Logarithmic Space

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.

$\Rightarrow$ we need to differentiate

- memory used for the computation
- memory used for storing input and output

Note: input and output memory should not be abused for temporary storage.

$\Rightarrow$
- input tape is read-only (and read 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.

$$DSPACE(s(n)) = \{ L \mid L = \mathcal{L}(D) \text{ for some DTM } D \text{ with space complexity } s(n) \}$$

$$NSPACE(s(n)) = \{ L \mid L = \mathcal{L}(N) \text{ for some NTM } N \text{ with space complexity } s(n) \}$$

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

binary space complexity = number of tape cells $\cdot \log_2 |\Gamma|$
$+ \log_2 |Q|$

Difference is immaterial:

added constant $\log_2 |Q|$ and

constant factor $\log_2 |\Gamma|$

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 $\ell(n) = \log\log n$: $\mathrm{DSPACE}(O(\ell)) \supsetneq \mathrm{DSPACE}(O(1))$

Example: For $k \in \mathbb{N}$, let w_k be the concatenation of all k-bit long strings in lexicographic order separated by #'s

$$w_k = 0^k \# 0^{k-1}1 \# 0^{k-2}10 \# 0^{k-2}11 \# \cdots \# 1^k$$

Then $S = \{w_k \mid k \in \mathbb{N}\}$ is not regular, and can be decided by a DTM that uses loglog space.

Proof: [exercise]

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

- On 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:

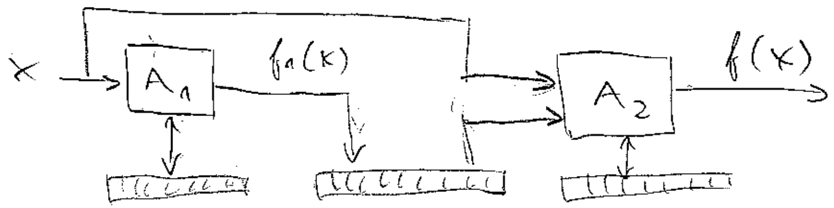
We consider the composition of two functions over the binary alphabet $\Sigma = \{0,1\}$:

$f_1 : \Sigma^* \to \Sigma^*$ computable in space $s_1(n)$ by A_1

$f_2 : \Sigma^* \times \Sigma^* \to \Sigma^*$ -"- $s_2(n)$ by A_2

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

1) Trivial composition: without attempt to economize storage



Let $l_1(n) = \max_{x \in \Sigma^n} |f_1(x)|$ ... maximum output produced by A_1 on inputs of length n

Then $s_{trivial}(n) = s_1(n) + s_2(n + l_1(n)) +$

$+ l_1(n) +$ ... for storing the result of A_1

$+ \delta(n)$

with $\delta(n) = O(\log(l_1(n) + s_2(n + l_1(n))))$

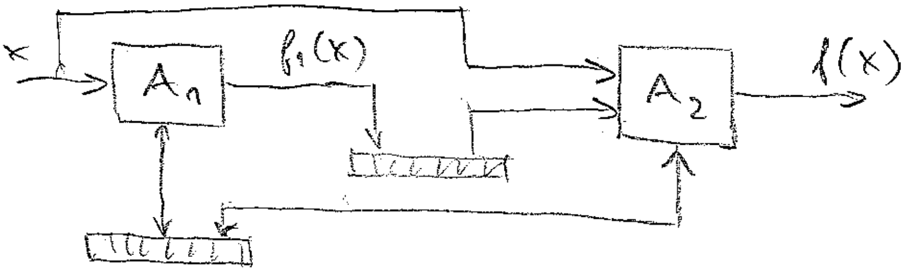
Note: $\delta(n)$ takes into account that A_2 refers to two storage devices (one for storing $f_1(x)$ and one as its work tape), and it needs to maintain its location on both parts.

This can be done by storing the two pointers on the tape (which has overall length $l_1(n) + s_2(n + l_1(n))$)

$\Rightarrow$ 2 times $\log_2(l_1(n) + s_2(n + l_1(n)))$ bits.

2) Naive composition:

We economize storage by reusing the work-tape of A_1 for the computation of A_2



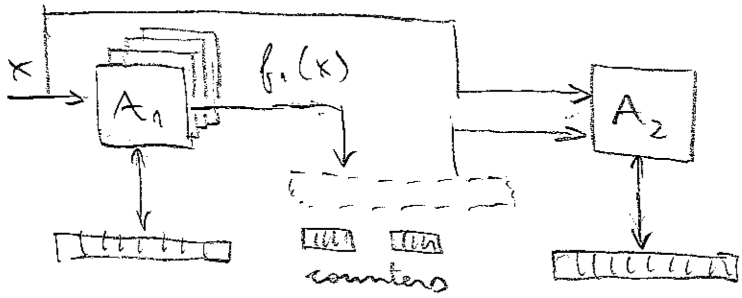
We get $s_{naive}(n) = \max\left(s_1(n) + s_2(n + l_1(n))\right) +$
$+ \, l_1(n) + \ldots$ for storing the result of A_1
$+ \, \delta(n)$

Note: both in the naive and in the trivial composition, the most costly storage may be that for the intermediate result.
E.g., when: $s_1(n) = s_2(n) = \log n$, but the output of A_1 is of length polynomial in n, the overall space used is not $\log n$, but a polynomial in n.

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

3) Emulative composition:



The intermediate result is not stored explicitly (we view A_2 as if it was accessing a virtual storage device)

Theorem: Let Σ, f_1, f_2, f, s_1, s_2, l_1 be as above, i.e.

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

$f_1 : \Sigma^* \to \Sigma^*$ computable in space $s_1(n)$

$f_2 : \Sigma^* \times \Sigma^* \to \Sigma^*$ — " — $s_2(n)$

$f(x) = f_2(x, f_1(x))$

$l_1(n) = \max\limits_{x \in \Sigma^n} |f_1(x)|$

Then f is computable in space s where

$$s(n) = s_1(n) + s_2(n + l_1(n)) + O(\log(n + l_1(n))) + \delta'(n)$$

where $\delta'(n) = O(\log(s_1(n) + s_2(n + l_1(n))))$

Note: in the overall space used in the emulative composition, we do not have anymore the term $l_1(n)$ which accounts for storing the intermediate result $f_1(x)$.

Instead, we need to store pointers (i.e., counters) to a virtual storage for $f_1(x)$.

Proof: Idea: the computation starts directly with A_2 computing $f_2(x, f_1(x))$ although some of the input bits (those of $f_1(x)$) are not available to A_2. Whenever A_2 needs such a bit, it recomputes it by calling A_1.

More in detail:

Let A_1 be an algorithm computing $f_1(x)$ in space $s_1(n)$.
A_2 — " — $f_2(x, y)$ — " — $s_2(n)$.

We assume that A_1 never rewrites an output bit.

We start the computation by invoking A_2.

Input for A_2:

- x: first n bits, taken from the input tape.
- $f_1(x)$: next $l_1(n)$ bits on a virtual input tape

Whenever A_2 needs the i-th bit of $f_1(x)$ (i.e., its $(n+i)$-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 $(n+i)$th input bit).

To do so, we need:

- a counter for the bits produced by A_1: $\log_2(l_1(n))$ bits
- a counter for the input bit currently read by A_2:
  $\log_2(n + l_1(n))$ 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: $\delta'(n)$ takes again 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

$\Rightarrow 2 \cdot \log_2(s_1(n) + s_2(n + l_1(n)))$ bits

q.e.d.

Observation: to save the intermediate storage space, we pay the price of recomputing $f_1(x)$ 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 $s_1(n)$, i.e.

$$L_1 <_{s_1(n)} L_2$$

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

$$s(n) = s_1(n) + s_2(l_1(n)) + O(\log l_1(n)) + \delta'(n)$$

with $\delta'(n) = O(\log(s_1(n) + s_2(\log l_1(n))))$

For example: let R be computable in logarithmic space.

Then the output of R is at most polynomial in its input [Proof as exercise], hence $l_1(n)$ is a polynomial.

When A_2 is a logspace algorithm, (i.e. $s_1(n) = O(\log n)$), 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

1) $w \in L_1 \iff R(w) \in L_2$
2) R is computable in log-space

We have already shown that a TM that has space complexity $s(n)$ (for $s(n)$ a polynomial) has running time bounded by $2^{O(s(n))}$

It is easy to see that the same result holds for $s(n)$ being any function that is at least $O(\log(n))$. Hence, we get that every logspace reduction is also a poly-time reduction:

(since $2^{O(\log(n))} = O(p(n))$, where $p(n)$ 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)

$L = \{S \mid S = \mathcal{L}(D)$ for some DTM D with log-space complexity$\} =$

$= \bigcup_c \mathrm{DSPACE}(c \cdot \log_2 n)$

$NL = \{S \mid S = \mathcal{L}(N)$ for some NTM M with log-space complexity$\}$

$= \bigcup_c \mathrm{NSPACE}(c \cdot \log_2 n)$

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 $\underline{N}TM$ the number of different configurations is polynomial, the same argument shows also $NL \subseteq P$.

Note: the exact relationship between L, NL, P, NP is still an open problem

Prototypical problems for L and NL are <u>graph-connectivity problems</u>:

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 $s \to_G t$)

UCONN (connectivity in undirected graphs)
- input: an undirected graph $G=(V,E)$
- output: yes iff G is connected

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

<u>Theorem</u>: st-CONN is complete for NL (under log-space red.)

Proof:

1) st-CONN $\in NL$

We construct a NTM N that decides st-CONN using only log-space

Let $G=(V,E)$ be the input graph, and s,t two vertices in V.

M stores on the tape:

- two vertices v_1, v_2 of G : $2 \cdot \log_2 |V|$ bits
- a counter c of the traversed edges: $\log_2 |E|$ bits

$\Rightarrow O(\log_2 |G|)$ bits

Algorithm implemented by M:

$v_1 \leftarrow s$; $c \leftarrow 0$;
while $v_1 \neq t$ and $c < |E|$
  do guess a vertex v_2;
    if $(v_1, v_2) \in E$ then $v_1 \leftarrow v_2$
                $c \leftarrow c+1$
            else reject
if $v_1 = t$ then accept
      else reject

It is easy to see that M accepts iff there is a path from s to t in G.

2) st-CONN is NL-hard

Given a log-space NTM M and an input w,
we have to construct a directed graph $G_{M,w} = (V, E)$ and two vertices s, t in V s.t.

$$w \in \mathcal{L}(M) \quad \text{iff} \quad s \rightarrow_{G_{M,w}} t$$

- The set V of vertices is the set of IDs of M, 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 M and on the single input bit of w at the position indicated by the input-tape head position in v_1.
- The start vertex s is the one given by the initial ID of M on w.
- The end vertex t is the one corresponding to a canonical accepting ID (we can assume that M, when it accepts, erases the whole tape and moves left)

Then $s \xrightarrow[G_{M,w}]{} t$ iff M on w reaches the accepting ID
iff $w \in \mathcal{L}(M)$

Notice that, for a fixed machine M, $(G_{M,w}, s, t)$ can be constructed in log-space in $|w|$. (by enumerating through pairs of IDs and outputting as edges only the valid ones). □

The proof above shows that st-CONN captures the essence of NL.

What about L and UCONN?
The complexity of UCONN was open for more than 20 years.

<u>Theorem</u> [Reingold, 2004]

UCONN $\in$ L

Proof: very complicated

note that traditional breadth-first and depth-first graph search algorithms use linear space.

NL vs. coNL:

$\text{coNL} = \{ S \mid \overline{S} = \Sigma^* \setminus S \in \text{NL} \}$ ... complements of problems in NL

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

For $S \in \text{NL}$ and M a log-space NTM accepting S:

if $w \in S$ then there exists an accepting computation of M on w

if $w \notin S$ then all computations of M on w are not accepting

Note that:

- non-det-time complexity classes are not known to be closed under complementation (e.g. NP vs. coNP)
- non-det. space complexity classes are closed under complementation
  - for classes above PSPACE, this follows immediately from the quadratic simulation of non-deterministic computations via deterministic ones (e.g. PSPACE = NPSPACE)
  - for L vs. NL, it is open whether $L \subsetneq NL$

    However, we have the following result

<u>Theorem</u> [Immerman, Szelepcsenyi 1988]: NL = coNL

Proof complicated