Jeffrey Shallit
School of Computer Science
University of Waterloo
shallit@uwaterloo.ca
https://cs.uwaterloo.ca/~shallit
Up until now, in our discussion of complexity, we have only been concerned with time.

But time is not the only scarce resource that a computer uses.

Another one is space.

So we need to define the space used by a Turing machine.
We define the *space used* by a deterministic always-halting Turing machine $M$ to be the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that $M$ scans when processing an input $w$, over all inputs $w$ of length $n$.

For nondeterministic machines the definition is similar, except that now this maximum is over *all possible nondeterministic paths* during a computation *and* over all inputs $w$ of length $n$. 
Example of space usage

Recall our TM to decide the language

\[ \{a^{2^i} : i \geq 0\}. \]

On an input of size \( n \), it made \( O(\log n) \) passes through the input, changing every other \( a \) to a blank.

But each pass visits the same \( O(n) \) tape cells.

So this TM uses \( O(n) \) space on an input of size \( n \).
The main difference is pretty easy to see: space is a resource that can be reused, while time cannot.

For example, consider the satisfiability problem SAT.

Nobody knows how to solve SAT in polynomial time.

But we can easily solve SAT in linear space...
Solving SAT in linear space

Given a logical formula \( \varphi \) with \( t \) variables, all we have to do is loop over all \( 2^t \) possible settings of the variables and evaluate the formula.

If \( \varphi \) takes \( n \) symbols to write down, then \( t \leq n \).

We can use a binary counter on one track of the tape to handle all \( 2^t \) settings of the variables.

We copy the input to another track, substitute the values of the variables, and evaluate the formula.

If it ever evaluates to \( \text{true} \), we halt and accept. Otherwise we halt and reject.
Now we define space complexity classes.

\[
\text{DSPACE}(f) = \{ L : L \text{ is decided by some DTM using } O(f(n)) \text{ space} \},
\]
\[
\text{NSPACE}(f) = \{ L : L \text{ is decided by some NDTM using } O(f(n)) \text{ space} \}.
\]

There are typically infinitely many different strategies to decide membership in a language; some may use lots of space and some very little space.

When we say \( L \) is in \( \text{DSPACE}(f) \), we just mean that there is \textit{at least one} DTM running in this space bound.
There are two relationships between space and time.

One of them is very easy.

**Theorem.** If a 1-tape always halting TM (deterministic or nondeterministic) uses at most $t$ time on an input of length $n$, then it uses at most $t + 1$ space.

**Proof.** In $t$ steps a TM can scan only $t + 1$ different cells.

The other direction is harder...
Theorem. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a 1-tape always-halting deterministic TM. If $M$ uses $s$ space on an input of length $n$, then $M$ halts in at most $(|Q| + 2)s(|\Gamma| + 1)^s$ steps.

Proof. If $M$ scans at most $s$ tape cells on an input of size $n$, since each cell can hold at most $|\Gamma| + 1$ different symbols, there can be at most $(|\Gamma| + 1)^s$ different possible tape contents during a computation.

Counting the possible states, and head positions, this means there are at most

$$(|Q| + 2)s(|\Gamma| + 1)^s$$

different possible configurations of the tape.

If the TM used more steps than this number, then some configuration would be repeated and the TM would be in an infinite loop and hence could not halt.

So $M$ halts in at most $\leq (|Q| + 2)s(|\Gamma| + 1)^s$ steps.
Savitch’s theorem

We can now prove one of the most fundamental results about space complexity: Savitch’s theorem.

Savitch’s theorem says that we can simulate a nondeterministic Turing machine with a deterministic Turing machine, using only a small blowup in space.

**Theorem.** For any function \( f : \mathbb{N} \to \mathbb{N} \) where \( f(n) \geq n \), we have

\[
\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f(n)^2).
\]

What does it mean?

It means: if some NDTM for \( L \) uses \( O(f(n)) \) space to decide \( L \), then we can find a DTM for \( L \) that uses only \( O(f(n)^2) \) space.

This is *quadratic space blow-up.*
We could try to use our simulation of a nondeterministic TM with a deterministic TM. Remember how that worked?

We used a breadth-first search of the computation tree, examining all computational paths.

The breadth-first search used a binary counter to record all the needed nondeterministic choices corresponding to a computational path of length $t$.

From our result above, if an NDTM $M$ uses $O(f(n))$ space then it could use as much as $c^{f(n)}$ time for some constant $c \geq 2$. 
A failed strategy for proving Savitch’s theorem

So the computational paths could be exponentially long, and we would have to record each one to go to the next one.

This would mean our simulation uses exponential space—not good enough to prove the theorem!

Instead we have to use a trick based on directed graph search and recursion.

Let’s solve a more general problem...
Savitch’s theorem

Let $M$ be an NDTM that uses $f(n)$ space.

Given two configurations of $M$’s tape, $c_1$ and $c_2$, let us build a small-space deterministic TM that decides whether it is possible to get from $c_1$ to $c_2$ by $\leq t$ steps.

This is solving the *graph reachability problem* for TM configurations. Solving it computes a boolean function $\text{yield}(c_1, c_2, t)$.

Idea of solution: divide and conquer.
Savitch’s theorem

To determine $\text{yield}(c_1, c_2, t)$, first determine

$$\text{yield}(c_1, c_m, \lfloor t/2 \rfloor)$$

and

$$\text{yield}(c_m, c_2, \lceil t/2 \rceil).$$

If there is such a $c_m$ then $\text{yield}(c_1, c_2, t)$ is true!

Key observation: we can reuse the space from the first computation to do the second!

Base case of the recursion: $t = 0$ or $t = 1$.

If $t = 0$ then this means $c_1 = c_2$, which we can easily check.

If $t = 1$ this means $c_2$ follows from $c_1$ by exactly one step of $M$, which we can also easily check.
yield\((c_1, c_2, t)\):

If \(t = 0\) accept if \(c_1 = c_2\) and reject otherwise.
If \(t = 1\) accept if \(c_1 \vdash c_2\) and reject otherwise.

For each possible configuration \(c_m\) of NDTM \(M\) on input \(w\) using space at most \(f(n)\) do

- Compute \(\text{yield}(c_1, c_m, \lceil t/2 \rceil)\);
- Compute \(\text{yield}(c_m, c_2, \lfloor t/2 \rfloor)\);

If both of the previous accept, then accept and halt.
If we haven’t yet accepted, then reject.
Space analysis for yield

How much space is needed for the execution of yield?

Writing down a configuration uses $O(f(n))$ space.

When yield is called recursively, it stores $c_1, c_2, t$ on a stack so it can recover them later.

The depth of the recursion is at most $\log_2 t$.

But $t \leq (|Q| + 2)f(n)(|\Gamma| + 1)^{f(n)}$. So

$\log_2 t \leq f(n) \log(|\Gamma| + 1) + \log f(n) + \log(|Q| + 2) = O(f(n))$.

So the total space needed is $f(n)$ levels of recursion times $O(f(n))$ for each stack frame (activation record) stored, for a total of $O(f(n)^2)$ space.
Now we can determine whether or not $M$ goes from its initial configuration to an accepting configuration.

To avoid having to deal with many different accepting configurations, we can easily modify $M$ so that if it ever enters the accept state, it erases all the tape cells it has ever visited and moves its head to the starting cell.

In this way there is only one single accepting configuration.

Modifying this might add some constant number of states and increase the space by two cells over the original machine.

It doesn’t change the $O(f(n)^2)$ space bound that we’ll get.
So now what we have to do is, on input \( w \) of length \( n \), compute

\[
yield(c_{\text{start}}, c_{\text{accept}}, t(n))
\]

for \( t(n) := (|Q| + 2)f(n)(|\Gamma| + 1)^{f(n)} \).

One technical problem arises: we need to be able to compute this bound \( t(n) \) in \( O(f(n)) \) space. If \( f \) were very complicated this might not be obvious to do.

We could solve this by also demanding that \( f(n) \) be computable in \( f(n) \) space, which would be true for “most” functions \( f \) that people care about.

But there’s a way around this.
Instead, what we do is call yield with the appropriate $t(n)$ from the formula above, assuming that $f(n) = 1, 2, 3, \ldots$, sequentially.

How do we know when to stop?

When we run yield with a particular value of $f$, say $f(n) = i$, we also use yield to check if $M$ could reach any configuration of length $i$ from the start configuration.

If the accept configuration is reached, our simulation accepts.

If no configuration of length $i$ is reachable from the start, our simulation rejects.

Otherwise we set $i := i + 1$ and continue. Since $M$ was guaranteed to run in $f(n)$ space, at some point we will get an $i$ so that no configuration of length $i$ is reachable, and so the simulation halts.

In this way we do not need to know the value of $f(n)$ at all; it's enough to be guaranteed that $M$ uses no more than $f(n)$ space.
Polynomial space

Now let’s define polynomial space, in analogy with polynomial time:

\[
\text{PSPACE} = \bigcup_{i \geq 1} \text{DSPACE}(n^i).
\]

Intuitively, PSPACE captures the notion of those problems solvable not using “lots” of space.

Of course there is a counterpart, namely nondeterministic polynomial space:

\[
\bigcup_{i \geq 1} \text{NSPACE}(n^i).
\]

But this class has no special name, because it is just PSPACE!

Why? Because from Savitch’s theorem,

\[
\text{NSPACE}(n^i) \subseteq \text{DSPACE}(n^{2i}).
\]
Connecting the hierarchies of space and time

Putting together everything we know:

\[ P \subseteq NP \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXP}. \]

We know that the leftmost class, \( P \), is a strict subset of the rightmost class \( \text{EXP} \).

But which of the \( \subseteq \) are strict, we still don’t know!

At least one must be.
Just like with \( \text{NP} \) and \( \text{NP} \)-completeness, there is a corresponding theory of \( \text{PSPACE} \)-complete problems.

We say a language \( L \) is \( \text{PSPACE} \)-complete if

- \( L \in \text{PSPACE} \); and
- For all languages \( L' \in \text{PSPACE} \), we have that \( L' \leq_p L \).

\( \text{PSPACE} \)-complete problems form an interesting class and seem to be even harder than \( \text{NP} \)-complete problems.
QBF: quantified Boolean formulas.

Here the instance is a Boolean formula $\varphi$ (as in SAT) with quantifiers $\exists$ and $\forall$ corresponding to each variable.

The decision problem is whether $\varphi$ is true or not.

Example instance:

$$\exists x_1 \ \forall x_2 \ \exists x_3 \ (x_1 \lor x_2) \land (x_2 \lor x_3) \land (\overline{x_2} \lor \overline{x_3}).$$

**Theorem.** QBF is PSPACE-complete.
PSPACE-complete games

The structure of a QBF formula is like that of a two-player game.

The $\exists$ quantifiers correspond to moves for player A: is there a winning move?

The $\forall$ quantifiers correspond to moves for player B: here we have to be able to handle every single move by player B.

So a formula that starts with $\exists$ is like asking: is there a winning strategy for player A to the game specified by the rest of the formula?

The formula says something like: “if player A makes this specific move, then no matter what player B responds to player A’s first move, player A wins.”
Because of this relationship between two-player games and quantified Boolean formulas, it should not be a surprise that many two-player games are PSPACE-complete.

Because games with a finite number of possible configurations are always trivially solvable, we need to generalize these games to arbitrarily large instances (e.g., arbitrarily large boards).

The following are examples of games that have been proven to be PSPACE-complete:

- Generalized Go (on $n \times n$ board)
- Generalized Geography
- Generalized Reversi (aka Othello)
- Generalized Super Mario
PSPACE-complete problems

Another PSPACE-complete problem: regular expression universality.

Given a regular expression $r$ over the alphabet $\Sigma$, using the usual operations of union, concatenation, and star, it is PSPACE-complete to determine if $L(r) = \Sigma^*$.

Similarly, given an NFA $M$ over the input alphabet $\Sigma$, it is PSPACE-complete to determine if $L(M) = \Sigma^*$. 