each of these invocations. Thus, if we let $s_{N,i}$ be the space complexity of $\mathsf{REACH}(u,v,i)$ on an $N$-vertex graph we get that $s_{N,i} = s_{N,i-1} + O(\log N)$ and thus $s_{N,\log N} = O(\log^2 N) = O(S(n)^2)$.

# PSPACE: Optimum Strategies for Games

The central feature of **NP**-complete problems is that a yes answer has a *short certificate*. The analogous concept for **PSPACE**-complete problems is that of a *winning strategy* for a two-player game with perfect information.

A good example of such a game is Chess: two players alternately make moves, and the moves are made on a board visible to both. Thus moves have no hidden side effects; hence the term "perfect information".

What does it mean for a player to have a "winning strategy?" The first player has a winning strategy iff there is a 1st move for player 1 such that for every possible 1st move of player 2 there is a 2nd move of player 1 such that ...(and so on) such that at the end player 1 wins.

Thus deciding whether or not the first player has a winning strategy seems to require searching the tree of all possible moves. This is reminiscent of $\mathbf{NP}$, for which we also seem to require exponential search.

But the crucial difference is the lack of a “short certificate”, since the only certificate we can think of is the winning strategy itself, which as noticed, requires exponentially many bits to even describe. Thus we seem to be dealing with a fundamentally different phenomenon than the one captured by $\mathbf{NP}$.

# The QBF Game

The interplay of existential and universal quantifiers in the description of the the winning strategy motivates us to invent the following game:

*Example 4.13 (The QBF game)*: The "board" for the QBF game is a Boolean formula $\varphi$ whose free variables are $x_1, x_2, \dots, x_{2n}$. The two players alternately make moves, which involve picking values for $x_1, x_2, \dots$ in order. Thus player 1 will pick values for the odd-numbered variables $x_1, x_3, x_5, \dots$ (in that order) and player 2 will pick values for the even-numbered variables $x_2, x_4, x_6, \dots$. We say player 1 wins iff at the end $\varphi$ becomes true. Clearly, player 1 has a winning strategy iff

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots \forall x_{2n} \colon \varphi(x_1, x_2, \dots, x_{2n}) \ ,$$

namely, iff this quantified boolean formula is true. Thus deciding whether player 1 has a winning strategy for a given board in the QBF game is $\mathbf{PSPACE}$-complete.

# $\mathbf{NL}$ completeness

We cannot use $\leq_p$ since $\mathbf{NL} \subseteq \mathbf{P}$ and every language in $\mathbf{NL}$ is Karp reducible to the trivial language $\{1\}$. Reduction: "decide in poly-time whether the input is in the $\mathbf{NL}$-language, and then map to 1 or 0 accordingly". Such trivial languages should not be the "hardest" languages of $\mathbf{NL}$.

When choosing the type of reduction to define completeness for a complexity class, we must keep in mind the complexity phenomenon we seek to understand. In this case, the complexity question is whether or not $\mathbf{NL} = \mathbf{L}$. The reduction should not be more powerful than $\mathbf{L}$. For this reason we use *log-space reductions*. To define such reductions we must tackle the tricky issue that a reduction typically maps instances of size $n$ to instances of size at least $n$, and so a log-space machine computing such a reduction does not have even the memory to write down its output!

# Log-Space Reductions

The way out is to require that the reduction should be able to compute *any desired bit* of the output in logarithmic space. The formal definition is as follows:

*Definition 4.14 (log-space reduction)*: Let $f\colon \{0,1\}^* \to \{0,1\}^*$ be a polynomially-bounded function (i.e., theres a constant $c > 0$ such that $f(x) \leq |x|^c$ for every $x \in \{0,1\}^*$). We say that $f$ is *implicitly log-space computable*, if the following two languages are in $\mathbf{L}$:

$$L_f = \{\langle x, i\rangle\colon\ f(x)_i = 1\} \quad \text{and} \quad L'_f = \{\langle x, i\rangle\colon\ i \leq |f(x)|\}\ .$$

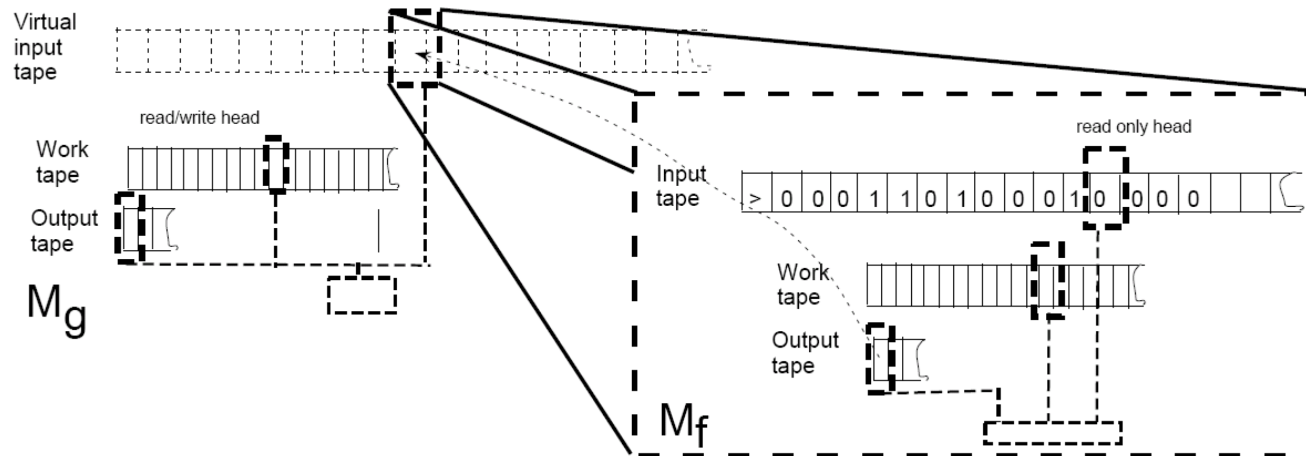
Informally, we can think of a single $O(\log|x|)$-space machine that given input $(x, i)$ outputs $f(x)_i$ provided that $i \leq |f(x)|$. Language $A$ is *log-space reducible* to language $B$, denoted by $A \leq_l B$, if there is an implicitly log-space computable function $f\colon \{0,1\}^* \to \{0,1\}^*$ such that $x \in A$ iff $f(x) \in B$ for every $x \in \{0,1\}^*$.

# Properties of Log-Space Reductions

*Lemma 4.15*: (a) If $A \leq_l B$ and $B \leq_l C$ then $A \leq_l C$. (b) If $A \leq_l B$ and $B \in \mathbf{L}$ then $A \in \mathbf{L}$.

*Proof*: We prove that if $f, g$ are implicitly log-space computable, then so is $h(x) = g(f(x))$. Then part (b) follows by letting $g$ be the characteristic function of $B$,i.e. $g(y) = 1$ iff $y \in B$.

Let $M_f, M_g$ be the log-space machines that compute the mappings $x, i \mapsto f(x)_i$ and $y, j \mapsto g(y)_j$ respectively. We construct $M_h$ that computes the mapping $x, j \mapsto g(f(x))_j$, in other words, given as input $(x, j)$ outputs $g(f(x))_j$ provided that $j \leq |g(f(x))|$. Machine $M_h$ will pretend that it has an additional (fictitious) input tape on which $f(x)$ is written, and it is simulating $M_g$ on this input.




The true input tape has $x, j$ written on it. To maintain its fiction, $M_h$ holds on its work-tape the position $i$ of the "head" on the fictitious tape that $M_g$ is currently reading; this requires only $\log |f(x)|$ space. To make one step, $M_g$ needs to know the "input symbol", i.e. $f(x)_i$. At this point, $M_h$ temporarily suspends its simulation of $M_g$ (copying $M_g$-s work-tape to a safe place on its own work-tape) and invokes $M_f$ on inputs $x, i$ to get $f(x)_i$. Then it resumes its simulation of $M_g$ using this bit. The total space $M_h$ uses is $O(\log |g(f(x))|) + O(\log |x|) + O(\log |f(x)|) = O(\log |x|)$.

# $\mathbf{NL}$-completeness

A language $A$ is $\mathbf{NL}$-complete if $A \in \mathbf{NL}$ and $A \leq_l B$ for every $B \in \mathbf{NL}$. Note that an $\mathbf{NL}$-complete language is in $\mathbf{L}$ iff $\mathbf{NL} = \mathbf{L}$.

*Theorem 4.16*: PATH is $\mathbf{NL}$-complete.
*Proof*: We already have PATH $\in \mathbf{NL}$. Let $L \in \mathbf{NL}$ and $M$ decides it in space $O(\log n)$. We describe an implicitly log-space computable $f$ that reduces $L$ to PATH. For any $x \in \{0, 1\}^n$, define $f(x)$ as a triple $\langle G_{M,x}, C_{\mathsf{start}}, C_{\mathsf{accept}} \rangle$. There is a path from $C_{\mathsf{start}}$ to $C_{\mathsf{accept}}$ iff $M$ accepts $x$. The graph is represented by an adjacency matrix that contain 1 in the $\langle C, C' \rangle$-th position (i.e. rows and columns are numbers between 0 and $2^{O(\log n)}$) iff there is an edge from $C$ to $C'$ in $G_{M,x}$. It remains to show that this adjacency matrix can be computed by a log-space reduction. This is easy since given $\langle C, C' \rangle$, a deterministic machine can examine in space $O(|C| + |C'|) = O(\log |x|)$ whether $C'$ is one of the (at most two) configurations that can follow $C$ according to the transition function of $M$.