Complexity Theory: Zooming Out

Eric Price

UT Austin

CS 331, Spring 2020 Coronavirus Edition

Class Outline

1. Complexity classes
2. Computability

A few complexity classes

- P: Polynomial time

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.
- PSPACE: Polynomial space

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.
- PSPACE: Polynomial space
- NPSPACE: Nondeterministic, polynomial space

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.
- PSPACE: Polynomial space
- NPSPACE: Nondeterministic, polynomial space
  - NPSPACE $=$ PSPACE: try all proofs.

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.
- PSPACE: Polynomial space
- NPSPACE: Nondeterministic, polynomial space
  - NPSPACE $=$ PSPACE: try all proofs.
- EXP: Exponential time

A few complexity classes

- P: Polynomial time
- NP: Nondeterministic polynomial time
- PP: failure probability $< 1/2$.
  - Kind of silly: $NP \subseteq PP$ (guess x; if $f(x)$ true, return True; if $f(x)$ false, flip a coin)
- BPP: Probabilistic polynomial time, failure probability at most $1/3$.
- BQP: Probabilistic quantum polynomial time, failure probability at most $1/3$.
- PSPACE: Polynomial space
- NPSPACE: Nondeterministic, polynomial space
  - NPSPACE $=$ PSPACE: try all proofs.
- EXP: Exponential time
- NEXP: Nondeterministic exponential time

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \dots$$

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \ldots$$

- Know: $P \neq EXP$, $PSPACE \neq EXPSPACE$.

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \ldots$$

- Know: $P \neq EXP$, $PSPACE \neq EXPSPACE$.
- That's about it.

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \dots$$

- Know: $P \neq EXP$, $PSPACE \neq EXPSPACE$.
- That's about it.

$$P \subseteq BPP \subseteq BQP \subseteq PSPACE$$

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \ldots$$

- Know: $P \neq EXP$, $PSPACE \neq EXPSPACE$.
- That's about it.

$$P \subseteq BPP \subseteq BQP \subseteq PSPACE$$

- Most people expect: $P = BPP$, everything else $\subsetneq$.

Relations of complexity classes

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \ldots$$

- Know: $P \neq EXP$, $PSPACE \neq EXPSPACE$.
- That's about it.

$$P \subseteq BPP \subseteq BQP \subseteq PSPACE$$

- Most people expect: $P = BPP$, everything else $\subsetneq$.
- Don't know NP compared to BPP or BQP (or even if one is inside the other).

Prototypical examples

- P: evaluate a *function*

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$
  - Think chess: do I have a move, so no matter what you do, I can find a move, so no matter, etc., etc., I end up winning?

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$
  - Think chess: do I have a move, so no matter what you do, I can find a move, so no matter, etc., etc., I end up winning?
- Caveat: requires the puzzle/game to only have a *polynomial number of moves*.

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$
  - Think chess: do I have a move, so no matter what you do, I can find a move, so no matter, etc., etc., I end up winning?
- Caveat: requires the puzzle/game to only have a *polynomial number of moves*.
  - Puzzles/games with exponentially many moves may be harder.

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$
  - Think chess: do I have a move, so no matter what you do, I can find a move, so no matter, etc., etc., I end up winning?
- Caveat: requires the puzzle/game to only have a *polynomial number of moves*.
  - Puzzles/games with exponentially many moves may be harder.
  - Go (Japanese rules): actually EXP-complete to solve a position.

Prototypical examples

- P: evaluate a *function*
  - Given a circuit f and input x, what is $f(x)$?
- NP: solve a *puzzle*
  - SAT: given f, determine if $\exists x : f(x) = 1$?
  - Think candy crush: is there any sequence of moves to achieve score X?
  - Easy to verify once the solution is found.
- PSPACE: solve a *2-player game*
  - TQBF: $\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_n : f(x) = 1$
  - Think chess: do I have a move, so no matter what you do, I can find a move, so no matter, etc., etc., I end up winning?
- Caveat: requires the puzzle/game to only have a *polynomial number of moves*.
  - Puzzles/games with exponentially many moves may be harder.
  - Go (Japanese rules): actually EXP-complete to solve a position.
  - Zelda: actually PSPACE-complete to solve a level.

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.
  - IP $=$ PSPACE [1992]

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.
  - IP = PSPACE [1992]
- The god of go:

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.
  - IP = PSPACE [1992]
- The god of go:
  - Probably *cannot* convince you (if PSPACE $\neq$ EXP)

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.
  - IP = PSPACE [1992]
- The god of go:
  - Probably *cannot* convince you (if PSPACE $\neq$ EXP)
  - But *two* gods of go, in different rooms unable to communicate, can!

Interactive proofs

- You're a lowly P peon, and can't solve NP problems (like candy crush), PSPACE ones (like chess), or EXP ones (like go).
- If a god appears before you, can they convince you of the answer?
  - But you're skeptical—maybe it's actually a devil before you.
- The god of candy crush can tell you the solution, and you can check.
- The god of chess can:
  - tell you one line of play that wins for white?
  - with *interactivity*: convince you he's better than you at chess?
  - Remarkable fact: with interactivity, and careful questioning, can *convince you white wins*.
  - IP = PSPACE [1992]
- The god of go:
  - Probably *cannot* convince you (if PSPACE $\neq$ EXP)
  - But *two* gods of go, in different rooms unable to communicate, can!
  - In fact, MIP=NEXP [1991]

Class Outline

1. Complexity classes
2. Computability

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.
- Consider the following function:

```
function TROUBLE(s)
    if HALTS(s, s) then
        while True do
            pass
    else
        return
```

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.
- Consider the following function:

```
function TROUBLE(s)
    if HALTS(s, s) then
        while True do
            pass
    else
        return
```

- Does $\textsc{Trouble}(\textsc{Trouble})$ halt?

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.
- Consider the following function:

```
function TROUBLE(s)
    if HALTS(s, s) then
        while True do
            pass
    else
        return
```

- Does $\textsc{Trouble}(\textsc{Trouble})$ halt?
  - If it does, it doesn't; if it doesn't, it does.

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.
- Consider the following function:

```
function TROUBLE(s)
    if HALTS(s, s) then
        while True do
            pass
    else
        return
```

- Does $\textsc{Trouble}(\textsc{Trouble})$ halt?
  - If it does, it doesn't; if it doesn't, it does.
- Resolution to paradox: $\textsc{Halts}$ cannot be written down.

Halting Problem

- Given a piece of code, determine if it runs forever or will halt.
- Suppose you had a program $\textsc{Halts}(p, x)$ that determines if the program with code p halts on input x.
- Consider the following function:

```
function TROUBLE(s)
    if HALTS(s, s) then
        while True do
            pass
    else
        return
```

- Does $\textsc{Trouble}(\textsc{Trouble})$ halt?
  - If it does, it doesn't; if it doesn't, it does.
- Resolution to paradox: $\textsc{Halts}$ cannot be written down.
- Implies that $\textsc{Halts}(p)$—with no input x—is also uncomputable.

Halting problem: attempts to solve it

- How about this solution:

```
function HALTS(p)
    Run p for T steps (e.g., 1 hour).
    If it halts, return TRUE
    Otherwise, return FALSE.
```

Halting problem: attempts to solve it

- How about this solution:

```
function HALTS(p)
    Run p for T steps (e.g., 1 hour).
    If it halts, return TRUE
    Otherwise, return FALSE.
```

- Works for short programs!

Halting problem: attempts to solve it

- How about this solution:

function $\textsc{Halts}(p)$
    Run p for $T(|p|)$ steps.
    If it halts, **return** $\textsc{True}$
    Otherwise, **return** $\textsc{False}$.

- Works for short programs!
- T needs to grow with the program size

Halting problem: attempts to solve it

- How about this solution:

```
function HALTS(p)
    Run p for T(|p|) steps.
    If it halts, return TRUE
    Otherwise, return FALSE.
```

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.

Halting problem: attempts to solve it

- How about this solution:

function $\textsc{Halts}(p)$
    Run p for $T(|p|)$ steps.
    If it halts, **return** $\textsc{True}$
    Otherwise, **return** $\textsc{False}$.

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.

Halting problem: attempts to solve it

- How about this solution:

function $\textsc{Halts}(p)$
    Run p for $T(|p|)$ steps.
    If it halts, **return** $\textsc{True}$
    Otherwise, **return** $\textsc{False}$.

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.
- ... but $\textsc{Halts}$ is uncomputable, so $\mathrm{BB}(k)$ is too.

Halting problem: attempts to solve it

- How about this solution:

```
function HALTS(p)
    Run p for T(|p|) steps.
    If it halts, return TRUE
    Otherwise, return FALSE.
```

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.
- ... but HALTS is uncomputable, so $\mathrm{BB}(k)$ is too.
- Still, there exists a program that solves HALTS on any 1Gb program.

Halting problem: attempts to solve it

- How about this solution:

function $\textsc{Halts}(p)$
    Run p for $T(|p|)$ steps.
    If it halts, **return** $\textsc{True}$
    Otherwise, **return** $\textsc{False}$.

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.
- ... but $\textsc{Halts}$ is uncomputable, so $\mathrm{BB}(k)$ is too.
- Still, there exists a program that solves $\textsc{Halts}$ on any 1Gb program.
  - And it's even short!

Halting problem: attempts to solve it

- How about this solution:

function $\textsc{Halts}(p)$
  Run p for $T(|p|)$ steps.
  If it halts, **return** $\textsc{True}$
  Otherwise, **return** $\textsc{False}$.

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.
- ... but $\textsc{Halts}$ is uncomputable, so $\mathrm{BB}(k)$ is too.
- Still, there exists a program that solves $\textsc{Halts}$ on any 1Gb program.
  - And it's even short! Just needs to know the slowest size-k machine.

Halting problem: attempts to solve it

- How about this solution:

function $\text{HALTS}(p)$
  Run p for $T(|p|)$ steps.
  If it halts, **return** TRUE
  Otherwise, **return** FALSE.

- Works for short programs!
- T needs to grow with the program size
- There are a finite number of size-k programs, and one of them takes the longest before halting. This is the *busy beaver* number $\mathrm{BB}(k)$.
- Picking any $T(k) \geq \mathrm{BB}(k)$ would work.
- ... but HALTS is uncomputable, so $\mathrm{BB}(k)$ is too.
- Still, there exists a program that solves HALTS on any 1Gb program.
  - And it's even short! Just needs to know the slowest size-k machine.

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$,

  $\mathrm{BB}(5) = 47176870$ (?),

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$,

  $\mathrm{BB}(5) = 47176870$ (?), $\mathrm{BB}(6) \geq 7.4 \times 10^{36534}$,

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$,

  $\mathrm{BB}(5) = 47176870$ (?), $\mathrm{BB}(6) \geq 7.4 \times 10^{36534}$, $\mathrm{BB}(7) \geq 10^{10^{10^{10^{10^{7}}}}}$.

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$, $\mathrm{BB}(5) = 47176870$ (?), $\mathrm{BB}(6) \geq 7.4 \times 10^{36534}$, $\mathrm{BB}(7) \geq 10^{10^{10^{10^{10^{7}}}}}$.
- To be clear, we have no clue what the *actual* values are.

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$, $\mathrm{BB}(5) = 47176870$ (?), $\mathrm{BB}(6) \geq 7.4 \times 10^{36534}$, $\mathrm{BB}(7) \geq 10^{10^{10^{10^{10^{7}}}}}$.
- To be clear, we have no clue what the *actual* values are.
- Doesn't really reveal the true enormousness of busy beavers! $9^{9^{9^{9}}}$ is big too, but BB is utterly different.

The uncomputability of busy beaver

- $\mathrm{BB}(k) :=$ longest number of steps any k-state Turing machine takes before halting.
- Per Wikipedia: $\mathrm{BB}(2) = 6$, $\mathrm{BB}(3) = 21$, $\mathrm{BB}(4) = 107$,

  $\mathrm{BB}(5) = 47176870$ (?), $\mathrm{BB}(6) \geq 7.4 \times 10^{36534}$, $\mathrm{BB}(7) \geq 10^{10^{10^{10^{10^{7}}}}}$.
- To be clear, we have no clue what the *actual* values are.
- Doesn't really reveal the true enormousness of busy beavers! $9^{9^{9^{9}}}$ is big too, but BB is utterly different.
- $\mathrm{BB}(2000)$ is *impossible to prove an upper bound on*. It's just a number, but you can't prove that the number is correct.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.
- Axioms are *inconsistent* if they can prove a contradiction.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.
- Axioms are *inconsistent* if they can prove a contradiction.

function $\textsc{FindInconsistency}(A)$
    for every possible string s **do**
        if s is a valid proof under A of a contradiction **then**
            return s

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.
- Axioms are *inconsistent* if they can prove a contradiction.

```
function FindInconsistency(A)
    for every possible string s do
        if s is a valid proof under A of a contradiction then
            return s
```

- $\textsc{FindInconsistency}(A)$ halts $\iff$ A is inconsistent.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.
- Axioms are *inconsistent* if they can prove a contradiction.

```
function FindInconsistency(A)
    for every possible string s do
        if s is a valid proof under A of a contradiction then
            return s
```

- $\textsc{FindInconsistency}(A)$ halts $\iff$ A is inconsistent.
- Therefore, if A is consistent, $\textsc{Halts}(\textsc{FindInconsistency}, A)$ cannot be proven under A.

Gödel's Incompleteness Theorem

Theorem (Gödel's second incompleteness theorem)

No consistent system of axioms can prove its own consistency.

- Mathematical proofs are based on a set of axioms
  - Euclidean geometry (two points determine a line, etc.)
  - ZFC: Zermelo-Fraenkel set theory with the axiom of choice is standard.
- Axioms are *inconsistent* if they can prove a contradiction.

```
function FindInconsistency(A)
    for every possible string s do
        if s is a valid proof under A of a contradiction then
            return s
```

- $\textsc{FindInconsistency}(A)$ halts $\iff$ A is inconsistent.
- Therefore, if A is consistent, $\textsc{Halts}(\textsc{FindInconsistency}, A)$ cannot be proven under A.
- Therefore $\mathrm{BB}(|\textsc{FindInconsistency}| + |ZFC|)$ cannot be upper bounded under ZFC.

The uncomputability of busy beaver

- Gödel says: we cannot prove any upper bound on $\mathrm{BB}(|\textsc{FindInconsistency}| + |\mathit{ZFC}|)$ is.

The uncomputability of busy beaver

- Gödel says: we cannot prove any upper bound on $\mathrm{BB}(|\textsc{FindInconsistency}| + |\mathit{ZFC}|)$ is.
  - Concretely: we cannot prove $\mathrm{BB}(2000)$. [O'Rear, Aaronson-Yedidia '16]

The uncomputability of busy beaver

- Gödel says: we cannot prove any upper bound on $\mathrm{BB}(|\textsc{FindInconsistency}| + |\mathit{ZFC}|)$ is.
  - Concretely: we cannot prove $\mathrm{BB}(2000)$. [O'Rear, Aaronson-Yedidia '16]
  - (Probably impossible to prove for much smaller values, too.)

The uncomputability of busy beaver

- Gödel says: we cannot prove any upper bound on $\mathrm{BB}(|\textsc{FindInconsistency}| + |\mathit{ZFC}|)$ is.
  - Concretely: we cannot prove $\mathrm{BB}(2000)$. [O'Rear, Aaronson-Yedidia '16]
  - (Probably impossible to prove for much smaller values, too.)
- Bounding $\mathrm{BB}(744)$ would show the Riemann hypothesis is provable (one way or the other).

Back to interactivity

- Recall: MIP $=$ NEXP:

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*
  - MIP* = RE [Ji-Natarajan-Vidick-Wright-Yuen '20].

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*
  - MIP* = RE [Ji-Natarajan-Vidick-Wright-Yuen '20].
- Note: unlike the halting problem, this is computable

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*
  - MIP* = RE [Ji-Natarajan-Vidick-Wright-Yuen '20].
- Note: unlike the halting problem, this is computable
  - If the program *doesn't* halt, the prover doesn't have to halt either—it just shouldn't give the wrong answer.

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*
  - MIP* = RE [Ji-Natarajan-Vidick-Wright-Yuen '20].
- Note: unlike the halting problem, this is computable
  - If the program *doesn't* halt, the prover doesn't have to halt either—it just shouldn't give the wrong answer.
  - So the prover could just run the program till it halts...

Back to interactivity

- Recall: MIP = NEXP:
  - Two non-interacting provers in separate rooms can convince a P verifier of anything computable in nondeterministic exponential time.
- MIP*: two *quantum entangled* non-interacting provers can convince a P verifier *that a program halts.*
  - MIP* = RE [Ji-Natarajan-Vidick-Wright-Yuen '20].
- Note: unlike the halting problem, this is computable
  - If the program *doesn't* halt, the prover doesn't have to halt either—it just shouldn't give the wrong answer.
  - So the prover could just run the program till it halts...
  - but certainly not in polynomial time!

Summary

$$P \subseteq NP \subseteq PSPACE \subseteq EXP \subseteq NEXP \subseteq EXPSPACE \subseteq \dots$$

$$P \subseteq BPP \subseteq BQP \subseteq PSPACE$$

- Halting problem and busy beaver are *uncomputable*
- Cannot prove BB(2000) in ZFC