Computational Complexity Theory

Lecture 10: Savitch's theorem; PSPACE-completeness; Log-space reductions

Department of Computer Science,
Indian Institute of Science

Recap: Time versus space

- **Obs.** $\mathrm{DTIME}(S(n)) \subsetneq \mathrm{DSPACE}(S(n)) \subseteq \mathrm{NSPACE}(S(n))$.

- **Theorem.** $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))})$, if S is space constructible.

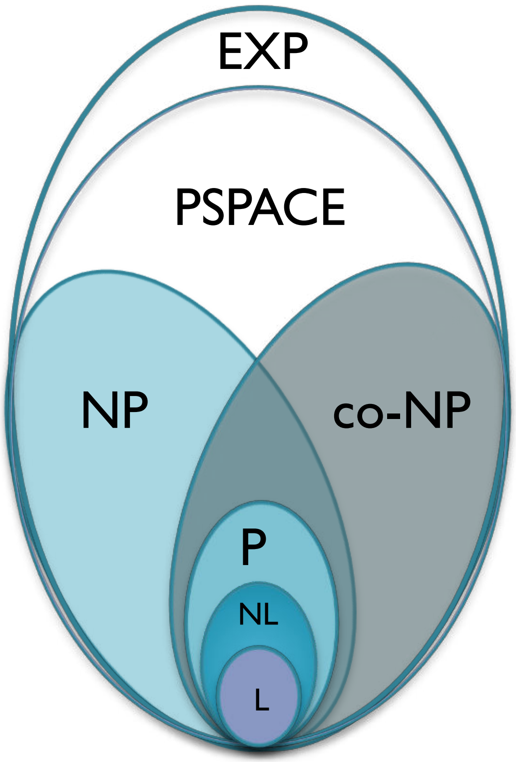
- **Definition.**

$$\begin{aligned} \mathrm{L} &= \mathrm{DSPACE}(\log n) \\ \mathrm{NL} &= \mathrm{NSPACE}(\log n) \\ \mathrm{PSPACE} &= \bigcup_{c>0} \mathrm{DSPACE}(n^c) \end{aligned}$$

Recap: Time versus space

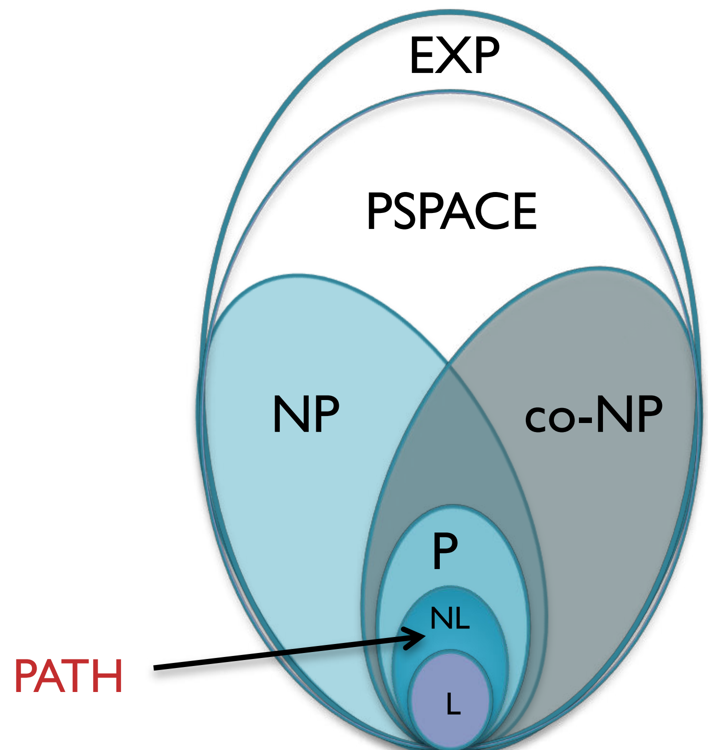
- Obs. $\mathrm{DTIME}(S(n)) \subsetneq \mathrm{DSPACE}(S(n)) \subseteq \mathrm{NSPACE}(S(n))$.

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DTIME}(2^{O(S(n))})$, if S is space constructible.



Recap: PATH is in NL

- PATH = {(G,s,t) : G is a directed graph having a path from s to t}.
- Obs. PATH is in NL.



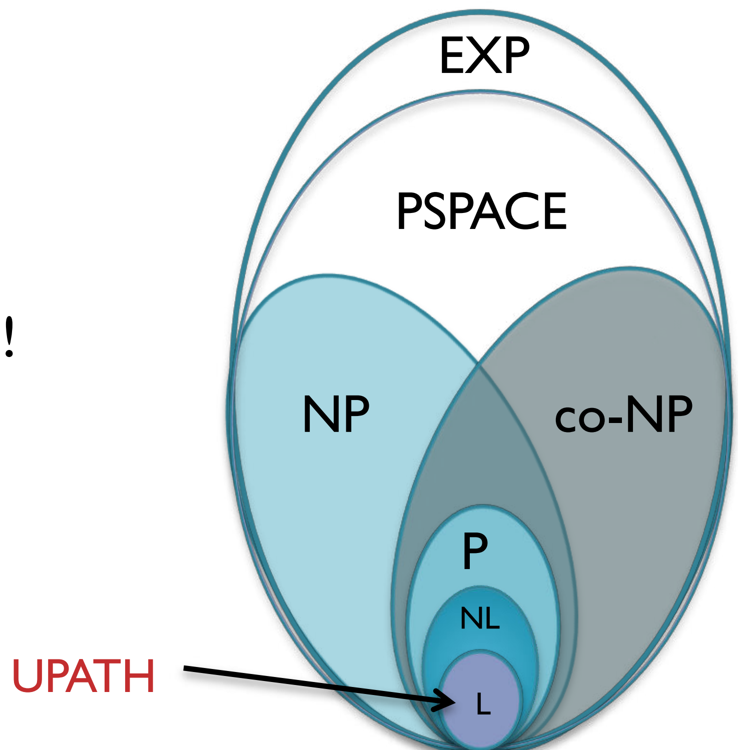
Recap: UPATH is in L

- UPATH = {(G,s,t) : G is an undirected graph having a path from s to t}.
- Theorem *(Reingold 2005)*. UPATH is in L.

Is PATH in L ?

If yes, then L = NL !

(will prove later)



Recap: Space Hierarchy Theorem

- Theorem. *(Stearns, Hartmanis & Lewis 1965)* If f and g are space-constructible functions and $f(n) = o(g(n))$, then $\mathrm{SPACE}(f(n)) \subsetneq \mathrm{SPACE}(g(n))$.

- Proof. Homework.

- Theorem. $\mathrm{L} \subsetneq \mathrm{PSPACE}$.

PSPACE = NPSPACE

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. Let L $\in$ NSPACE(S(n)), and M be an NTM requiring O(S(n)) space to decide L. We'll show that there's a TM N requiring O(S(n)^2) space to decide L.

Savitch's theorem

- Theorem. NSPACE(S(n)) ⊆ DSPACE($S(n)^2$), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. Let L ∈ NSPACE(S(n)), and M be an NTM requiring O(S(n)) space to decide L. We'll show that there's a TM N requiring O($S(n)^2$) space to decide L.

- On input x, N checks if there's a path from C_{start} to C_{accept} in $G_{M,x}$ as follows: Let |x| = n.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. (contd.) N computes $m = O(S(n))$, the no. of bits required to represent a configuration of M. It also finds out C_{start} and C_{accept}. Then N checks if there's a path from C_{start} to C_{accept} of length at most 2^m in $G_{M,x}$ recursively using the following procedure.

- REACH(C_1, C_2, i) : returns 1 if there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$; 0 otherwise.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof. (contd.) $\underbrace{\text{N computes m = O(S(n))}}_{\text{Space constructibility of S(n) used here}}$, the no. of bits required to represent a configuration of M. It also finds out C_{start} and C_{accept}. Then N checks if there's a path from C_{start} to C_{accept} of length at most 2^m in $G_{M,x}$ recursively using the following procedure.

- REACH(C_1, C_2, i) : returns 1 if there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$; 0 otherwise.

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- REACH(C_1, C_2, i) {

  If $i = 0$ check if C_1 and C_2 are adjacent.

  Else, <u>for every</u> configurations C,

  a_1 = REACH(C_1, C, i-1)

  a_2 = REACH(C, C_2, i-1)

  if a_1=1 & a_2=1, return 1. Else return 0.

}

Savitch's theorem

- Theorem. $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- $\mathrm{REACH}(C_1, C_2, i)$ {

  If $i = 0$ check if C_1 and C_2 are adjacent.

  Else, <u>for every</u> configurations C,

  $a_1 = \mathrm{REACH}(C_1, C, i\text{-}1)$

  $a_2 = \mathrm{REACH}(C, C_2, i\text{-}1)$

  if $a_1=1$ & $a_2=1$, return 1. Else return 0.

}

Require $O(S(n))$ space

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.
- REACH(C_1, C_2, i) {

  If i = 0 check if C_1 and C_2 are adjacent.

  Else, <u>for every</u> configurations C,

  $$\left.\begin{array}{l} a_1 = \text{REACH}(C_1, C, i-1) \\ a_2 = \text{REACH}(C, C_2, i-1) \end{array}\right\} \text{Reuse space}$$

  if a_1=1 & a_2=1, return 1. Else return 0.

}

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$\text{Space}(i) = \text{Space}(i\text{-}1) + O(S(n))$$

- Space complexity: O(S(n)^2)

Savitch's theorem

- Theorem. NSPACE(S(n)) $\subseteq$ DSPACE(S(n)^2), where S(n) is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$\text{Space}(i) = \text{Space}(i\text{-}1) + O(S(n))$$

- Space complexity: $O(S(n)^2)$

$$\text{Time}(i) = 2^m.2.\text{Time}(i\text{-}1) + O(S(n))$$

- Time complexity: $2^{O(S(n)^2)}$

Savitch's theorem

- Theorem. $NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$, where $S(n)$ is space constructible. (So, PSPACE = NPSPACE)

- Proof.

$$Space(i) = Space(i-1) + O(S(n))$$

- Space complexity: $O(S(n)^2)$

$$Time(i) = 2^m.2.Time(i-1) + O(S(n))$$

- Time complexity: $2^{O(S(n)^2)}$

Recall, $NSPACE(S(n)) \subseteq DTIME(2^{O(S(n))})$. There's an algorithm with time complexity $2^{O(S(n))}$, but higher space requirement.

PSPACE-completeness

PSPACE-completeness

- Recall, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is P = PSPACE ?

PSPACE-completeness

- Recall, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is P = PSPACE ? …use poly-time Karp reduction!

- Definition. A language L' is *PSPACE-hard* if for every L in PSPACE, L $\leq_p$ L'. Further, if L' is in PSPACE then L' is *PSPACE-complete*.

A PSPACE-complete problem

- Recall, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is P = PSPACE ? …use poly-time Karp reduction!

- Example. $L' = \{(M, w, 1^m) : M$ accepts w using m space$\}$

Natural PSPACE-complete problem

- Definition. A *quantified Boolean formula (QBF)* is a formula of the form

$$\underbrace{Q_1 x_1\ Q_2 x_2 \ldots Q_n x_n}_{\text{Quantifiers } \exists \text{ or } \forall}\ \underbrace{\phi(x_1, x_2, \ldots, x_n)}_{\text{Just a formula on Boolean variables}}$$

- A QBF is either *true* or *false* as all variables are quantified. This is unlike a formula we've seen before where variables were unquantified/free.

Natural PSPACE-complete problem

- Example. $\exists x_1 \exists x_2 \ldots \exists x_n \ \phi(x_1, x_2, \ldots, x_n)$

- The above QBF is true iff ϕ is satisfiable.

- We could have defined SAT as

  $$\text{SAT} = \{\exists \mathbf{x}\, \phi(\mathbf{x}) : \phi \text{ is a CNF and } \exists \mathbf{x}\, \phi(\mathbf{x}) \text{ is true}\}$$

  instead of

  $$\text{SAT} = \{\phi(\mathbf{x}) : \phi \text{ is a CNF and } \phi \text{ is satisfiable}\}$$

Natural PSPACE-complete problem

- Definition. A *quantified Boolean formula (QBF)* is a formula of the form

$$Q_1 x_1\ Q_2 x_2\ \dots\ Q_n x_n\ \ \underbrace{\phi(x_1, x_2, \dots, x_n)}_{\text{Just a formula on Boolean variables}}$$

Quantifiers $\exists$ or $\forall$

- Homework: By using auxiliary variables (as in the proof of Cook-Levin) and introducing some more $\exists$ quantifiers at the end, we can assume w.l.o.g. that ϕ is a 3CNF.

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: Easy to see that TQBF is in PSPACE – just think of a suitable **recursive procedure**. We'll now show that every L ∈ PSPACE reduces to TQBF via poly-time Karp reduction…

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Let M be a TM deciding L using S(n) = poly(n) space. We intend to come up with a poly-time reduction f s.t.

$$x \in L \overset{f}{\Longleftrightarrow} \psi_x \text{ is a true QBF}$$

Size of ψ_x must be bounded by poly(n), where $|x| = n$

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Let M be a TM deciding L using S(n) = poly(n) space. We intend to come up with a poly-time reduction f s.t.

$$x \in L \overset{f}{\longleftrightarrow} \psi_x \text{ is a true QBF}$$

Idea: Form ψ_x in such a way that ψ_x is true iff there's a path from C_{start} to C_{accept} in $G_{M,x}$.

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) f computes $S(n)$ from n (recall, any poly function $S(n)$ is time constructible). It also computes $m = O(S(n))$, the no. of bits required to represent a configuration in $G_{M,x}$.

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) f computes $S(n)$ from n (recall, any poly function $S(n)$ is time constructible). It also computes $m = O(S(n))$, the no. of bits required to represent a configuration in $G_{M,x}$. Then, it forms a *semi-QBF* $\Delta_i(C_1,C_2)$, such that $\Delta_i(C_1,C_2)$ is true iff there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$.

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) f computes $S(n)$ from n (recall, any poly function $S(n)$ is time constructible). It also computes $m = O(S(n))$, the no. of bits required to represent a configuration in $G_{M,x}$. Then, it forms a *semi-QBF* $\Delta_i(C_1,C_2)$, such that $\Delta_i(C_1,C_2)$ is true iff there's a path from C_1 to C_2 of length at most 2^i in $G_{M,x}$.

The variables corresponding to the bits of C_1 and C_2 are unquantified/free variables of Δ_i

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) QBF $\Delta_i(C_1,C_2)$ is formed, recursively, as follows:

(first attempt)

$$\Delta_i(C_1,C_2) = \exists C \ \big(\Delta_{i-1}(C_1,C) \wedge \Delta_{i-1}(C,C_2)\big)$$

Issue: Size of Δ_i is **twice** the size of Δ_{i-1} !!

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) QBF $\Delta_i(C_1,C_2)$ is formed, recursively, as follows:

(careful attempt)

$$\Delta_i(C_1,C_2) = \exists C\ \forall D_1 \forall D_2$$

$$\Big(\ \big((D_1 = C_1 \wedge D_2 = C) \vee (D_1 = C \wedge D_2 = C_2)\big) \Rightarrow \Delta_{i-1}(D_1,D_2)\ \Big)$$

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) QBF $\Delta_i(C_1,C_2)$ is formed, recursively, as follows:

(careful attempt)

$\Delta_i(C_1,C_2) = \exists C\ \forall D_1 \forall D_2$

$$\Big(\neg\big((D_1 = C_1 \wedge D_2 = C) \vee (D_1 = C \wedge D_2 = C_2)\big) \quad \vee \quad \Delta_{i-1}(D_1,D_2)\ \Big)$$

Note: Size of Δ_i = $O(S(n))$ + Size of Δ_{i-1}

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Finally,

$$\psi_x = \Delta_m(C_{start}, C_{accept})$$

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Finally,

$$\psi_x = \Delta_m(C_{start}, C_{accept})$$

- But, we need to specify how to form $\Delta_0(C_1, C_2)$.
- Size of $\psi_x = O(S(n)^2)$ + Size of Δ_0

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Finally,

$$\psi_x = \Delta_m(C_{start}, C_{accept})$$

- But, we need to specify how to form $\Delta_0(C_1, C_2)$.
- Size of $\psi_x = O(S(n)^2)$ + Size of Δ_0

Remark: We can easily bring all the quantifiers at the beginning in ψ_x (as in a *prenex normal form*).

Natural PSPACE-complete problem

- Definition. TQBF is the set of ***true*** quantified Boolean formulas.

- Theorem. TQBF is PSPACE-complete.
- Proof: (contd.) Finally,

$$\psi_x = \Delta_m(C_{start}, C_{accept})$$

- But, we need to specify how to form $\Delta_0(C_1,C_2)$.
- Size of $\psi_x = O(S(n)^2)$ + Size of Δ_0 $\longrightarrow$??

Adjacent configurations

- Claim. There's an $O(S(n)^2)$-size circuit $\phi_{M,x}$ on $O(S(n))$ inputs such that for every inputs C_1 and C_2, $\phi_{M,x}(C_1, C_2) = 1$ iff C_1 and C_2 encode two neighboring configurations in $G_{M,x}$.

- Proof. Think of a linear time algorithm that has the knowledge of M and x, and on input C_1 and C_2 it checks if C_2 is a neighbor of C_1 in $G_{M,x}$.

Adjacent configurations

- Claim. There's an $O(S(n)^2)$-size circuit $\phi_{M,x}$ on $O(S(n))$ inputs such that for every inputs C_1 and C_2, $\phi_{M,x}(C_1, C_2) = 1$ iff C_1 and C_2 encode two neighboring configurations in $G_{M,x}$.

- Proof. Think of a linear time algorithm that has the knowledge of M and x, and on input C_1 and C_2 it checks if C_2 is a neighbor of C_1 in $G_{M,x}$. Applying ideas from the proof of Cook-Levin theorem, we get our desired $\phi_{M,x}$ of size $O(S(n)^2)$.

Size of Δ_0

- Obs. We can convert the circuit $\phi_{M,x}(C_1, C_2)$ to a quantified CNF $\Delta_0(C_1,C_2)$ by introducing auxiliary variables (as in the proof of Cook-Levin theorem).

- Hence, size of $\Delta_0(C_1,C_2)$ is $O(S(n)^2)$.
- Therefore, size of $\psi_x = O(S(n)^2)$.

Other PSPACE complete problems

- Checking if a player has a winning strategy in certain two-player games, like (generalized) Hex, Reversi, Geography etc.
- Integer circuit evaluation (*Yang 2000*).
- Implicit graph reachability.
- Check the wiki page: *https://en.wikipedia.org/wiki/List_of_PSPACE-complete_problems*

NL-completeness

NL-completeness

- Recall again, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is L = NL ?

NL-completeness

- Recall again, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is L = NL ? …poly-time (Karp) reductions are much too powerful for L.
- We need to define a suitable *'log-space'* reduction.

Log-space reductions

$$x \xrightarrow{\text{Log-space TM}} f(x)$$

- Issue: A log-space TM may not have enough space to write down the whole output f(x) in one shot.

…unless we restrict $|f(x)| = O(\log |x|)$, in which case we're severely restricting the power of the reduction.

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output f(x) in one shot.
- Solution: Have the log-space TM output a bit of f(x).

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Definition: A function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is *implicitly log-space computable* if
  1. $|f(x)| \leq |x|^c$ for some constant c,
  2. The following two languages are in L :

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

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Definition: A language L_1 is *log-space reducible* to a language L_2, denoted $L_1 \leq_l L_2$, if there's an implicitly log-space computable function f such that

$$x \in L_1 \iff f(x) \in L_2$$

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- **Issue:** A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- **Solution:** Have the log-space TM output a bit of $f(x)$.

- **Claim:** If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- **Proof:** Let f be the reduction from L_1 to L_2, and g the reduction from L_2 to L_3. We'll show that the function $h(x) = g(f(x))$ is implicitly log-space computable which will suffice as,

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3$$

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Claim: If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- Proof: …Think of the following log-space TM that computes $h(x)_i$ from (x, i). Let
  - M_f be the log-space TM that computes $f(x)_j$ from (x, j),
  - M_g be the log-space TM that computes $g(y)_i$ from (y, i).

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Claim: If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- Proof: …On input x, simulate M_g on $(f(x), i)$ <u>pretending that</u> $f(x)$ is there in some fictitious tape. During the simulation whenever M_g tries to read a j-th bit of $f(x)$, <u>postpone</u> M_g's computation and start simulating M_f on input (x, j).

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- **Issue:** A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- **Solution:** Have the log-space TM output a bit of $f(x)$.

stores M_g's current configuration

- **Claim:** If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- **Proof:** …On input x, simulate M_g on $(f(x), i)$ <u>pretending that</u> $f(x)$ is there in some fictitious tape. During the simulation whenever M_g tries to read a j-th bit of $f(x)$, <u>postpone</u> M_g's computation and start simulating M_f on input (x, j). Space usage $= O(\log |f(x)|) + O(\log |x|)$.

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Claim: If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- Proof: …On input x, simulate M_g on $(f(x), i)$ <u>pretending that</u> $f(x)$ is there in some fictitious tape. During the simulation whenever M_g tries to read a j-th bit of $f(x)$, <u>postpone</u> M_g's computation and start simulating M_f on input (x, j). Space usage $= O(\log |x|)$.

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Claim: If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- Proof: …On input x, simulate M_g on $(f(x), i)$ pretending that $f(x)$ is there in some fictitious tape. During the simulation whenever M_g tries to read a j-th bit of $f(x)$, <u>postpone</u> M_g's computation and start simulating M_f on input (x, j). This shows L_h is in L.

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output f(x) in one shot.
- Solution: Have the log-space TM output a bit of f(x).

- Claim: If $L_1 \leq_l L_2$ and $L_2 \leq_l L_3$ then $L_1 \leq_l L_3$.
- Proof: …Similarly, L'_h is in L and so h is implicitly log-space computable.

Log-space reductions

$$(x, i) \xrightarrow{\text{Log-space TM}} f(x)_i$$

- Issue: A log-space TM may not have enough space to write down the whole output $f(x)$ in one shot.
- Solution: Have the log-space TM output a bit of $f(x)$.

- Claim: If $L_1 \leq_l L_2$ and $L_2 \in \mathbf{L}$ then $L_1 \in \mathbf{L}$.
- Proof: Same ideas. (*Homework*)