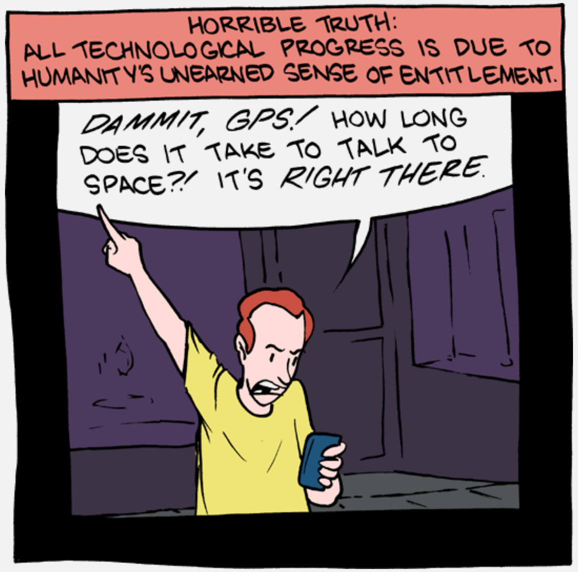
UMB CS622

PSPACE Completeness

Monday, November 29, 2021



Announcements

- HW 9 extended
  - Due Tues 11/30 11:59pm EST

- HW 10 released
  - Due Tues 12/7 11:59pm EST

- HW 11 will be last assignment
  - Due Tues 12/14 11:59pm EST

Flashback: Dynamic Programming Example

- Chomsky Grammar *G*:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

We are gaining time …

… by spending more space!

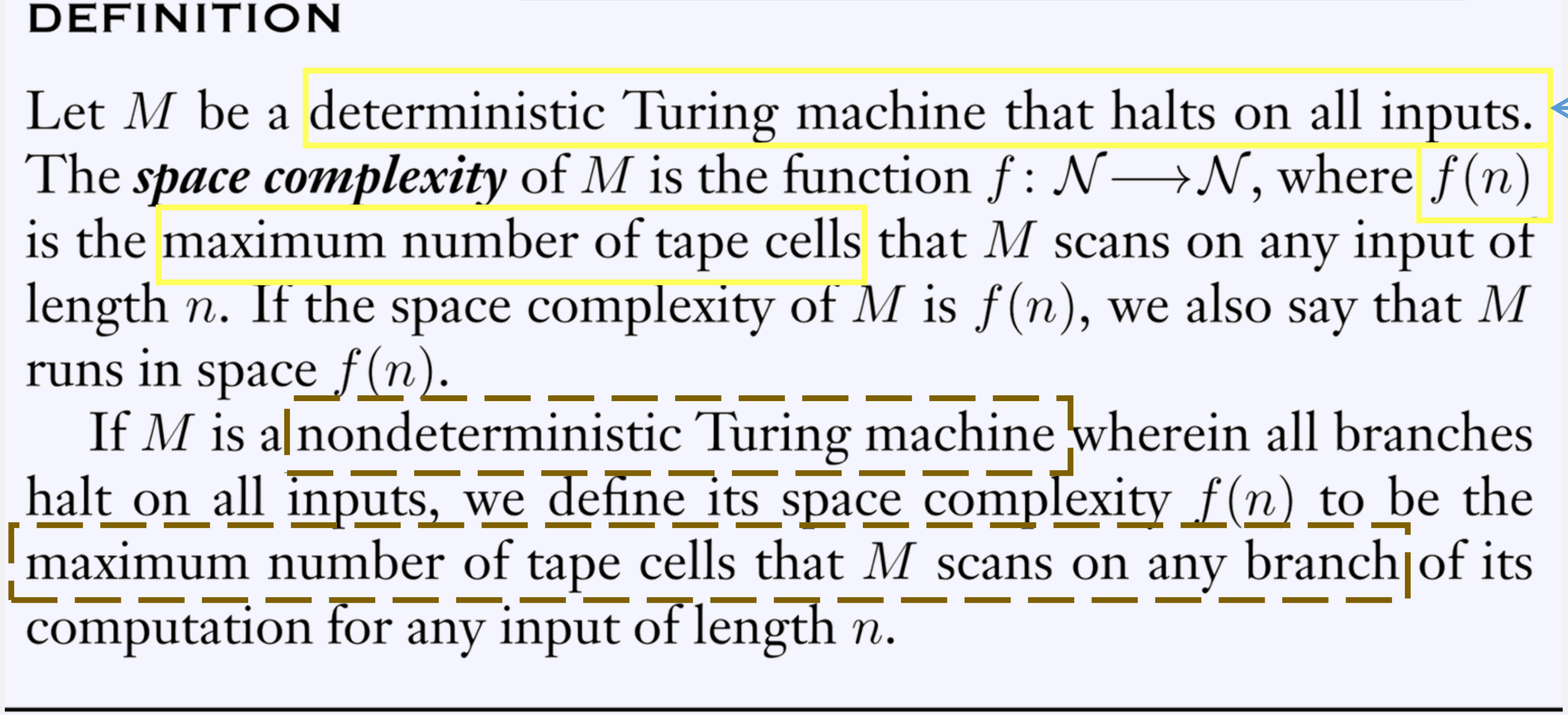
Substring end char

Substring start char

	b	**a**	**a**	**b**	**a**
b	vars for “b”	vars for “ba”	vars for “baa”	…	
a		vars for “a”	vars for “aa”	vars for “aab”	
a			…		
b					
a					

Space Complexity, Formally

TMs have a **space complexity**



DEFINITION

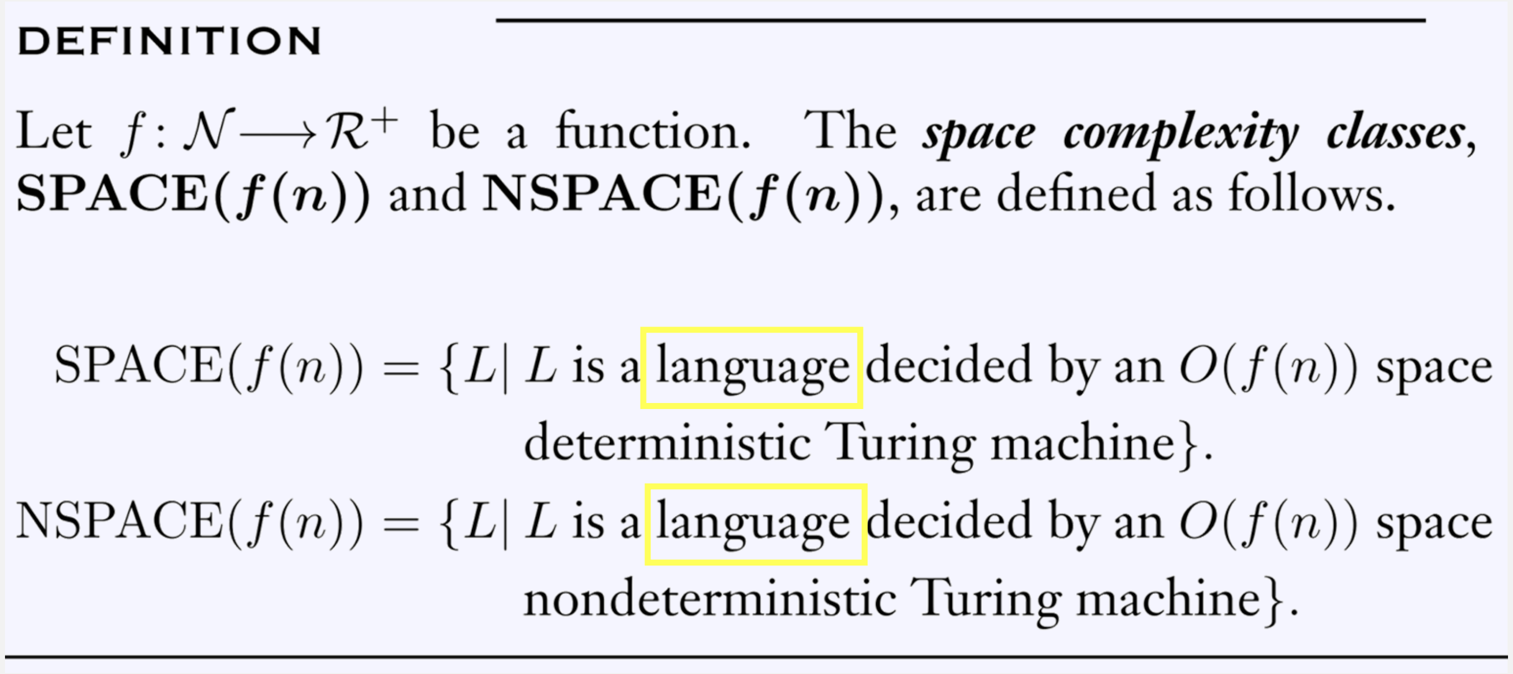
Let M be a deterministic Turing machine that halts on all inputs. The ***space complexity*** of M is the function $f\colon \mathcal{N}\longrightarrow\mathcal{N}$, where $f(n)$ is the maximum number of tape cells that M scans on any input of length n. If the space complexity of M is $f(n)$, we also say that M runs in space $f(n)$.

If M is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

decider

Space Complexity Classes

Languages are in a **space complexity <u>class</u>**



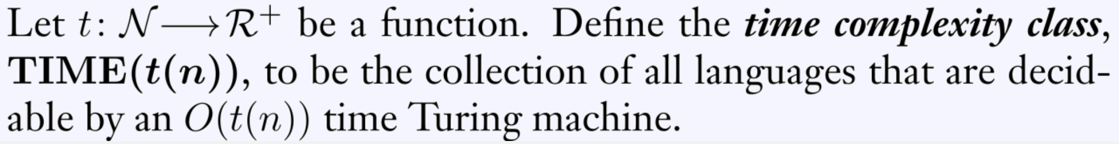
DEFINITION

Let $f\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. The ***space complexity classes***, **SPACE($f(n)$)** and **NSPACE($f(n)$)**, are defined as follows.

$$\text{SPACE}(f(n)) = \{L \mid L \text{ is a } \boxed{\text{language}} \text{ decided by an } O(f(n)) \text{ space deterministic Turing machine}\}.$$

$$\text{NSPACE}(f(n)) = \{L \mid L \text{ is a } \boxed{\text{language}} \text{ decided by an } O(f(n)) \text{ space nondeterministic Turing machine}\}.$$

Compare:



Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, **TIME($t(n)$)**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

$$\textbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

Example: *SAT* Space Usage

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

$2^{O(m)}$ exponential time machine

```
M_1 = "On input ⟨φ⟩, where φ is a Boolean formula:
    1. For each truth assignment to the variables $x_1, \ldots, x_m$ of $\phi$:
    2.     Evaluate $\phi$ on that truth assignment.
    3. If $\phi$ ever evaluated to 1, accept; if not, reject."
```

Each loop iteration requires $O(m)$ space

But the space is re-used on each loop!
(nothing is stored from the prev loop)

So this machine runs in $O(\boldsymbol{m})$ space complexity!

SAT is in $O(\boldsymbol{m})$ space complexity class!

Space is **"more powerful"** than time.

Example: Nondeterministic Space Usage

$$ALL_{\mathsf{NFA}} = \{\langle A\rangle \mid A \text{ is an NFA and } L(A) = \Sigma^*\}$$

Nondeterministic decider for $\overline{ALL_{\mathsf{NFA}}}$

(accepts NFAs that reject something)

```
N = "On input ⟨M⟩, where M is an NFA:
    1. Place a marker on the start state of the NFA.
    2. Repeat $2^q$ times, where $q$ is the number of states of $M$:
           Nondeterministically select an input symbol and change the
           positions of the markers on M's states to simulate reading
           that symbol.
    4. Accept if stages 2 and 3 reveal some string that M rejects; that
       is, if at some point none of the markers lie on accept states of
       M. Otherwise, reject."
```

Machine tracks "current" state(s) of NFA

q states = 2^q possible combinations (so exponential time)

But each loop uses only $O(\boldsymbol{q})$ space!

Additionally, need a counter to count to 2^q: this requires $\log(2^q) = \boldsymbol{q}$ extra space

So the whole machine runs in (nondeterministic) linear $O(\boldsymbol{q})$ space!

Facts About Time vs Space (for Deciders)

TIME → SPACE

- If a decider runs in time $\boxed{t(\boldsymbol{n})}$, then its maximum space usage is …
- … $\boxed{t(\boldsymbol{n})}$
- … because it can add at most 1 tape cell per step

What about deterministic vs non-deterministic?

SPACE → TIME

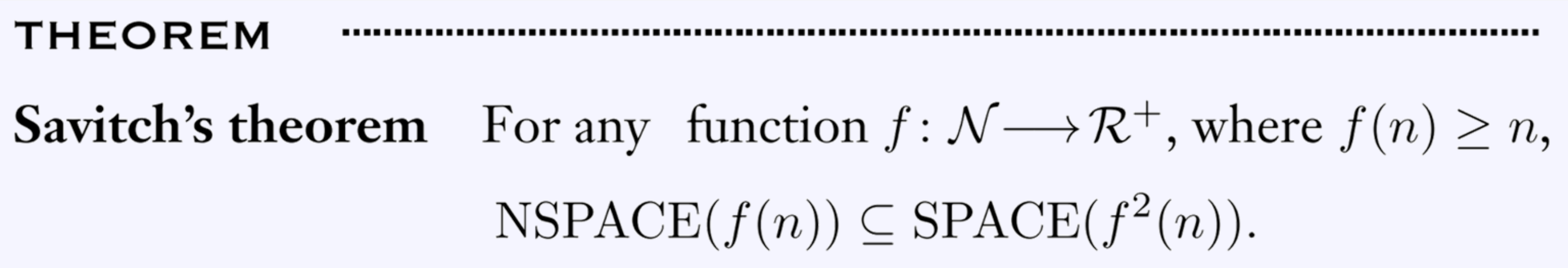
- If a decider runs in space $\boxed{f(\boldsymbol{n})}$, then its maximum time usage is …
- … $(|\Gamma| + |Q|)^{f(\boldsymbol{n})} = \boxed{2^{df(\boldsymbol{n})}}$
- … because that’s the number of possible configurations
- (and a decider cannot repeat a configuration)

Flashback: Deterministic vs Non-Det. Time

- If a non-deterministic TM runs in: $t(n)$ time
- Then an equivalent deterministic TM runs in: $2^{O(t(n))}$
    - **Exponentially** slower

What about space?

Deterministic vs Non-Det. Space



THEOREM

Savitch's theorem For any function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

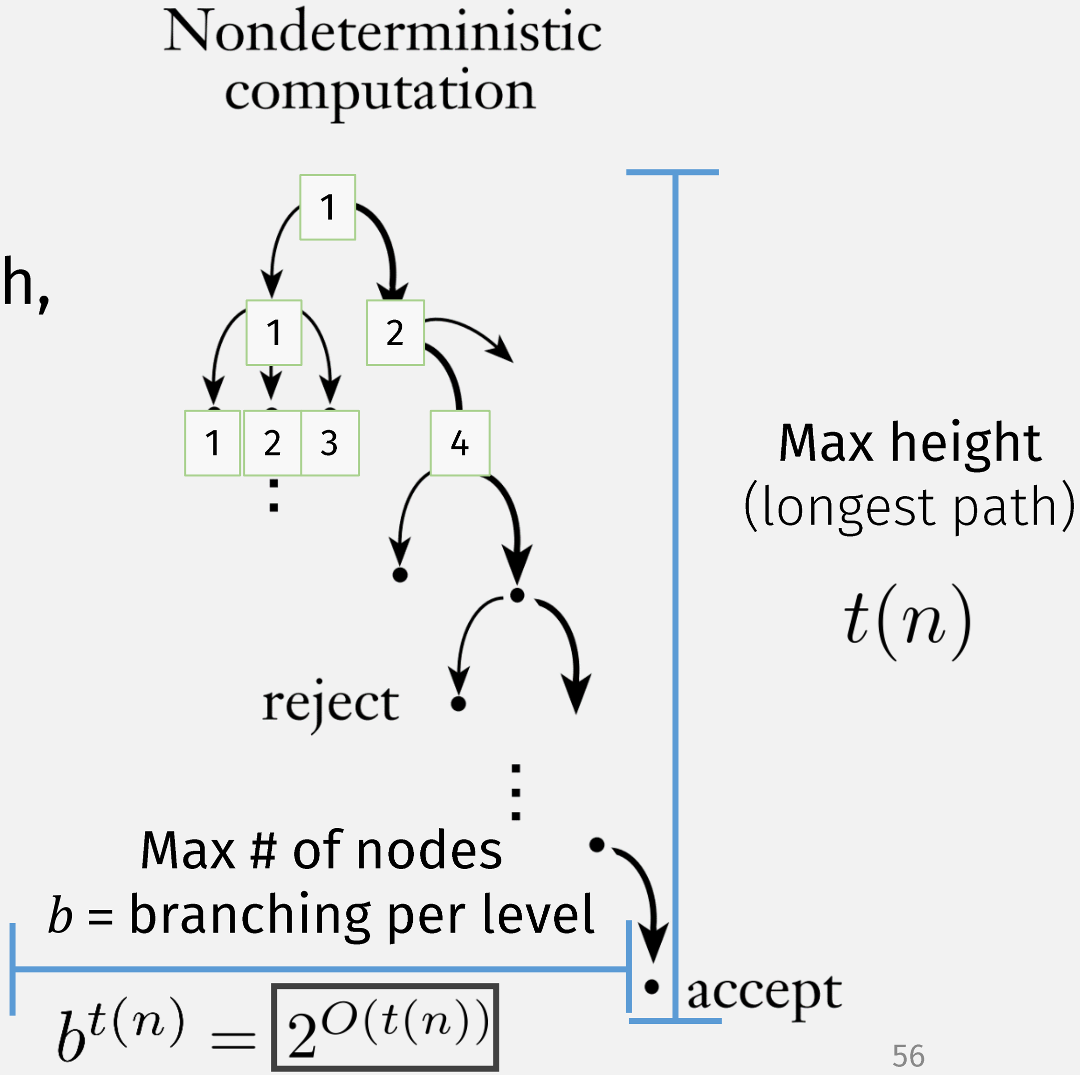
$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

- If a non-deterministic TM runs in: $f(n)$ space
- Then an equivalent deterministic TM runs in: $f^2(n)$ space
  - ~~**Exponentially**~~ Only **Quadratically** slower!

Flashback: Nondet → Deterministic TM: Time

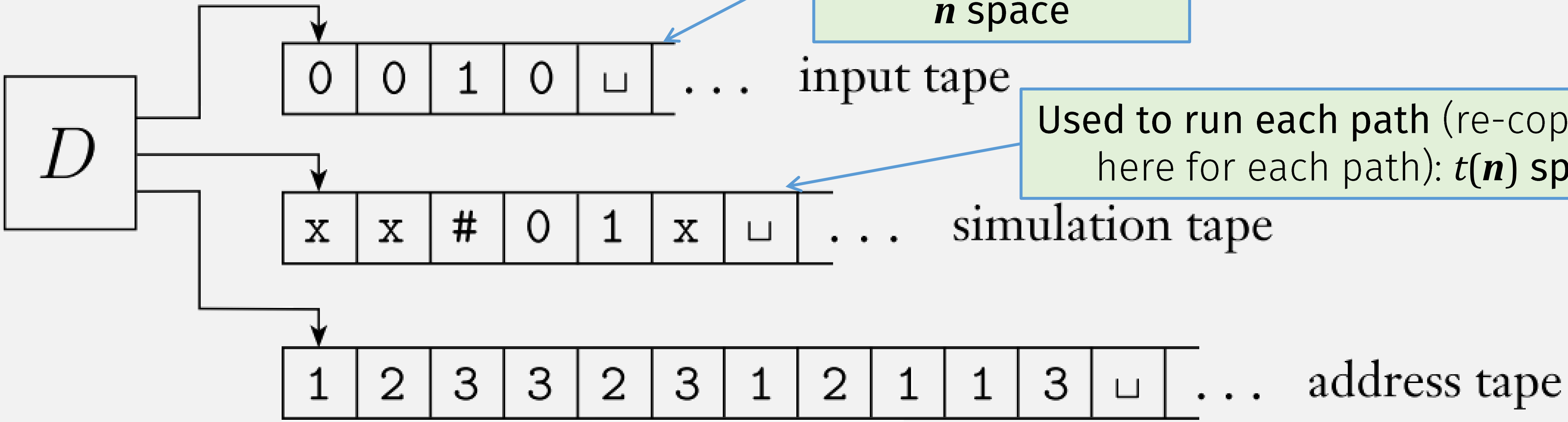
$t(n)$ time $\quad \boxed{2^{O(t(n))}}$ time

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - 1
    - 1-1
    - 1-2
    - **1-1-1**



Flashback: Nondet → Deterministic TM: Space

3 tapes

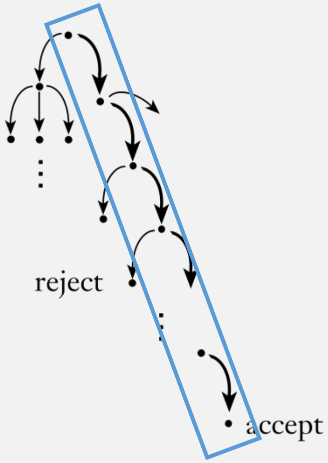


Always has input, never changes: n space

Used to run each path (re-copy input here for each path): $t(n)$ **space**

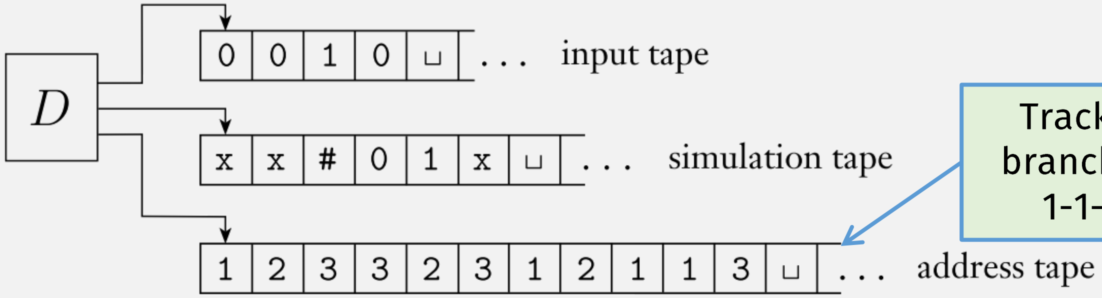
Tracks which node we are on, $2^{O(t(n))}$ (exponential) space??

Nondet → Deterministic TM: Space



Let N be an NTM deciding language A in space $f(n)$

- This means a single path could use $f(n)$ space
- That path could take $2^{df(n)}$ steps
    - (That's the possible ways to fill the space)
    - Each step could be a non-deterministic branch that must be saved
- So naïvely tracking these branches requires $2^{df(n)}$ space!



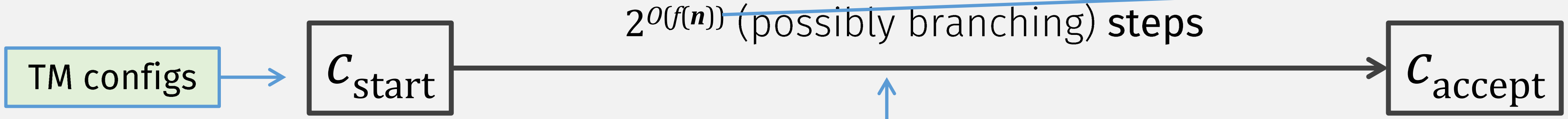
Tracks node branching e.g., 1-1-2, etc.

- Instead, let's "divide and conquer" to reduce space!

“Divide and Conquer” TM Config Sequences

- Want to check whether:

Number of splits: $\log(2^{O(f(n))}) = O(f(n))$

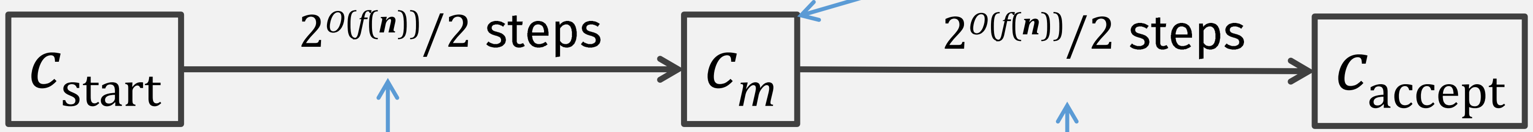


Remembering the branch at every step costs exponential space

Each split must remember a “c_m” config = $O(f(n))$ **space**

- Instead, we check whether:

So long as we save the intermediate config

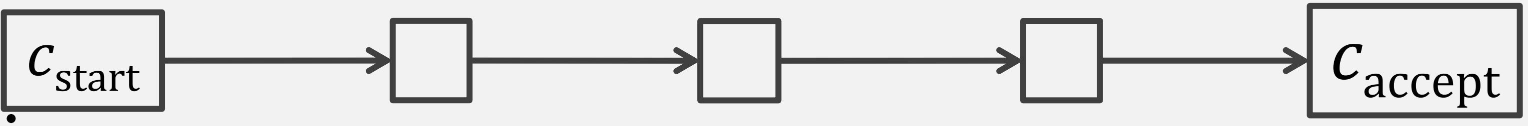


Remembering these steps costs half the space …

… and we can reuse that space to check the second half

Total:
$O(f(n)) * O(f(n))$
= $\boldsymbol{O(f^2(n))}$ **space**
(Savitch’s Thm)

- Keep dividing …



Formally: A “Yielding” Algorithm

Start config

End config

steps

Base case

```
CANYIELD = “On input $c_1$, $c_2$, and $t$:
1. If $t = 1$, then test directly whether $c_1 = c_2$ or whether $c_1$ yields
   $c_2$ in one step according to the rules of $N$. *Accept* if either test
   succeeds; *reject* if both fail.
2. If $t > 1$, then for each configuration $c_m$ of $N$ using space $f(n)$:
3.     Run CANYIELD($c_1, c_m, \frac{t}{2}$).
4.     Run CANYIELD($c_m, c_2, \frac{t}{2}$).
5.     If steps 3 and 4 both accept, then *accept*.
6. If haven’t yet accepted, *reject*.”
```

What’s the middle config? Try them all
(it doesn’t use any more space, per loop)

“divide and conquer”

Savitch’s Theorem: Proof

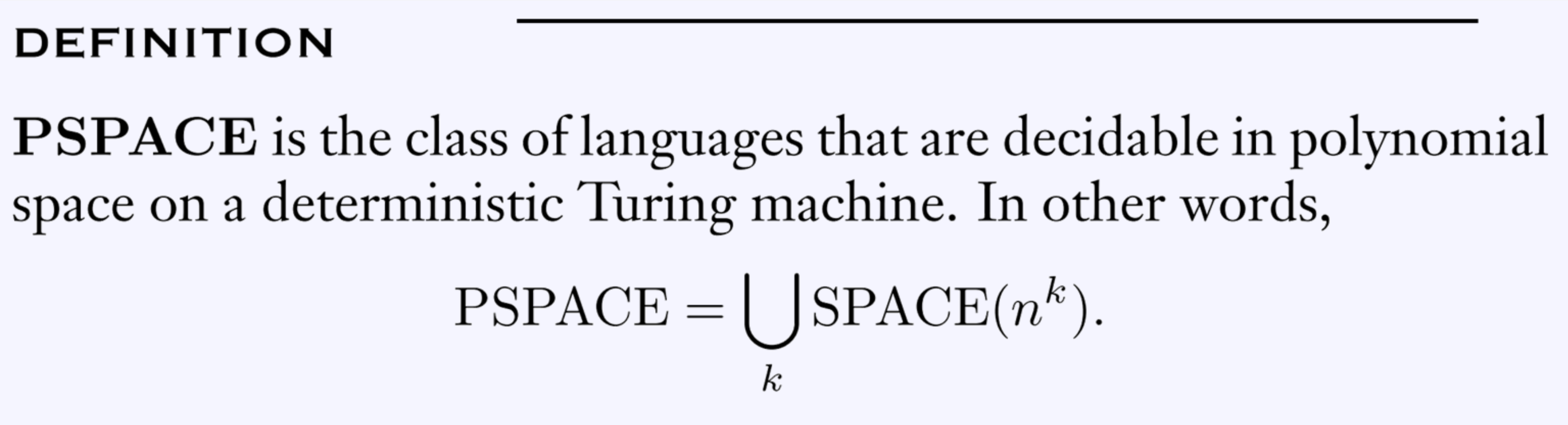
- Let N be an NTM deciding language A in space $f(n)$
- Construct equivalent deterministic TM M using $O(f^2(n))$ space:

```
$M$ = “On input $w$:
    1. Output the result of CANYIELD($c_{\text{start}}, c_{\text{accept}}, 2^{df(n)}$).”
```

Extra d constant depends on size of tape alphabet

  - c_{start} = start configuration of N
  - c_{accept} = new accepting config where all N’s accepting configs go

PSPACE



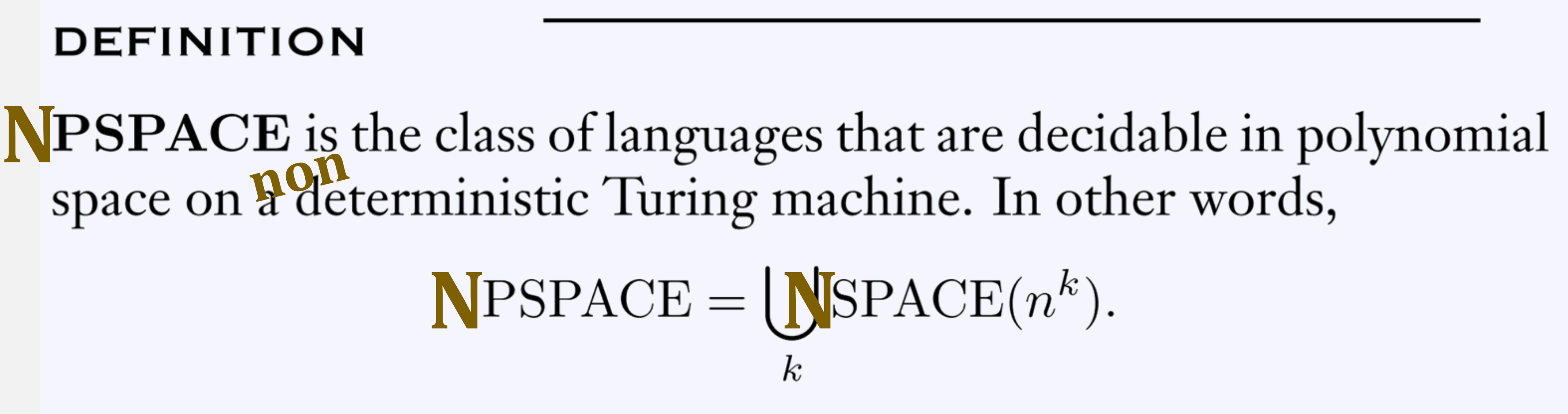
DEFINITION

PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

$$\mathrm{PSPACE} = \bigcup_k \mathrm{SPACE}(n^k).$$

NPSPACE

Analogous to **P** and **NP** for time complexity



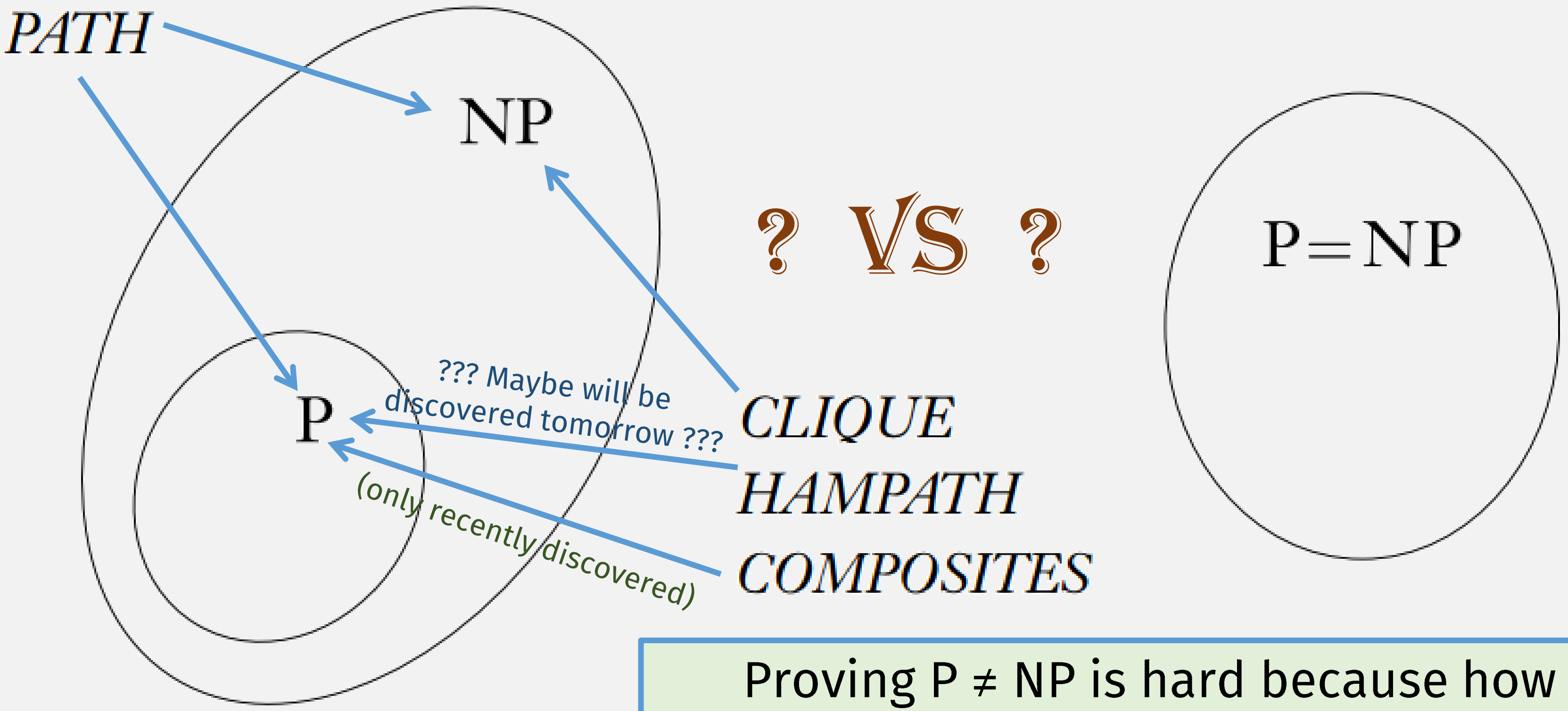
DEFINITION

NPSPACE is the class of languages that are decidable in polynomial space on a nondeterministic Turing machine. In other words,

$$\text{NPSPACE} = \bigcup_k \text{NSPACE}(n^k).$$

But **P** ⊆ **PSPACE** and **NP** ⊆ **NPSPACE**
- Because each step can use at most one extra tape cell
- But space can be re-used

Flashback: Does P = NP?



PATH

NP

P

??? Maybe will be discovered tomorrow ???

(only recently discovered)

CLIQUE

HAMPATH

COMPOSITES

? VS ?

P = NP

Proving P ≠ NP is hard because how do you prove an algorithm <u>doesn't</u> have a poly time algorithm?
(in general it's hard to prove that something <u>doesn't</u> exist)

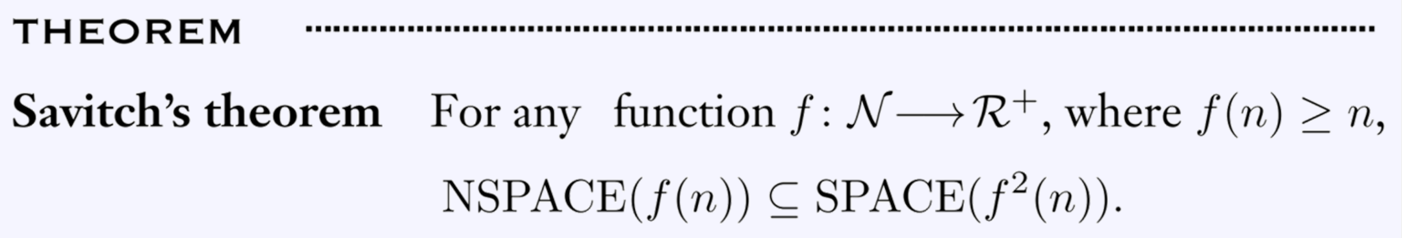
PSPACE = NPSPACE ?

- **PSPACE:** langs decidable in poly space on deterministic TM

- **NPSPACE:** langs decidable in poly space on nondeterministic TM

Theorem: **PSPACE = NPSPACE** !!!

Proof: By Savitch's Theorem!



THEOREM

Savitch's theorem For any function $f : \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

Space vs Time

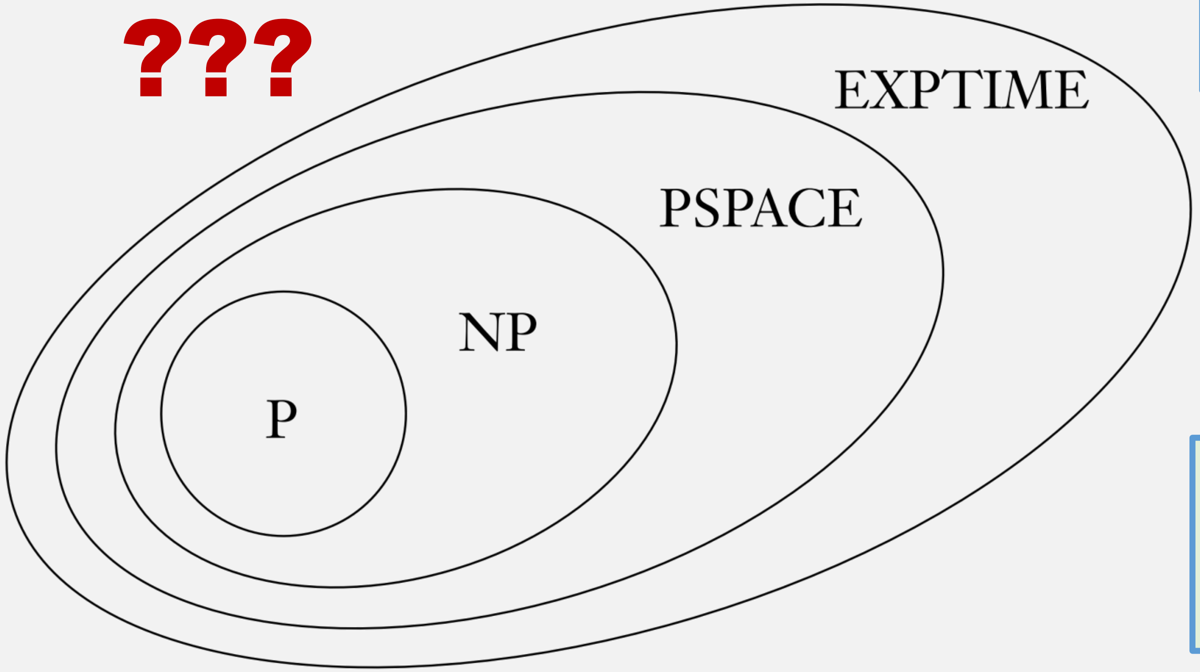
- $\mathbf{P} \subseteq \mathbf{PSPACE}$ and $\mathbf{NP} \subseteq \mathbf{NPSPACE}$
  - Because each step can use at most one extra tape cell
  - And space can be re-used

- $\mathbf{PSPACE} \subseteq \mathbf{EXPTIME}$
  - Because an $f(n)$ space TM has $2^{O(f(n))}$ possible configurations
  - And a halting TM cannot repeat a configuration

- We already know $\mathbf{P} \subseteq \mathbf{NP}$ and $\mathbf{PSPACE} = \mathbf{NPSPACE}$ … so:

$$\boxed{\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} = \mathbf{NPSPACE} \subseteq \mathbf{EXPTIME}}$$

Space vs Time: Conjecture



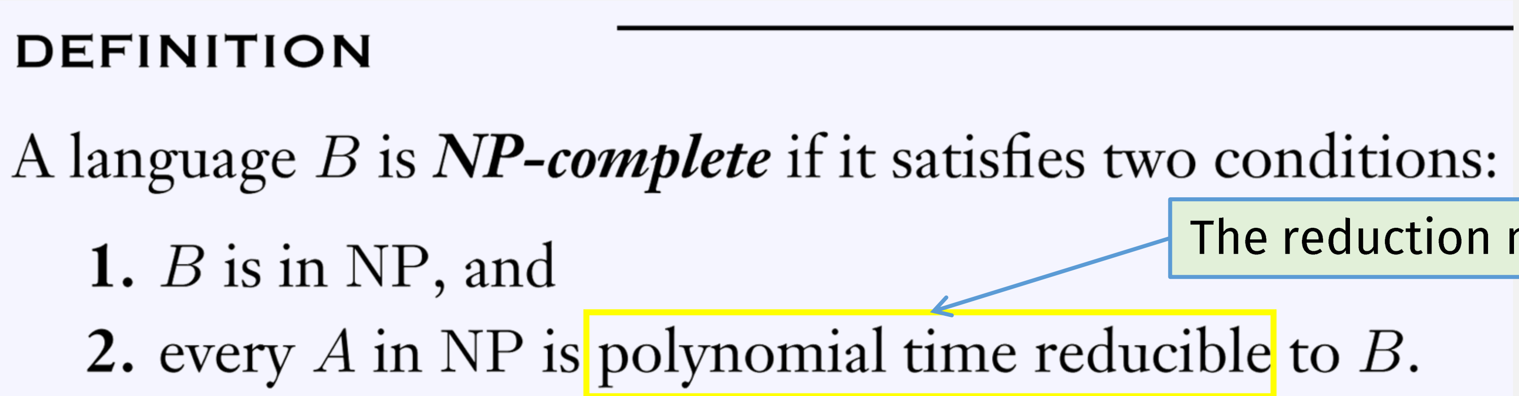
Researchers believe these are all completely contained within each other

But this is an open conjecture!

The only progress so far is:
P ⊂ EXPTIME
(we will prove next week)

P ⊂ NP ⊂ PSPACE = NPSPACE ⊂ EXPTIME

Review: **NP**-Completeness



DEFINITION

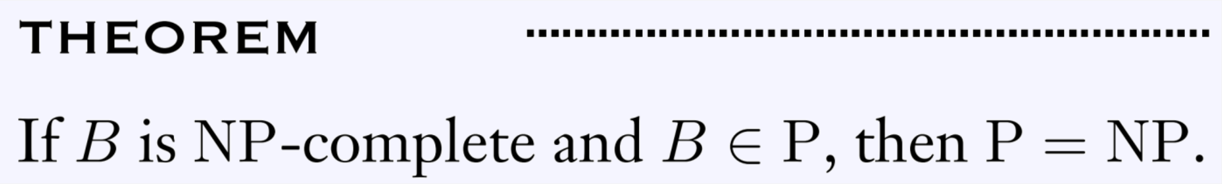
A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

The reduction must be “easy”

These are the “hardest” problems (in **NP**) to solve

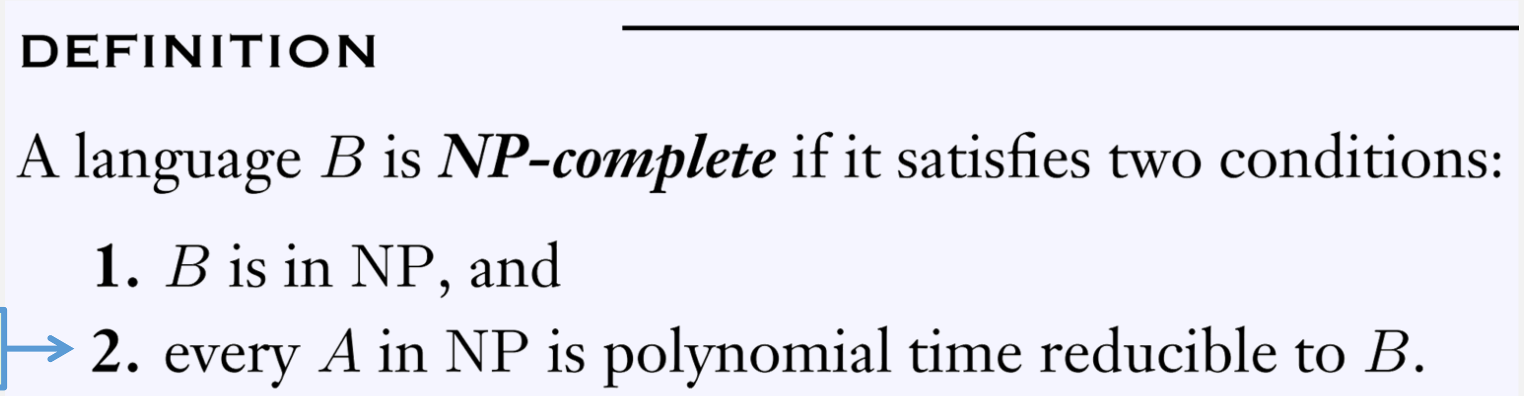
Potentially helps answer **P=NP**? question



THEOREM

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

NP-Completeness vs **NP**-Hardness



DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

"**NP**-Hard" → 2.

"**NP**-Complete" = in **NP** + "**NP**-Hard"

So a language can be **NP**-hard but not **NP**-complete!

The Halting Problem is **NP**-Hard

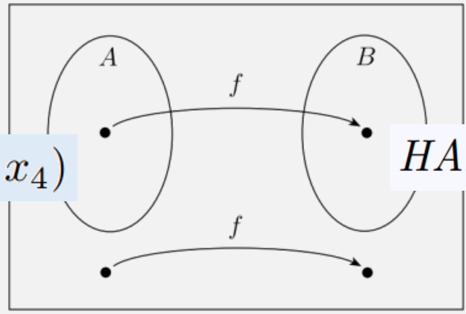
$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

<u>Proof</u>: Reduce *3SAT* to the Halting Problem

(Why does this prove that the Halting Problem is **NP**-hard?)

Because *3SAT* is **NP**-complete! (so every **NP** problem is poly time reducible to *3SAT*)

$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$



$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

The Halting Problem is **NP**-Hard

$HALT_{\mathsf{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

Computable function, from $3SAT \rightarrow HALT_{\mathsf{TM}}$:

On input ϕ, a formula in 3cnf:

- Construct TM M

M = on input ϕ

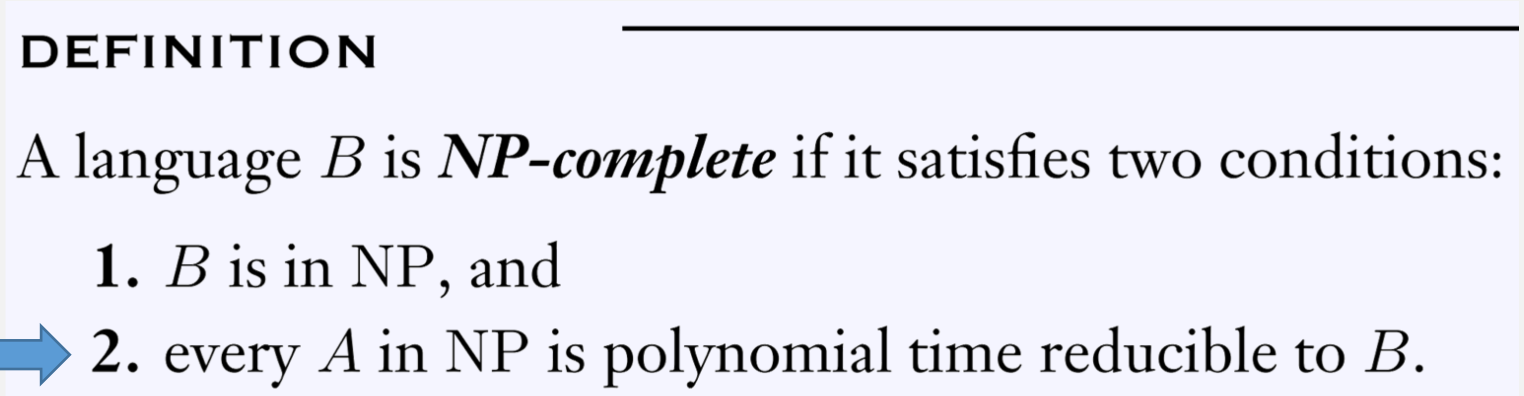
- Try all assignments
  - If any satisfy ϕ, then accept
- When all assignments have been tried, start over

This loops when there is no satisfying assignment!

- Output $<M, \phi>$

⇒ If ϕ has a satisfying assignment, then M halts on ϕ
⇐ If ϕ has no satisfying assignment, then M loops on ϕ

Review:

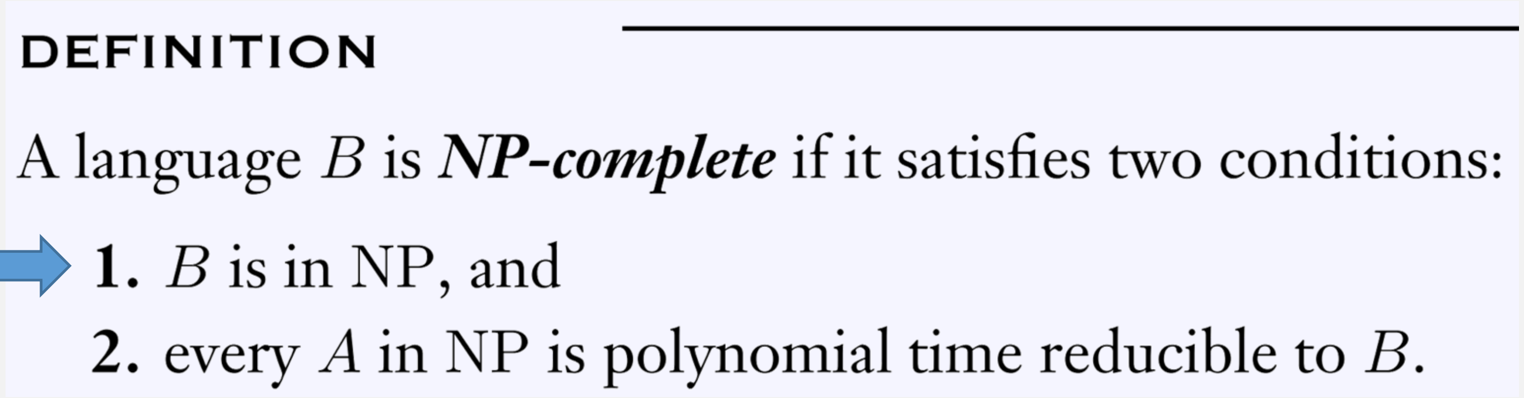

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So a language can satisfy only condition #2

Review:



DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So a language can satisfy only condition #2

Can a language satisfy only condition #1?

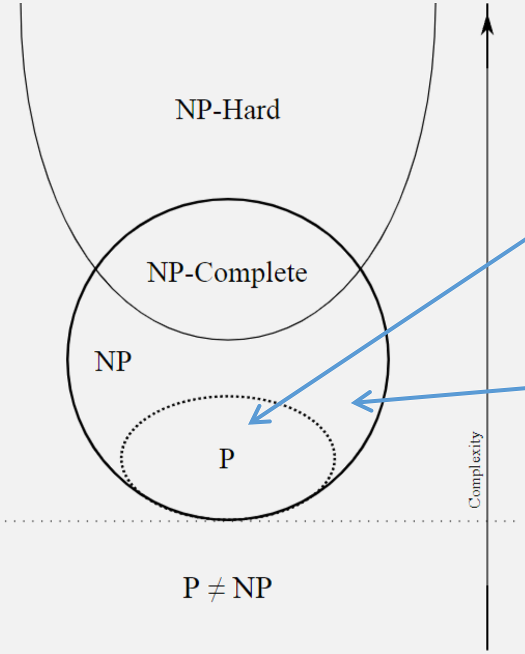
Yes, every language in **P** ...

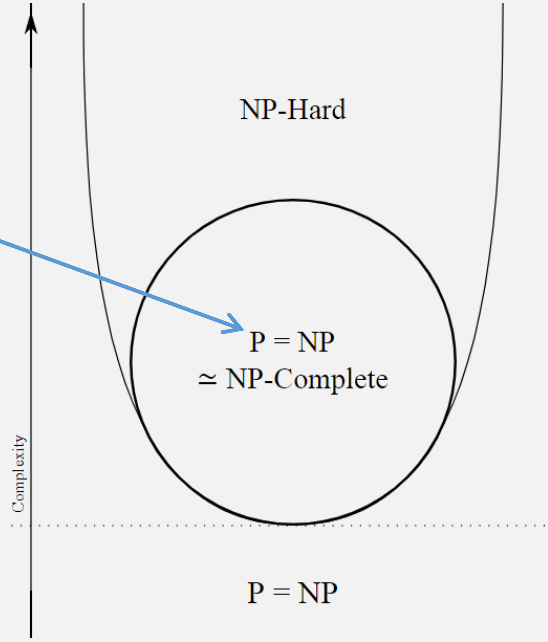
(... unless **P** = **NP**)

Can a non-**P** language satisfy only condition #1?

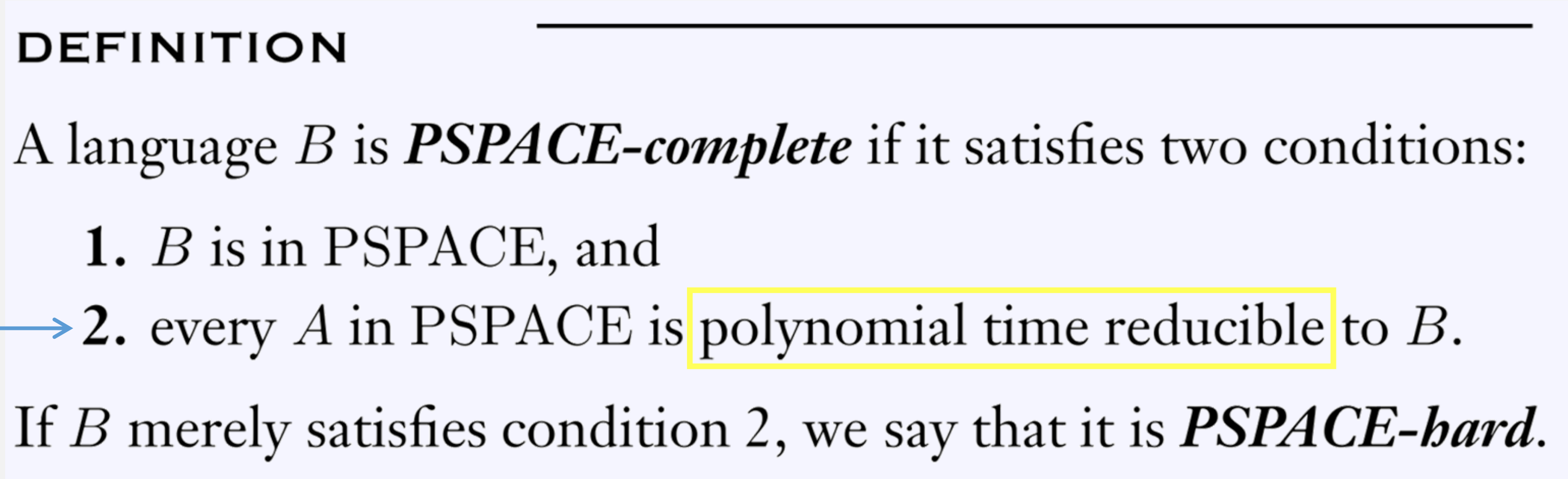
Yes ...

... but that implies **P** ≠ **NP**,
so it's not known for sure





PSPACE-Completeness



DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

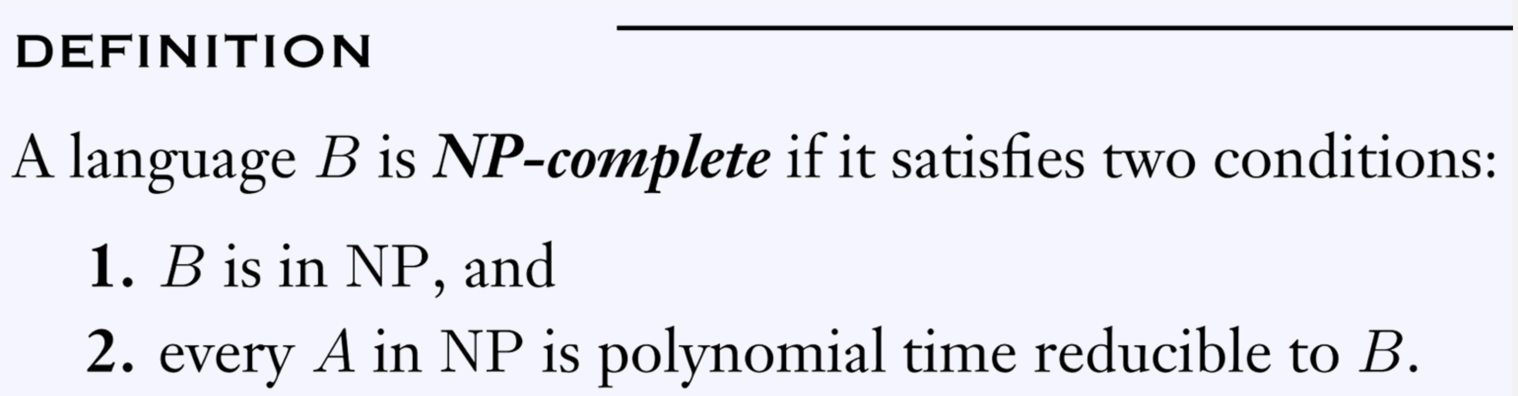
1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

Condition #2 hard to prove the first time

The reduction must still be “easy”

Flashback: **NP**-Completeness

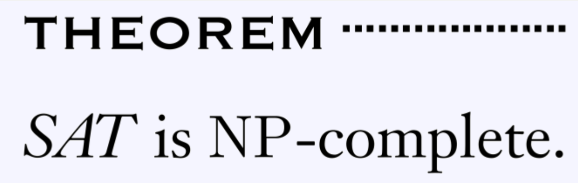


DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

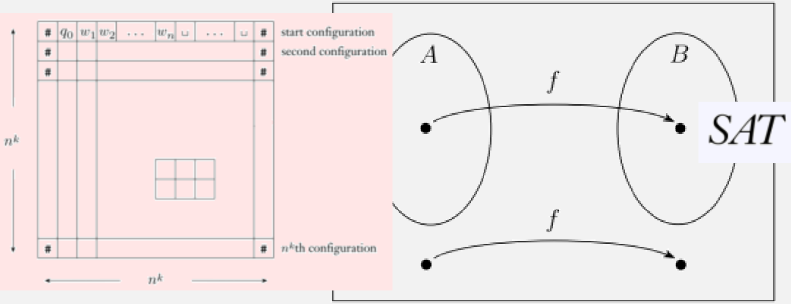
1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

The first **NP**-complete problem:



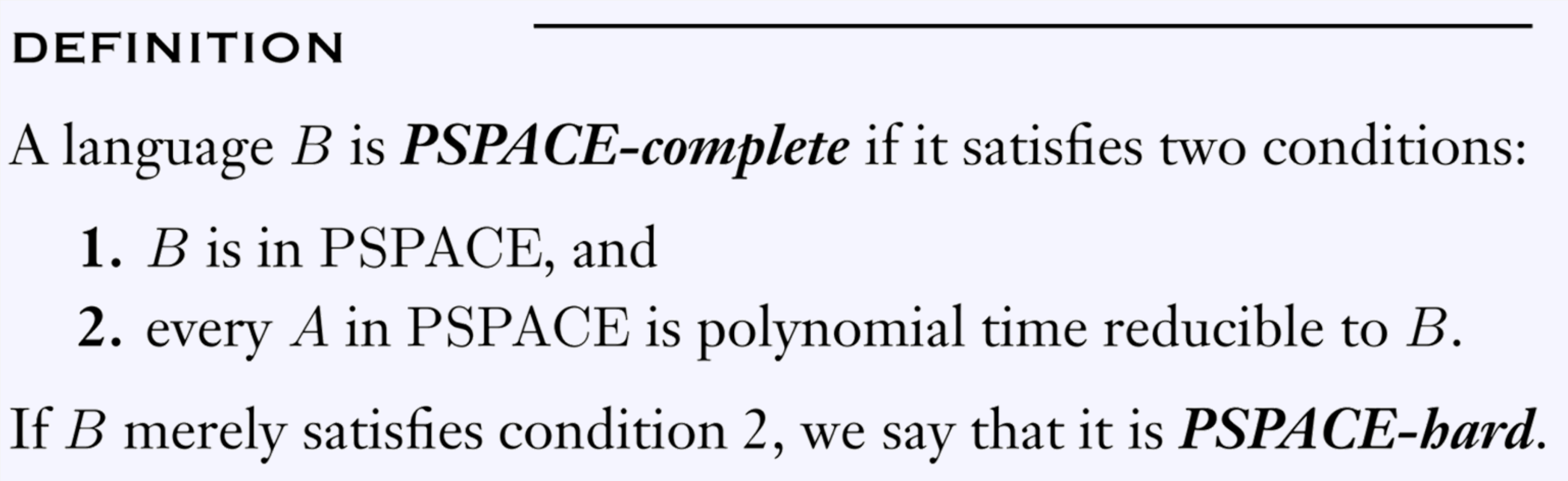
THEOREM

SAT is NP-complete.



$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

PSPACE-Completeness



DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

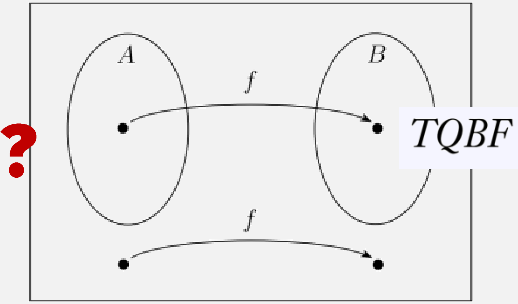
If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

The first **PSPACE**-complete problem:

THEOREM

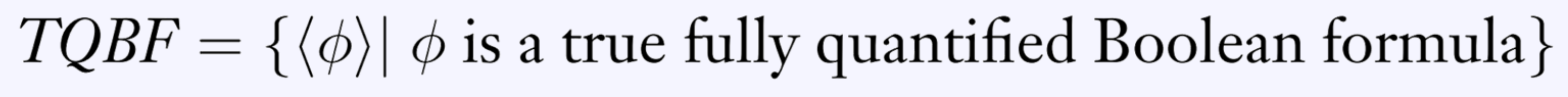
TQBF is PSPACE-complete.

???



$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

TQBF

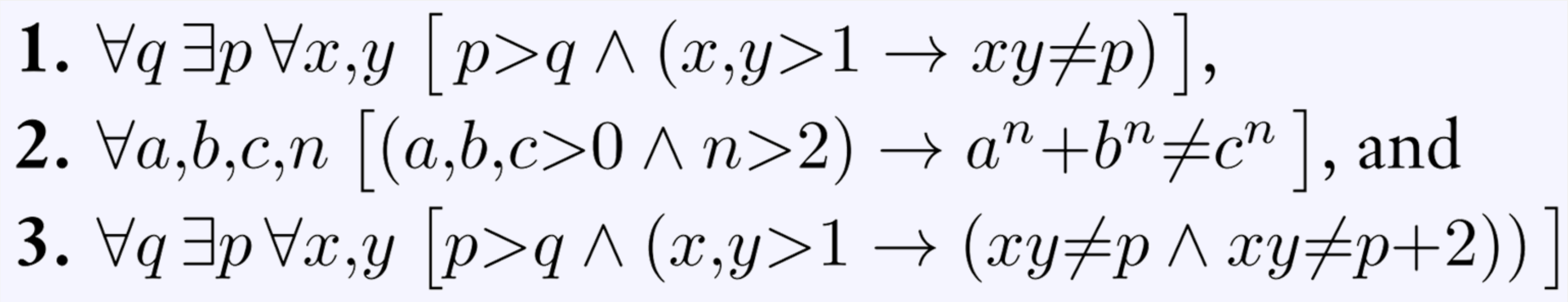


$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

Flashback: Boolean Formulas

A Boolean _____	**Is …**	**Example:**
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	Literals ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$

Flashback: The Language of Math Statements



1. $\forall q\,\exists p\,\forall x,y\ \big[\,p>q \wedge (x,y>1 \rightarrow xy\neq p)\,\big]$,

2. $\forall a,b,c,n\ \big[(a,b,c>0 \wedge n>2) \rightarrow a^n+b^n\neq c^n\,\big]$, and

3. $\forall q\,\exists p\,\forall x,y\ \big[p>q \wedge (x,y>1 \rightarrow (xy\neq p \wedge xy\neq p+2))\,\big]$

Flashback: Mathematical Statements Alphabet

- Strings in the language are drawn from the following chars:

$\wedge, \vee, \neg$	Boolean operations
$(,), [,]$	parentheses
$\forall, \exists$	quantifiers
x	variables
$R_1, \ldots, R_k$	Relation symbols

Flashback: Formulas and Sentences

- A mathematical statement is <u>well-formed</u>, i.e., a **formula**, if it's:
  - an atomic formula: $R_i(x_1, \ldots, x_k)$
  - $\phi_1 \wedge \phi_2$
  - $\phi_1 \vee \phi_2$
  - $\neg\phi$
    - where ϕ, ϕ_1, and ϕ_2 are formulas
  - $\forall x\ [\ \phi\ ]$
  - $\exists x\ [\ \phi\ ]$
    - where ϕ is a formula
    - x's "scope" is in the following brackets
    - A <u>free variable</u> is a variable that is outside the scope of a quantifier

$$R_1(x_1) \wedge R_2(x_1, x_2, x_3)$$

$$\forall x_1\ \left[\ R_1(x_1) \wedge R_2(x_1, x_2, x_3)\ \right]$$

$$\boxed{\forall x_1\ \exists x_2\ \exists x_3\ \left[\ R_1(x_1) \wedge R_2(x_1, x_2, x_3)\ \right]}$$

- A **sentence** is a formula with <u>no free variables</u>

Flashback: Universes, Models, and Theories

- A **universe** is the set of values that variables can represent
  - E.g., the universe of the natural numbers
  - Boolean Formulas use values from the universe of {True, False}

- A **model** is:
  1. a universe, and
  2. an assignment of relations to relation symbols, e.g., AND, OR, NOT

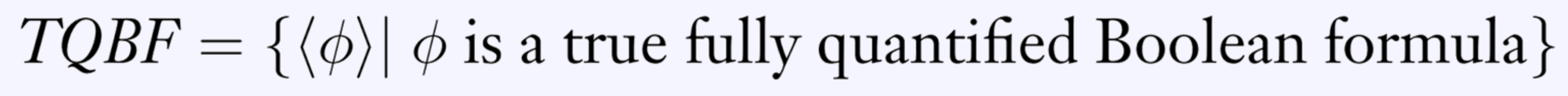
- A **theory** is the set of all true sentences in a model's language

Quantified Boolean Formulas

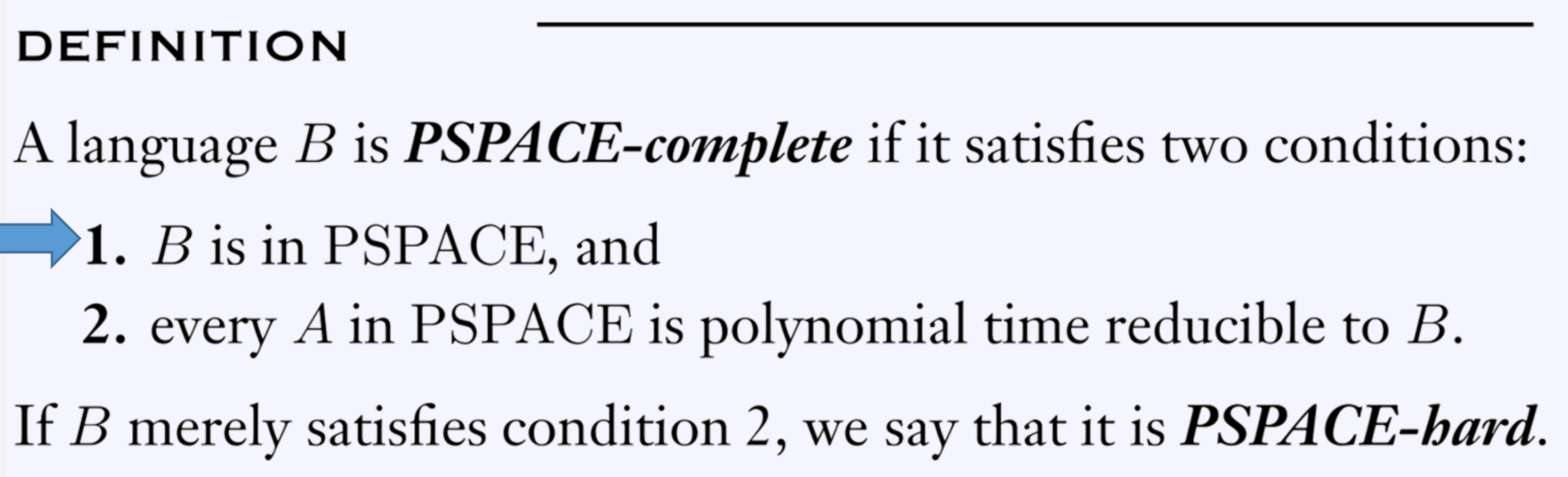
A Boolean _____	**Is …**	**Example:**
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	Literals ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
Quantifiers	∃ or ∀	
Quantified Formula	Formula with quantifiers	$\phi = \forall x \, \exists y \, \big[(x \vee y) \wedge (\overline{x} \vee \overline{y}) \big]$
Fully Quantified Formula	Sentence, no free vars	

THEOREM

TQBF is PSPACE-complete.



$TQBF = \{\langle \phi \rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$



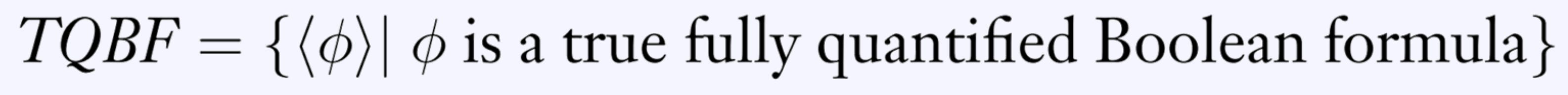
DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

1. B is in PSPACE, and
2. every A in PSPACE is polynomial time reducible to B.

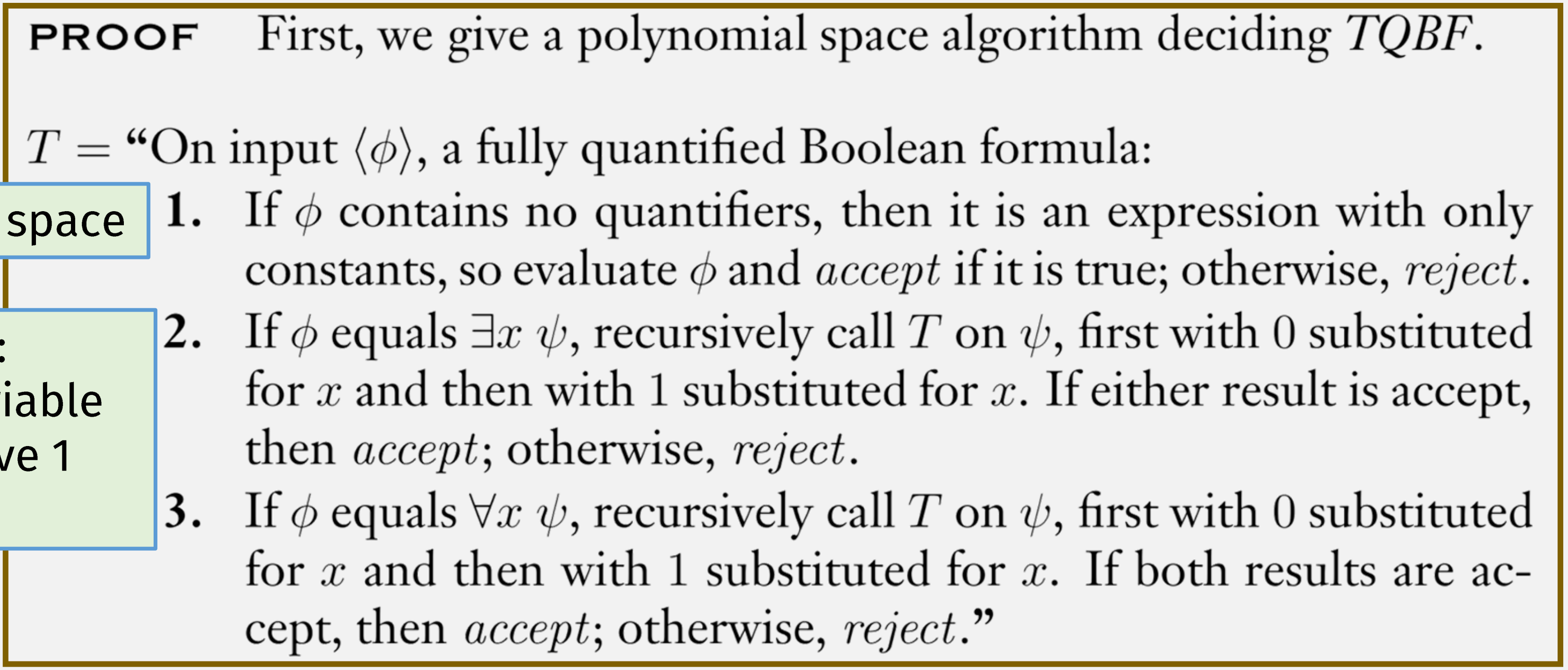
If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

TQBF is in **PSPACE**



$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

Let $\boldsymbol{m}$ = # variables in formula



PROOF First, we give a polynomial space algorithm deciding *TQBF*.

T = "On input $\langle\phi\rangle$, a fully quantified Boolean formula:

1. If ϕ contains no quantifiers, then it is an expression with only constants, so evaluate ϕ and *accept* if it is true; otherwise, *reject*.
2. If ϕ equals $\exists x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If either result is accept, then *accept*; otherwise, *reject*.
3. If ϕ equals $\forall x\ \psi$, recursively call T on ψ, first with 0 substituted for x and then with 1 substituted for x. If both results are accept, then *accept*; otherwise, *reject*."

Base case: $O(\boldsymbol{m})$ space

Recursive calls:
- 2 for each variable
- each time, save 1 bool value

At most $\boldsymbol{m}$ recursive calls, so $O(\boldsymbol{m})$ space

THEOREM

TQBF is PSPACE-complete.

$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

☑ **1.** B is in PSPACE, and

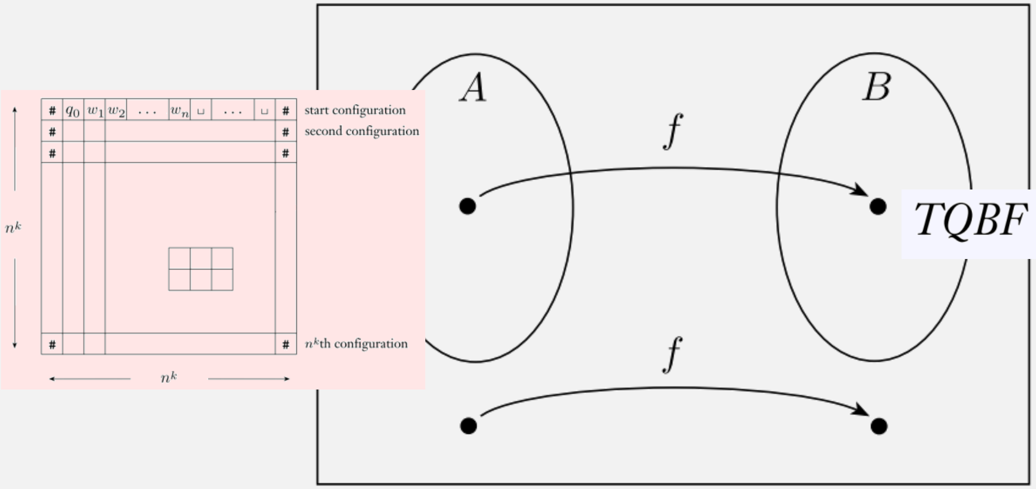
→ **2.** every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

TQBF is **PSPACE**-Hard

$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

<u>Idea</u>: Imitate Cook-Levin Theorem



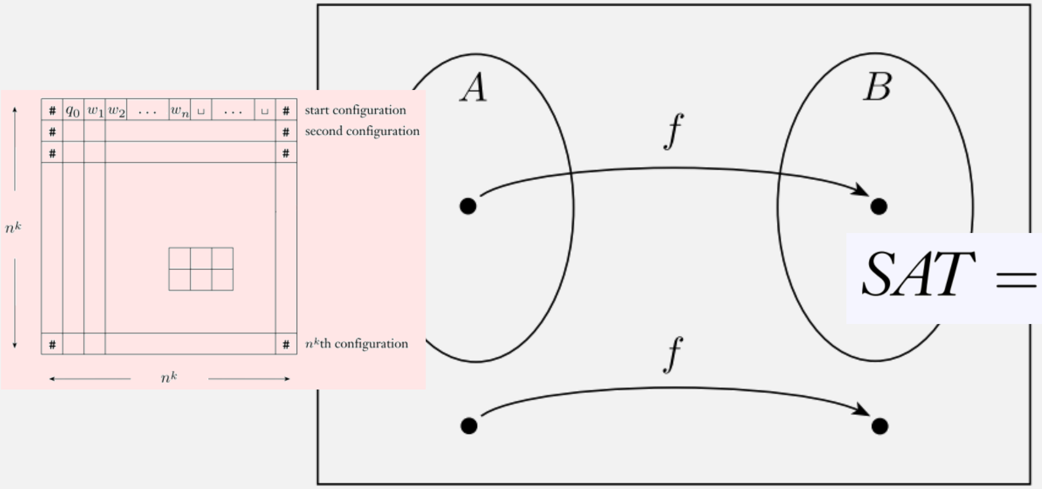
$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

Flashback: SAT is NP-complete

- Proof idea:
  - Give an algorithm that reduces accepting tableaus to satisfiable formulas

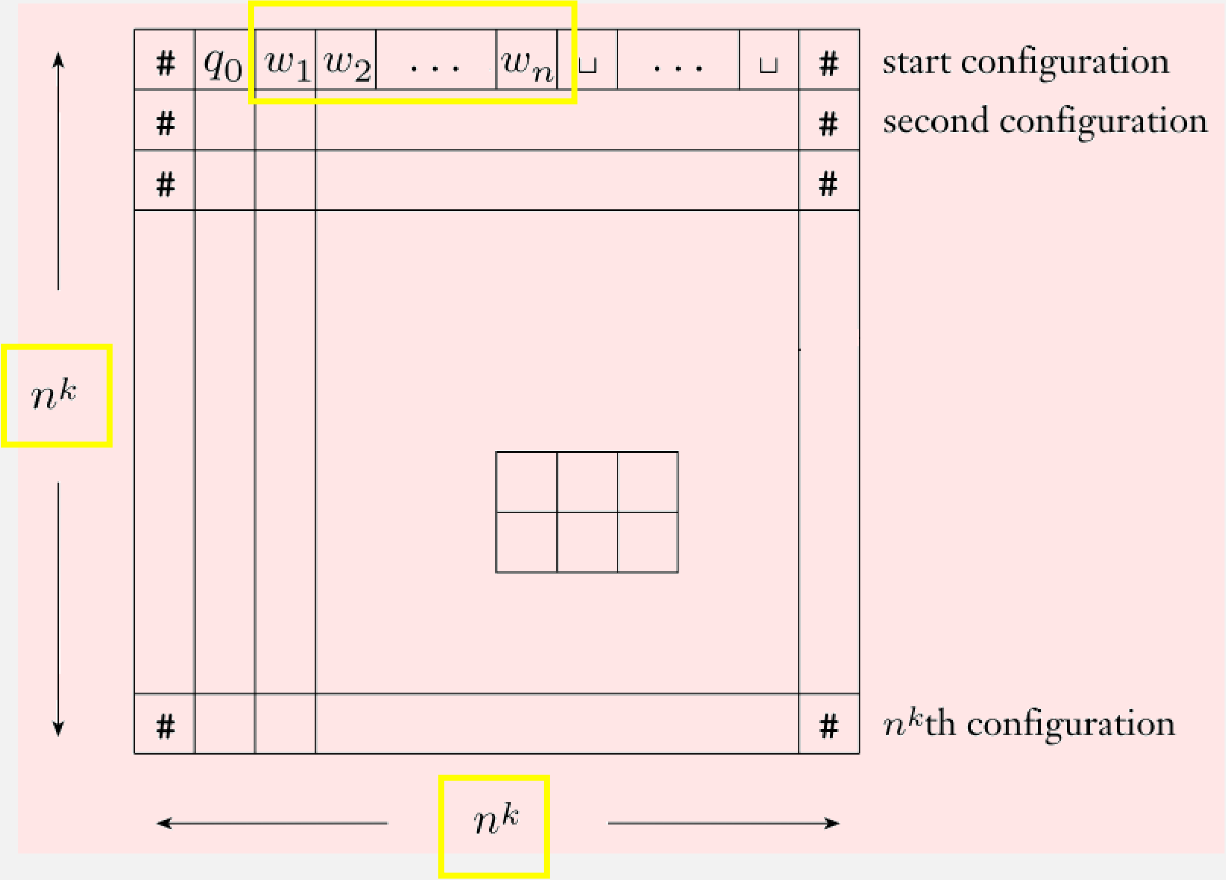
- Thus every string in the **NP** lang will be mapped to a sat. formula
  - and vice versa

Resulting formulas will have four components:
$\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$



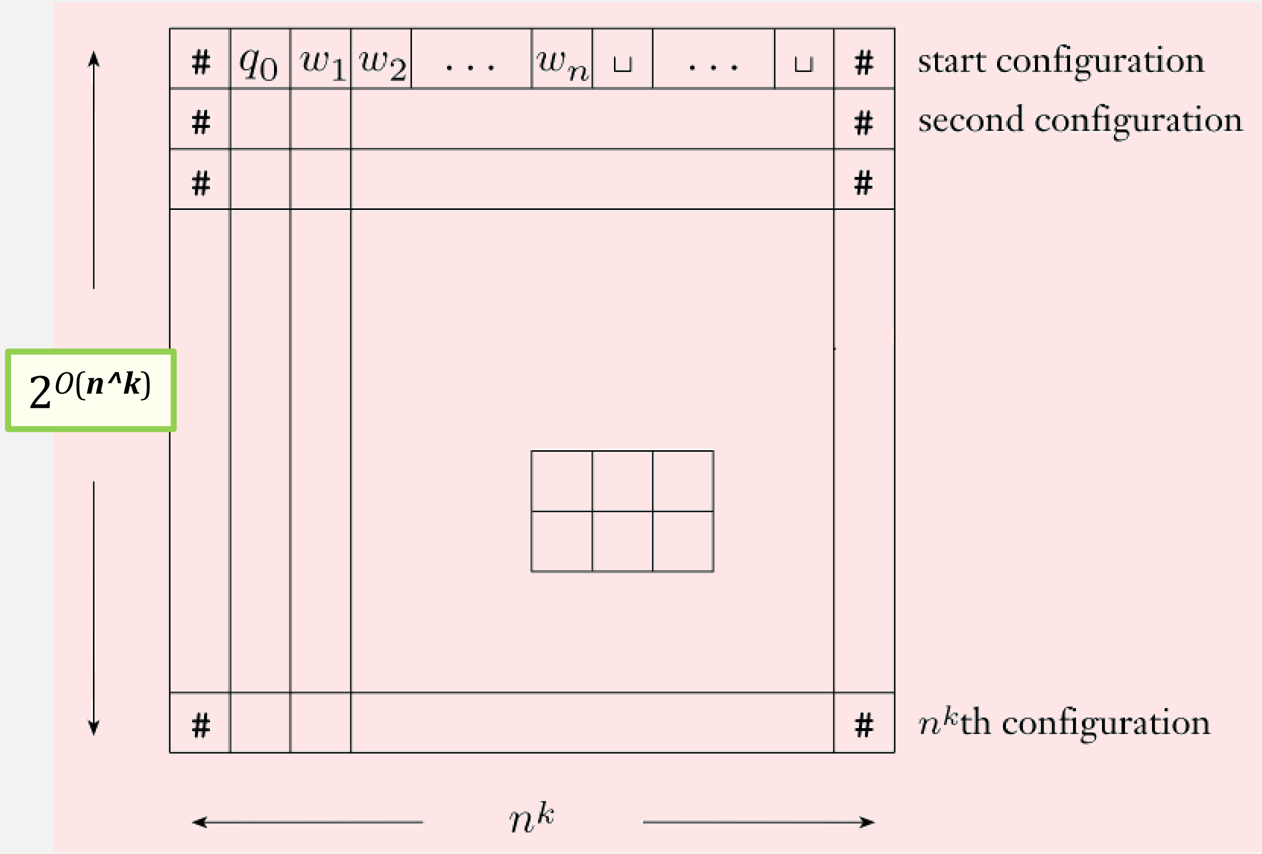
$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

An **NP** "Tableau"



- input $w = w_1 \ldots w_n$
- At most $\boldsymbol{n^k}$ configs
  - (why?)
- Each config has length $\boldsymbol{n^k}$
  - (why?)

A **PSPACE** “Tableau”



- input $w = w_1 \ldots w_n$
- At most $\boxed{2^{O(n^k)}}$ configs
  - (why?)
- Each config has length $\boldsymbol{n^k}$
  - (why?)

$\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$

Converting this to a formula would take <u>exponential</u> space and time!

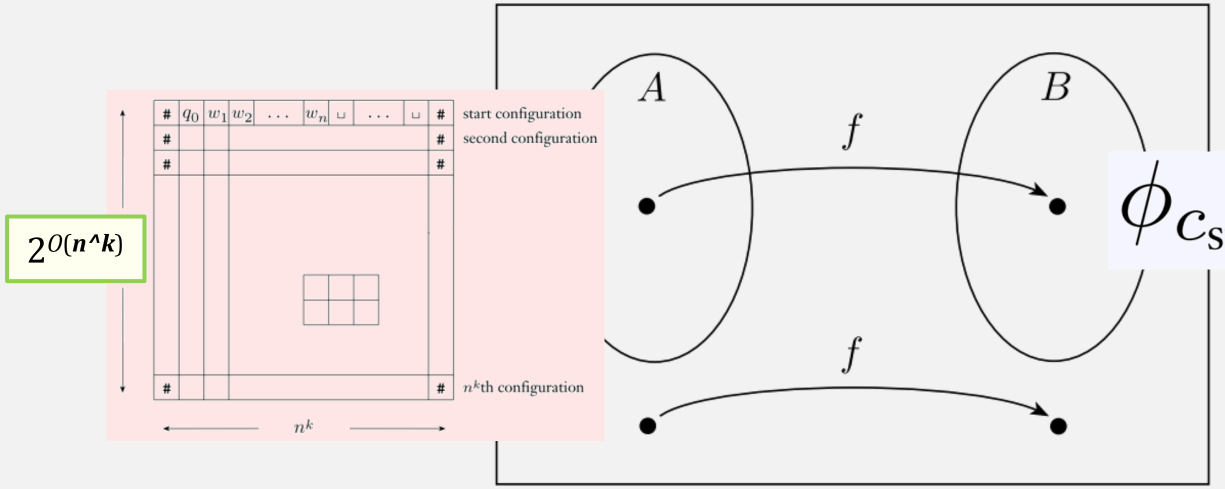
TQBF is **PSPACE**-Hard

$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

<u>Another Idea</u>: use quantifiers to “divide and conquer”

(like we did for Savitch’s Theorem)

Let $f(n) = n^k$ be the space usage of the TM



$2^{O(n^k)}$

$\phi_{c_{\text{start}}, c_{\text{accept}}, 2^{df(n)}}$

Recursively Defined Formulas: Try # 1

$$\phi_{c_1,c_2,t} = \exists m_1 \left[\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}\right]$$

t halved, but formula doubles in size
(two subformulas)

Doesn’t work! Still exponential!

Recursively Defined Formulas: Try # 2

~~$\phi_{c_1,c_2,t} = \exists m_1 \left[\phi_{c_1,m_1,\frac{t}{2}} \wedge \phi_{m_1,c_2,\frac{t}{2}}\right]$~~

$$\phi_{c_1,c_2,t} = \exists m_1 \, \forall (c_3,c_4) \in \{(c_1,m_1),(m_1,c_2)\} \left[\phi_{c_3,c_4,\frac{t}{2}}\right]$$

What's this?

Use ∀ quantifier to consolidate formula size

$\forall x \in \{y,z\} \, [\ldots]$ Shorthand for $\forall x \, [\, (x = y \vee x = z) \rightarrow \ldots]$

(And =, → can be converted to AND and OR)

t = $2^{O(f(n))}$,
so # halvings (subformulas) = $\log(2^{O(f(n))}) = O(f(n))$

t halved,
only 1 subformula

What do the base case subformulas look like?

Recursively Defined Formulas: Base Case

$$\phi_{c_1,c_2,t}$$

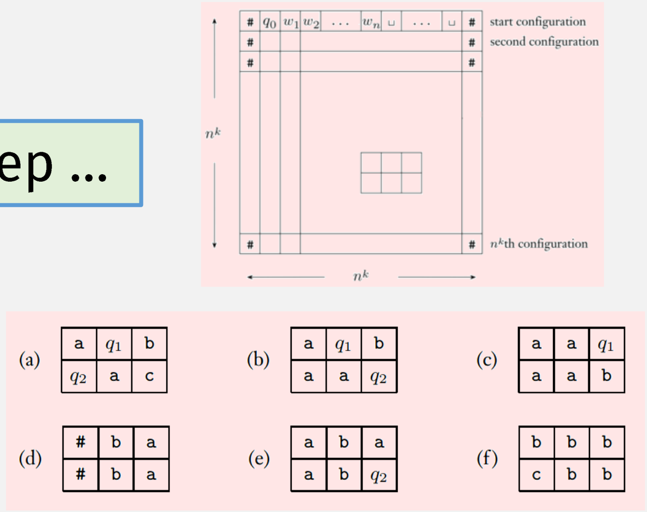
$t = 1$

This formula must encode that $c_1 \rightarrow c_2$ is a valid TM step ...

... using the same encoding as Cook-Levin!

Size of this subformula = $O(f(\boldsymbol{n}))$

Total size of all subformulas = $O(f(\boldsymbol{n})) * O(f(\boldsymbol{n})) = O(f^2(\boldsymbol{n}))$



TQBF is **PSPACE**-Hard

$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

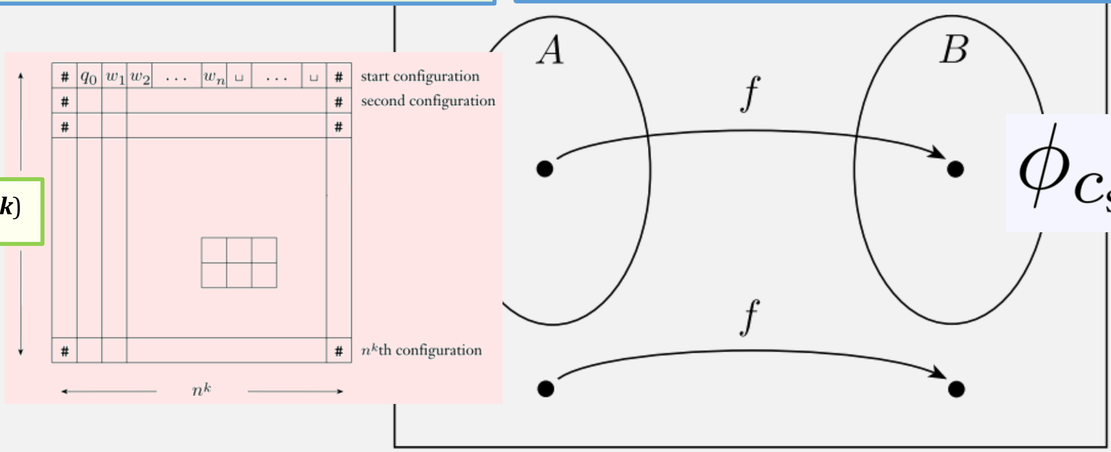
Another Idea: use quantifiers to "divide and conquer"

(like we did for Savitch's Theorem)

⇒ If M accepts w, then the formula is TRUE
- Because formula encodes accepting config seqs

⇐ If M rejects w, then the formula is FALSE
- Because there's no config seq reaching accept state

Let $f(n) = n^k$ be the space usage of the TM M deciding some language A



$2^{O(n^k)}$

$\phi_{c_{\text{start}}, c_{\text{accept}}, 2^{df(n)}}$

THEOREM

$TQBF$ is PSPACE-complete.

$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified Boolean formula}\}$

DEFINITION

A language B is ***PSPACE-complete*** if it satisfies two conditions:

☑ **1.** B is in PSPACE, and
☑ **2.** every A in PSPACE is polynomial time reducible to B.

If B merely satisfies condition 2, we say that it is ***PSPACE-hard***.

■

i.e., the “hardest” problem in **PSPACE**

Check-in Quiz 11/29

On gradescope