When considering space complexity, the "used space" is meant to represent temporary storage needed by the TM to perform a task.

We want to focus on sub-linear storage.

\[ \Rightarrow \text{we need to differentiate} \]
- memory used for the computation
- memory used for storing input and output

\[ \text{Note: input and output memory should not be abused for temporary storage.} \]

\[ \Rightarrow \]
- input tape is read-only (and need only to end of input)
- output tape is write-only
- working tape is read-write; is the only one that counts

Space complexity \( S(n) \): number of work tape cells scanned on all inputs of length \( n \).

\[ \text{DSPACE}(S(n)) = \{ L \mid L = \mathcal{L}(D) \text{ for some DTM } D \text{ with space complexity } S(n) \} \]

\[ \text{NSPACE}(S(n)) = \{ L \mid L = \mathcal{L}(N) \text{ for some NTM } N \text{ with space complexity } S(n) \} \]

In order to avoid dependence on the tape alphabet, we will consider the binary space complexity: number of bits that can be stored in the tape cells + finite state of the machine

\[ \text{binary space complexity} = \text{number of tape cells} \times \log_2 |T| + \log_2 |Q| \]
Difference is miniscule:

added constant \( \log_2 |a| \) and

constant factor \( \log_2 |n| \)

What is the smallest amount of meaningful space?

- constant space: finite state machines
  - recognize the regular languages

- to compute the input length, we require logarithmic space.

However, meaningful computations that require more than constant space (i.e., beyond regular languages) can be done with less than log space:

For \( L(n) = \log \log n \): \( \text{DSPACE} \left( O(n) \right) \neq \text{DSPACE} \left( O(1) \right) \)

Example: In \( k \in \mathbb{N} \), let \( w_k \) be the concatenation of all \( k \)-bit long strings in lexicographic order separated by \( \# \)’s

\[
 w_k = 0^k \# 0^{k-1} 1 \# 0^{k-2} 10 \# 0^{k-2} 11 \# \ldots \# 1^k
\]

Then \( S = \{ w_k | k \in \mathbb{N} \} \) is not regular, and can be decided by a DTM that uses \( \log \log \) space.

[Proof: [exercise]]

Note that \( |w_k| > 2^k \) and thus \( O(\log k) = O(\log \log |w_k|) \)

- In the other hand, \( \log \log n \) is the smallest amount of space that is more useful than constant space.
Space used in composition of computations:

We consider the composition of two functions over the binary alphabet \( \Sigma = \{0, 1\} \):

\[
\begin{align*}
&f_1 : \Sigma^* \to \Sigma^* \quad \text{computable in space } S_1(n) \text{ by } A_1 \\
&f_2 : \Sigma^* \times \Sigma^* \to \Sigma^* \quad \text{by } S_2(n) \text{ by } A_2
\end{align*}
\]

We want to compute \( f(x) = f_2(f_1(x), f_1(x)) \) by composing the algorithms \( A_1 \) and \( A_2 \).

1) **Trivial composition:** without attempt to economize storage

\[
\begin{array}{c}
\times \\
A_1 \\
\downarrow \\
\rightarrow \leftarrow \rightarrow \\
A_2 \\
\downarrow \\
\rightarrow \rightarrow \\
\rightarrow \\
\rightarrow \\
\rightarrow \rightarrow \rightarrow \rightarrow \\
\rightarrow \\
f(x)
\end{array}
\]

Let \( l_1(m) = \max_{k} |f_1(x)| \) ... maximum output produced by \( A_1 \) on input of length \( m \).

Then \( S_{\text{triv}}(n) = l_1(m) + S_2(m + l_1(m)) + l_1(m) + \cdots \) for storing the result of \( A_1 \) + \( S(m) \)

with \( S(n) = O(\log (l_1(m) + S_2(m + l_1(m)))) \)

Note: \( S(n) \) takes into account that \( A_2 \) refers to two storage devices (one for storing \( f_1(x) \) and one as its work tape), and it needs to maintain its location on both spits.

This can be done by storing the two pointers on the tape (which has overall length \( l_1(m) + S_2(m + l_1(m)) \))

\[ \Rightarrow 2 \text{ times: } \log_2(l_1(m) + S_2(m + l_1(m))) \text{ bits.} \]
2) naive composition:

We economize storage by reusing the worktape of $A_1$ for the computation of $A_2$

We get $\Delta_{\text{naive}}(m) = \max \left( \sigma_1(m) + \sigma_2(m + l_2(m)) \right) + \tau l_2(m) + \ldots$ for storing the result of $A_1$ and $\delta(m)$

Note: both in the naive and in the trivial composition, the most costly storage may be that for the intermediate result.

E.g., when $\sigma_1(m) = \sigma_2(m) = \log m$ but the output of $A_2$ is of length polynomial in $m$, the overall space used is not $\log m$ but a polynomial in $m$.

Can we avoid to explicitly store the intermediate result?

We can, by paying some price in computation time.

Idea: instead of storing the output of $A_1$ on the tape, we recompute it whenever $A_2$ needs to access it.
Theorem: Let $\Sigma, f_0, f_2, f_1, A_1, A_2, l_q$ be as above, i.e.

$\Sigma = \{0, 1\}$

$f_0 : \Sigma^* \rightarrow \Sigma^*$

is computable in space $S_0(m)$

$f_2 : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

is computable in space $S_2(m)$

$f(x) = f_2(\epsilon_1, f_0(x))$

$l_q(m) = \max_{x \in \Sigma^m} |f_0(x)|$

Then $f$ is computable in space $S$ where

$S(m) = S_0(m) + S_2(m + l_q(m)) +$

$+ O(\log (m + l_q(m))) +$

$+ \delta^1(m)$

where $\delta^1(m) = O(\log (S_0(m) + S_2(m + l_q(m))))$

Note: in the overall space used in the cumulative composition, we do not have anymore the term $l_q(m)$ which accounts for storing the intermediate result $f_0(x)$.

Instead, we need to store pointers (i.e., counters) to a virtual storage for $f_0(x)$. 
Proof: Idea: The computation starts directly with $A_2$, computing $f_2(x, y)$, although some of the input bits (those of $f_2(x)$) are not available to $A_2$. Whenever $A_2$ needs such a bit, it re-computes it by calling $A_1$.

More in detail:

Set $A_1$ be an algorithm computing $f_1(x)$ in space $O_1(n)$. Let $f_2(x, y)$ be $O_2(n)$. We assume that $A_1$ never re-writes an output bit.

We start the computation by invoking $A_2$:

Input for $A_2$:
- $x$: first $m$ bits, taken from the input tape.
- $f(x)$: next $l_2(n)$ bits on a virtual input tape.

Whenever $A_2$ needs the $i$-th bit of $f_1(x)$ (i.e., its $(m+i)$th input bit), it gets it by calling $A_1$.

$A_1$ computes this bit, but is provided with a virtual output tape, i.e., $A_1$ computes its bits one by one and discards them, until its $i$-th output bit is computed and passed to $A_2$ (as its $(m+i)$th input bit).

So to do so, we need:
- a counter for the bits produced by $A_1$: $\log_2 (l_2(n))$ bits.
- a counter for the input bit currently read by $A_2$: $\log_2 (m + l_2(n))$ bits.

Note: when invoking $A_1$, we need to suspend the execution of $A_2$ and then resume it when $A_1$ has produced the bit. Hence we need separate storage for the computations of $A_1$ and $A_2$. 
Note 2: $S'(n)$ takes into account that the algorithm refers to two storage devices:

- one for emulating the storage of $A_1$: $S_1(n)$ cells
- $A_2$: $S_2(n + l_1(n))$ cells

Hence, we need two counters on the overall tape

\[ 2 \cdot \log_2(S_1(n) + S_2(n + l_1(n))) \text{ bits} \]

q.e.d.

Observation: to save the intermediate storage space, we pay the price of recomputing $f_2(x)$ again and again (once for each access of $A_2$ to a bit of its second input).

Impact of the previous theorem on reductions:

Consider two languages $L_1$ and $L_2$, and let $R$ be a reduction from $L_1$ to $L_2$ computed in space $S_0(n)$, i.e.,

\[ L_1 \leq_{S_0(n)} L_2 \]

Let $A_2$ be an algorithm for $L_2$ with space complexity $S_2(n)$.

We can apply the theorem by taking $f_2(x, y)$ to be $R(y)$.

We get an algorithm for $A_1$ with space complexity

\[ S(n) = S_0(n) + S_2(l_1(n)) + O(\log l_1(n)) + S'(n) \]

(with $S'(n) = O(\log (S_1(n) + S_2(\log l_1(n))))$)

For example: let $R$ be computable in logarithmic space. Then the output of $R$ is at most polynomial in its input.\[ \text{[proof as exercise]}; \text{ hence } l_1(n) \text{ is a polynomial.} \]

When $A_2$ is a logspace algorithm, (i.e., $S_2(n) = O(\log n)$), then also $S(n)$ is logspace.
**Log-space reductions**

Let $L_1$ and $L_2$ be two languages.

A function $R$ is a log-space reduction of $L_1$ to $L_2$ if
\[ \begin{align*}
   & 1) \text{ } w \in L_1 \iff R(w) \in L_2 \\
   & 2) \text{ } R \text{ is computable in log-space}
\end{align*} \]

We have already shown that if a TM that has space complexity $O(n)$ (for $O(n)$ a polynomial) has running time bounded by $O(s(n))$.

It is easy to see that the same result holds for $s(n)$ being any function that is at least $O(\log_2(n))$. Hence, we get that every log-space reduction is also a poly-time reduction:
\[
   (\text{since } O(\log_2(n)) = O(p(n)) \text{, where } p(n) \text{ is a polynomial})
\]

In fact, all reductions used to show NP-hardness and encountered in practice are log-space reductions.

**The classes L and NL:**

(Note: we use $S$ to denote languages, to avoid confusion with $L$)

\[
L = \{ S \mid S = L(D) \text{ for some DTM } D \text{ with log-space complexity} \}
\]

\[
= \bigcup_{c} \text{DSPACE}(c \cdot \log_2 m)
\]

\[
NL = \{ S \mid S = L(M) \text{ for some NTM } M \text{ with log-space complexity} \}
\]

\[
= \bigcup_{c} \text{NSPACE}(c \cdot \log_2 m)
\]

From the above observation on the relationship between space and time complexity, we get immediately that $L \subseteq P$.

$NL \subseteq NP$
since also for a log-space NTM the number of different configurations is polynomial, the same argument shows also \( \text{NL} \subseteq \text{P} \).

Note: the exact relationship between \( \text{L}, \text{NL}, \text{P}, \text{NP} \) is still an open problem.

Prototypical problems for \( \text{L} \) and \( \text{NL} \) are graph-connectivity problems:

- \( \text{ST-CONN} \) (source-target connectivity in directed graphs):
  - Input: a directed graph \( G = (V,E) \) two vertices \( s, t \) in \( V \)
  - Output: yes iff there is a path in \( G \) from \( s \) to \( t \) (denoted \( s \rightarrow^* t \))

- \( \text{UCONN} \) (connectivity in undirected graphs)
  - Input: an undirected graph \( G = (V,E) \)
  - Output: yes iff \( G \) is connected

In order to define completeness with \( \text{L} \) or \( \text{NL} \), we need to make use of log-space reductions (rather than polytime red.)

**Theorem:** \( \text{ST-CONN} \) is complete for \( \text{NL} \) (under log-space red.)

**Proof:**

1. \( \text{ST-CONN} \in \text{NL} \)

We construct a NTM \( M \) that decides \( \text{ST-CONN} \) using only log-space.

Set \( G = (V,E) \) be the input graph, and \( s, t \) two vertices in \( V \).
N stores on the tape:
- two vertices \( v_1, v_2 \) of \( G \): \( \log_2 |V| \) bits
- a counter \( c \) of the traversed edges: \( \log_2 |E| \) bits
\[ \Rightarrow O(\log_2 |E|) \text{ bits} \]

Algorithm implemented by \( N \):

\[
\begin{align*}
  v_i & \leftarrow s; \quad c \leftarrow 0; \\
  \text{while } v_i \neq t \text{ and } c < |E| & \quad \text{do} \\
  & \quad \text{guess a vertex } v_i; \\
  & \quad \text{if } (v_i, v_i) \notin E \text{ then } v_i \leftarrow v_i; \\
  & \quad \quad c \leftarrow c + 1 \\
  \text{else reject} \\
  \text{if } v_i = t \text{ then accept} \\
  \text{else reject}
\end{align*}
\]

It is easy to see that \( N \) accepts iff there is a path from \( s \) to \( t \) in \( G \).

2) \textit{ST-COV} is NL-hard

Given a log-space NTM \( N \) and an input \( w \), we have to construct a directed graph \( G = (V, E) \) and two vertices \( s, t \) in \( V \) s.t.

\[ w \in \mathcal{L}(N) \quad \text{iff} \quad s \rightarrow G, v \rightarrow t \]

- The set \( V \) of vertices is the set of IDs of \( N \), where each ID consists of:
  - the content of the work tape
  - the TM state
  - the head position on work tape and on input tape
  - the finite control
The directed edges represent single transitions between the configurations.

- an edge \((v_1, v_2)\) depends only on \(N\) and on the single input bit of \(w\) at the position indicated by the input tape head position in \(v_1\).

- The start vertex is the one given by the initial ID of \(N\) on \(w\).

- The end vertex \(t\) is the one corresponding to a canonical accepting ID (we can assume that \(N\), when it accepts, erases the whole tape and moves left).

Then \(\Delta \rightarrow t\) iff \(M\) on \(w\) reaches the accepting ID \(G_{N^x, w, t}\) iff \(w \in L(N)\).

Notice that, for a fixed machine \(N\), \((G_{N^x, w, t}\), \(t\)) can be constructed in log-space in \(|w|\). (by enumerating through pairs of IDs and outputting its edges only the valid ones).

The proof above shows that \(\text{ST-CONN}\) captures the essence of \(\text{NL}\).

What about \(L\) and \(\text{UCONN}\)?

The complexity of \(\text{UCONN}\) was open for more than 20 years.

**Theorem [Reingold, 2004]**

\[\text{UCONN} \in L\]

**Proof**: Very complicated

Note that traditional breadth-first and depth-first graph search algorithms use linear space.
$\text{coNL} = \{ S \mid \bar{S} = \Sigma^* \setminus S \in \text{NL} \}$ ... complements of problems in $\text{NL}$

Recall that the acceptance condition of non-deterministic computations is asymmetric in nature:

For $S \in \text{NL}$ and $N$ a log-space NTM accepting $S$:
- if $w \in S$ then there exists an accepting computation of $N$ on $w$
- if $w \notin S$ then all computations of $N$ on $w$ are not accepting

Note that:
- non-det-time complexity classes are not known to be closed under complementation (e.g. $\text{NP} \neq \text{coNP}$)
- non-det-space complexity classes are closed under complementation
  - for classes above $\text{PSPACE}$, this follows immediately from the quadratic simulation of non-deterministic computations via deterministic ones (e.g. $\text{PSPACE} = \text{NPSPACE}$)
  - for $L$ vs. $\text{NL}$, it is open whether $L \neq \text{NL}$
  
However, we have the following result

**Theorem [Immerman / Szelepcsenyi 1988]:** $\text{NL} = \text{coNL}$

Proof: complicated