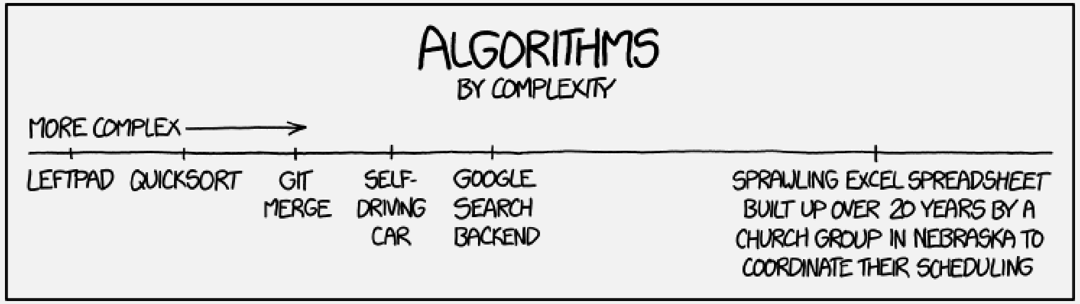
UMB CS 420

Time Complexity

Wednesday, May 1, 2024



Announcements

- HW 10 in
  - ~~Due Wed 5/1 12pm noon~~

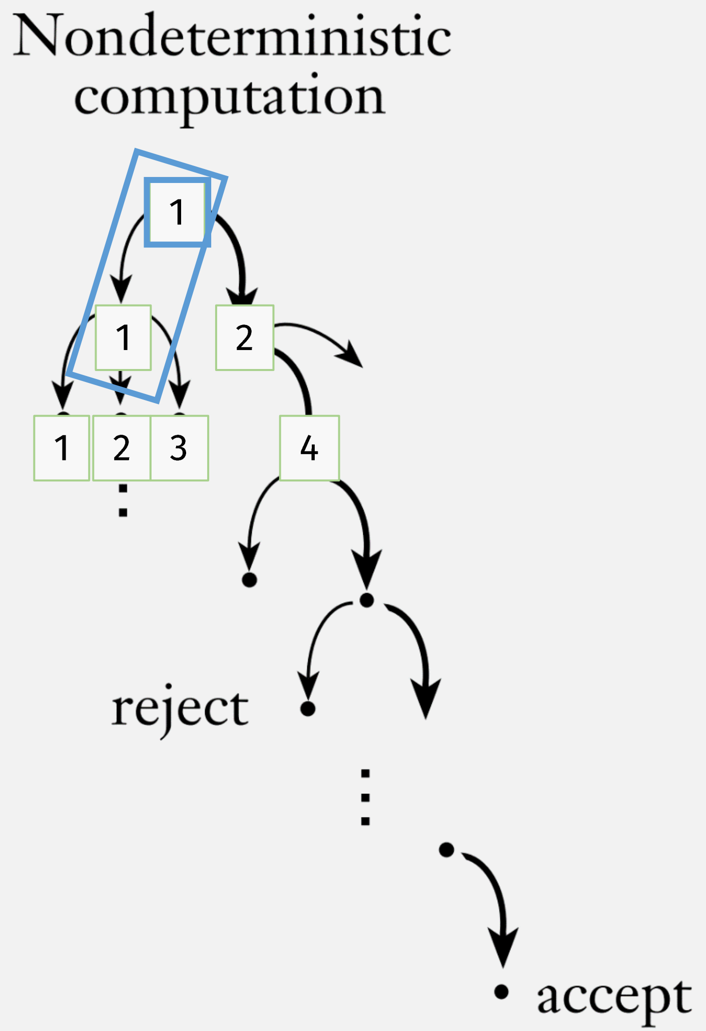
- HW 11 out
  - Due Wed 5/8 12pm noon

<u>Lecture Participation Question 5/1</u>

- What is the worst case number of steps of a deterministic single-tape Turing machine called?

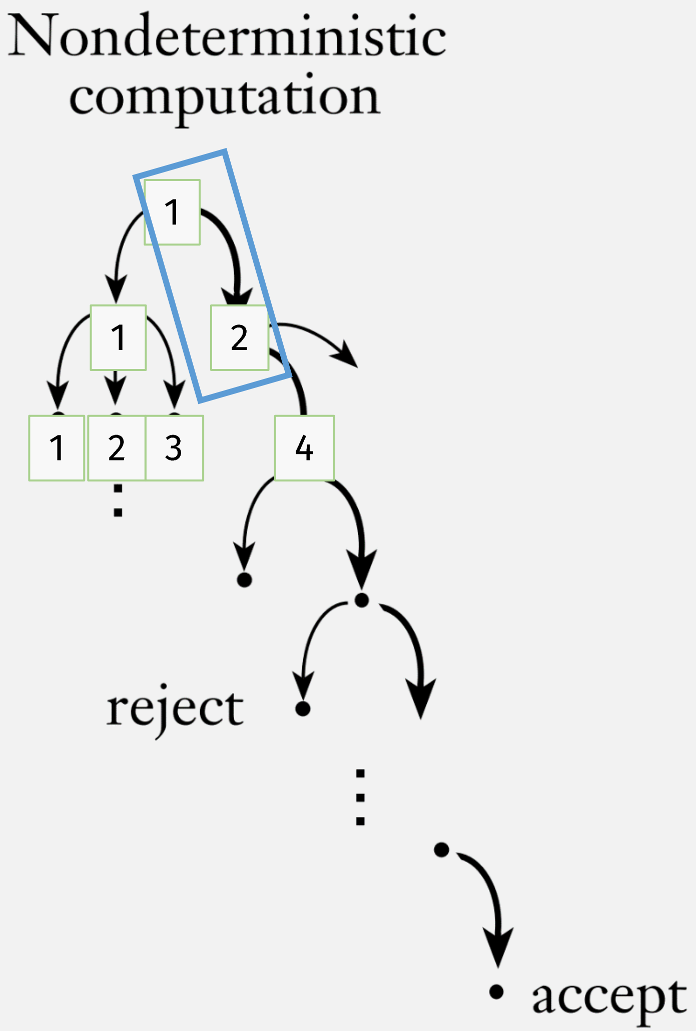
Flashback: Nondet. TM → Deterministic TM

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



Flashback: Nondet. TM → Deterministic TM

- To <u>simulate</u> NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - Root node: 1
    - 1-1
    - **1-2**



Flashback: Nondet. TM → Deterministic TM

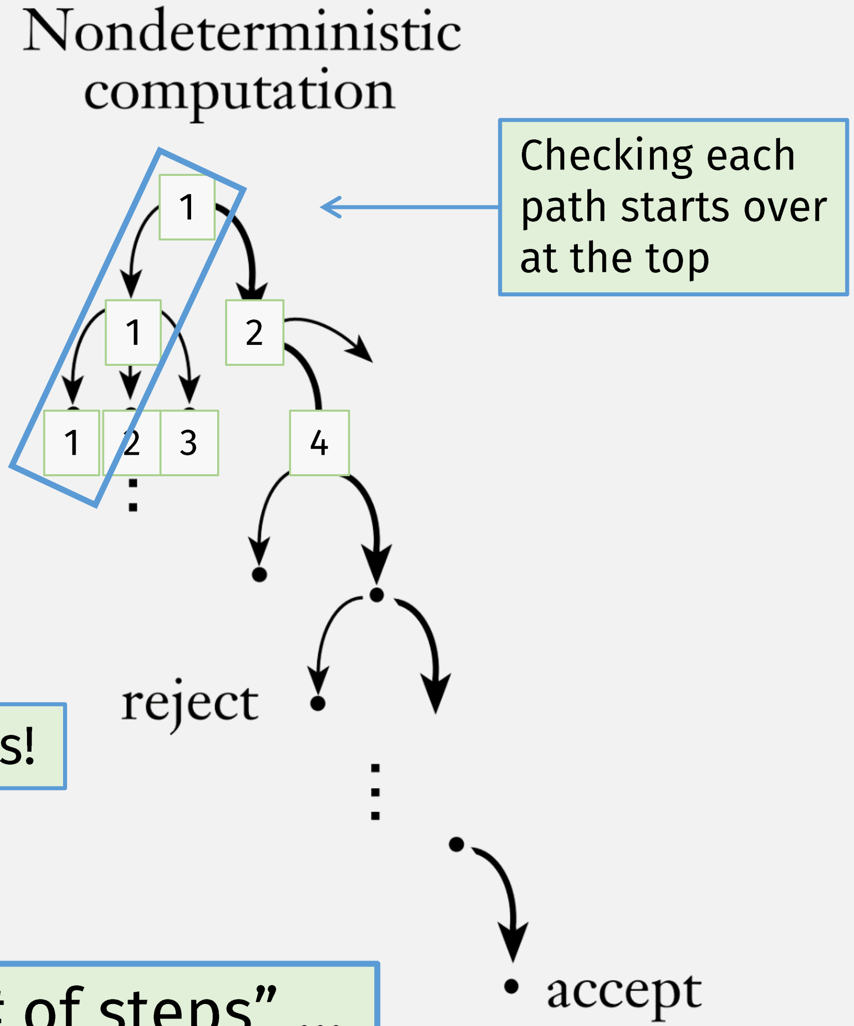
- To simulate NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - Root node: 1
    - 1-1
    - 1-2
    - **1-1-1**

A **TM** and an **NTM** are "equivalent" …

.. but NOT if we care about the # of steps!

So **how inefficient is it?**

First, **we need a formal way to count "# of steps" …**



A Simpler Example: $A = \{0^k 1^k | k \geq 0\}$

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

# of steps (worst case), n = length of w input:

➢ TM Line 1:

- ***n*** steps to scan + ***n*** steps to return to beginning = ***2n* steps**

A Simpler Example: $A = \{0^k 1^k | k \geq 0\}$

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

# of steps (worst case), ***n*** **= length of** ***w*** **input:**

- TM Line 1:
  - ***n*** steps to scan + ***n*** steps to return to beginning = ***2n* steps**

➢ Lines 2-3 (loop):

- **steps/iteration** (line 3): ***n*/2** steps to find "1" + ***n*/2** steps to return = ***n*** steps
- **# iterations** (line 2): Each scan crosses off 2 chars, so at most ***n*/2** scans
- **Total** = steps/iteration * # iterations = ***n*** **(*n*/2)** = **$n^2/2$ steps**

A Simpler Example: $A = \{0^k 1^k | k \geq 0\}$

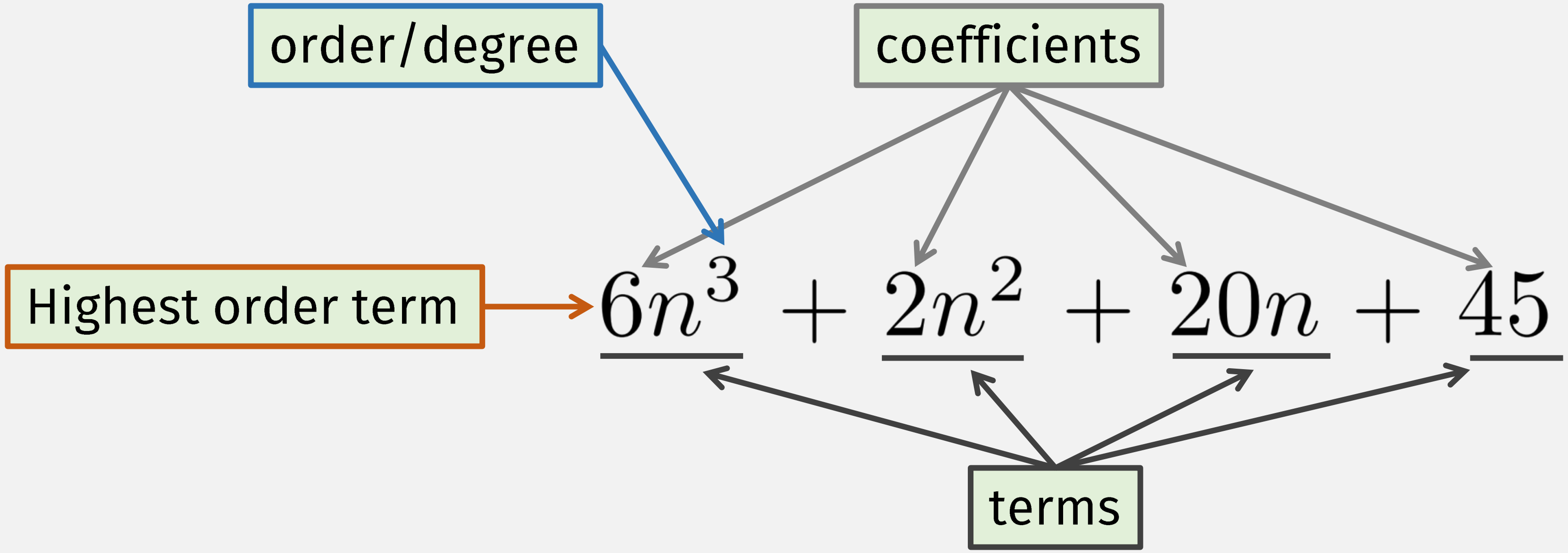
M_1 = "On input string w:
1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

$n^2/2 + 3n$

# of steps (worst case), n = length of w input:

- TM Line 1:
  - n steps to scan + n steps to return to beginning = **$2n$ steps**
- Lines 2-3 (loop):
  - steps/iteration (line 3): $n/2$ steps to find "1" + $n/2$ steps to return = n steps
  - # iterations (line 2): Each scan crosses off 2 chars, so at most $n/2$ scans
  - Total = steps/iteration * # iterations = $n(n/2)$ = **$n^2/2$ steps**
- Line 4:
  - **n steps** to scan input one more time
- Total: $2n + n^2/2 + n$ = **$n^2/2 + 3n$ steps**

Interlude: Polynomials



Definition: Time Complexity

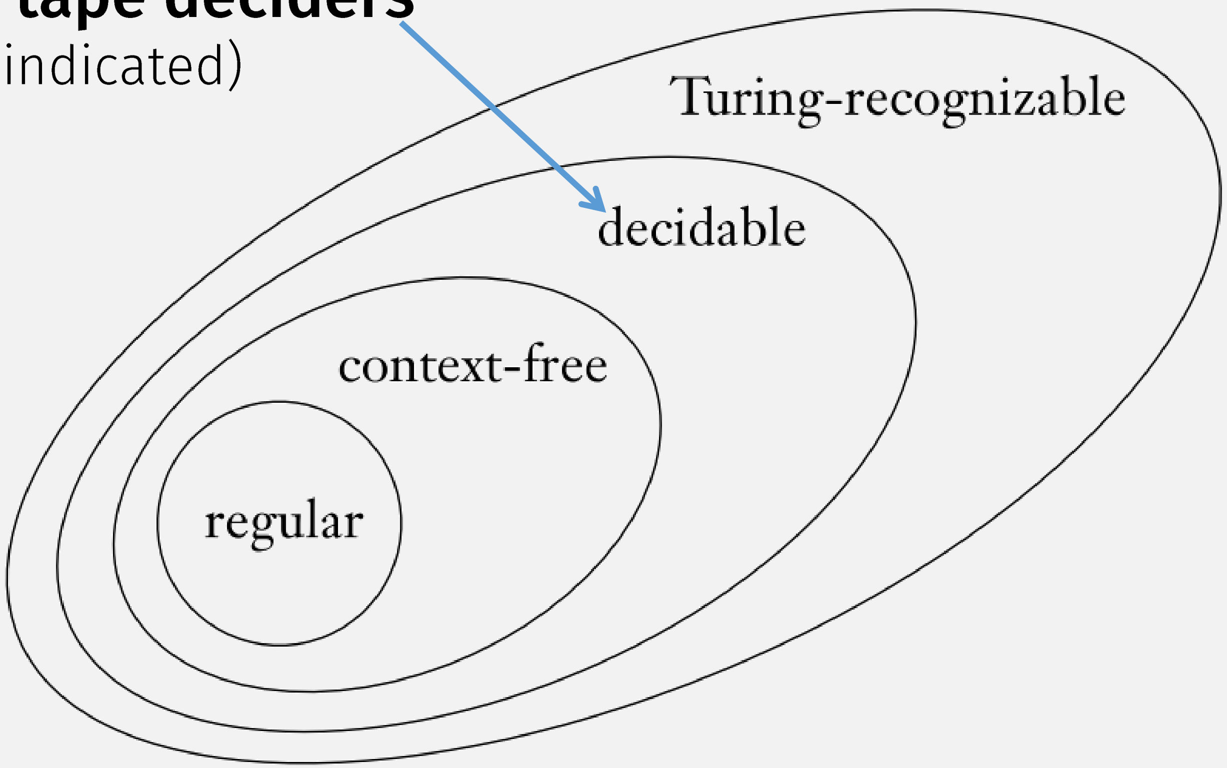
i.e., a decider (algorithm)

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Running Time or **Time Complexity** is a **property of a (Turing) Machine**

Where Are We Now?

We are back in here now:
deterministic, single-tape deciders
(unless otherwise indicated)



Definition: Time Complexity

NOTE: n has no units, it's only *roughly* "length" of the input

We can use **any unit for** n that is correlated with the input length

n could be:
characters,
states,
nodes, …

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Worst case

Depends on size of input

- Machine M_1 that decides $A = \{0^k 1^k | k \geq 0\}$
  - Running time or Time Complexity: $n^2/2+3n$

M_1 = "On input string w:
1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Interlude: Asymptotic Analysis

Total: $\boldsymbol{n^2 + 3n}$

- If $\boldsymbol{n} = 1$
  - $\boldsymbol{n^2} = 1$
  - $\boldsymbol{3n} = 3$
  - Total = 4
- If $\boldsymbol{n} = 10$
  - $\boldsymbol{n^2} = 100$
  - $\boldsymbol{3n} = 30$
  - Total = 130
- If $\boldsymbol{n} = 100$
  - $\boldsymbol{n^2} = 10{,}000$
  - $\boldsymbol{3n} = 300$
  - Total = 10,300
- If $\boldsymbol{n} = 1{,}000$
  - $\boldsymbol{n^2} = 1{,}000{,}000$
  - $\boldsymbol{3n} = 3{,}000$
  - Total = 1,003,000

$\boldsymbol{n^2 + 3n} \approx \boldsymbol{n^2}$ as n gets large

asymptotic analysis only cares about large n

Definition: Big-*O* Notation

notation

Let f and g be functions $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

"only care about <u>large</u> n"

When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

In other words: **Keep only highest order term, drop all coefficients**

- **Machine M_1 that decides** $A = \{0^k 1^k \mid k \geq 0\}$
  - is an $\boldsymbol{n^2 + 3n}$ **time Turing machine**
  - is an $O(\boldsymbol{n^2})$ **time Turing machine**, i.e., $\boldsymbol{n^2 + 3n} = O(\boldsymbol{n^2})$
  - has **asymptotic upper bound** $O(\boldsymbol{n^2})$

Definition: Small-o Notation (less used)

Let f and g be functions $f, g \colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

Analogy: Big-O : ≤ :: small-o : <

Let f and g be functions $f, g \colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\,g(n).$$

When $f(n) = O(g(n))$, we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Other "Oh"s (not used in this course)

- "Big Theta" Θ
- "Small Omega" ω
- "Big Omega" Ω

Don't use these by mistake!

Pay attention to our <u>exact</u> definitions!

Big-O arithmetic

- $O(\boldsymbol{n}^2) + O(\boldsymbol{n}^2)$
  $= O(\boldsymbol{n}^2)$

- $O(\boldsymbol{n}^2) + O(\boldsymbol{n})$
  $= O(\boldsymbol{n}^2)$

- $\mathbf{2}\boldsymbol{n} = O(\boldsymbol{n})$?
  - TRUE

NOTE: <u>Other courses</u> might use Big-Θ notation (which is a tighter bound) where some of these equalities won't be true, e.g., $\mathbf{2}\boldsymbol{n} \neq \Theta(\boldsymbol{n}^2)$

- $\mathbf{2}\boldsymbol{n} = O(\boldsymbol{n}^2)$?
  - TRUE

NOTE: <u>In this course</u>, we use Big-O only, <u>not</u> Big-Θ (so do not confuse the two)

- $\mathbf{1} = O(\boldsymbol{n}^2)$?
  - TRUE

- $\mathbf{2}^{\boldsymbol{n}} = O(\boldsymbol{n}^2)$?
  - FALSE

Definition: Time Complexity Classes

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

Remember: **TMs:** have a **time complexity** (i.e., a running time); **languages:** are in a **time complexity class**

complexity class of a language is determined by the **time complexity** (running time) of its decider TM

A **language** could be in more than one **time complexity class**

- Machine M_1 decides language $A = \{0^k 1^k | k \geq 0\}$
  - M_1 has time complexity (running time) of $O(n^2)$
  - A is in time complexity class $\mathbf{TIME}(n^2)$

A Faster Machine? $A = \{0^k 1^k \mid k \geq 0\}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Previously:

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

A Faster Machine? $A = \{0^k 1^k \mid k \geq 0\}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Number of steps (worst case), ***n*** **= length of input:**

➢ Line 1:
- ***n*** steps to scan + ***n*** steps to return to beginning = $O(n)$ steps

A Faster Machine? $A = \{0^k 1^k \mid k \geq 0\}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Number of steps (worst case), ***n*** **= length of input:**

- Line 1:
  - ***n*** steps to scan + ***n*** steps to return to beginning = $O(n)$ steps

➢ Lines 2-4 (loop):

  - **steps/iteration** (lines 3-4): a scan takes $O(n)$ steps
  - **# iters** (line 2): Each iter crosses off *half* the chars, so at most $O(\log n)$ scans
  - **Total**: $O(n) * O(\log n)$ = $O(n \log n)$ steps

Interlude: Logarithms (dual to exponentiation)

- If $2^x = y$ …
- … then $\log_2 y = x$
- $\log_2 n = O(\mathbf{log}\ \boldsymbol{n})$
  - "divide and conquer" algorithms = $O(\mathbf{log}\ \boldsymbol{n})$
  - E.g., binary search
- (In computer science, **base-2 is the only base!** So 2 is dropped)

A Faster Machine? $A = \{0^k 1^k | k \geq 0\}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Number of steps (worst case), ***n*** **= length of input:**

- Line 1:
  - ***n*** steps to scan + ***n*** steps to return to beginning = $O(n)$ steps
- Lines 2-4 (loop):
  - steps/iteration (lines 3-4): a scan takes $O(n)$ steps
  - # iters (line 2): Each iter crosses off *half* the chars, so at most $O(\log n)$ scans
  - Total: $O(n) * O(\log n) = O(n \log n)$ steps
- ➢ Line 5:
  - $O(n)$ steps to scan input one more time

A Faster Machine? $A = \{0^k 1^k \mid k \geq 0\}$

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

$O(n \log n)$

Prev: $n^2/2 + 3n = O(n^2)$

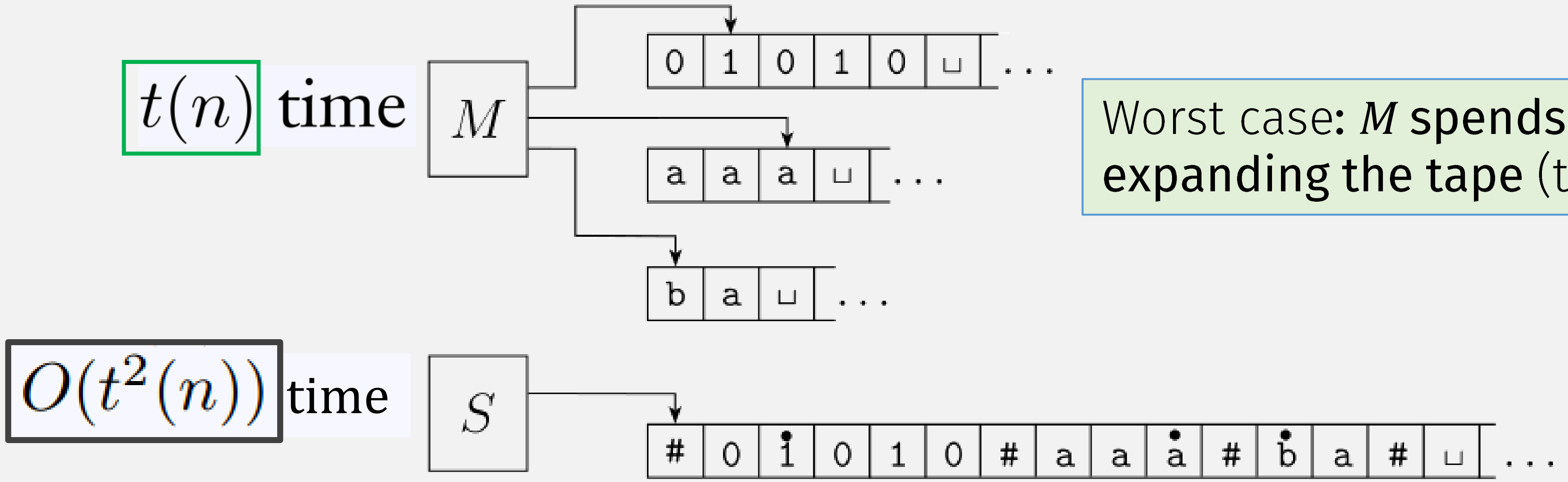
Number of steps (worst case), n = **length of input:**

- Line 1:
  - n steps to scan + n steps to return to beginning = $O(n)$ steps
- Lines 2-4 (loop):
  - steps/iteration (lines 3-4): a scan takes $O(n)$ steps
  - # iters (line 2): Each iter crosses off *half* the chars, so at most $O(\log n)$ scans
  - Total: $O(n) * O(\log n)$ = $O(n \log n)$ steps
- Line 5:
  - $O(n)$ steps to scan input one more time
- Total: $O(n) + O(n \log n) + O(n)$ =

Terminology: Categories of Bounds

- Exponential time
  - $O(2^{n \wedge c})$, for $c > 0$, or $2^{O(n)}$ (always base 2)
- Polynomial time
  - $O(n^c)$, for $c > 0$
- Quadratic time (special case of polynomial time)
  - $O(n^2)$
- Linear time (special case of polynomial time)
  - $O(n)$
- Log time
  - $O(\log n)$

Multi-tape vs Single-tape TMs: # of Steps



Worst case: M **spends all** $t(n)$ **steps expanding the tape** (to the right)

- For single-tape TM to <u>simulate 1 step</u> of multi-tape: $t(n)$
  1. Scan to find all "heads" = O(length of <u>all</u> M's tapes)
  2. "Execute" transition at all the heads = O(length of <u>all</u> M's tapes)
- # single-tape steps to simulate 1 multitape step (worst case)
  - = O(length of all M's tapes)
  - = $O(t(n))$, If M spends all its steps expanding its tapes
- <u>Total steps (single tape)</u>: $O(t(n))$ per step × $t(n)$ steps =

Flashback: Nondet. TM → Deterministic TM

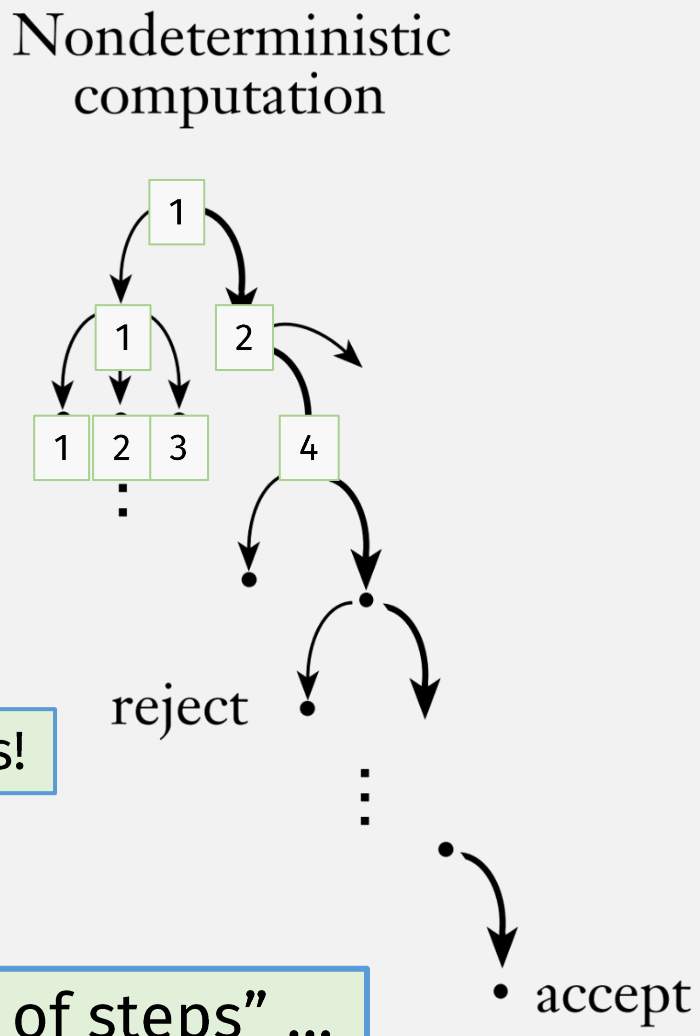
- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - 1
    - 1-1
    - 1-2
    - **1-1-1**

A **TM** and an **NTM** are "equivalent" ...

.. but NOT if we care about the # of steps!

So how inefficient is it?

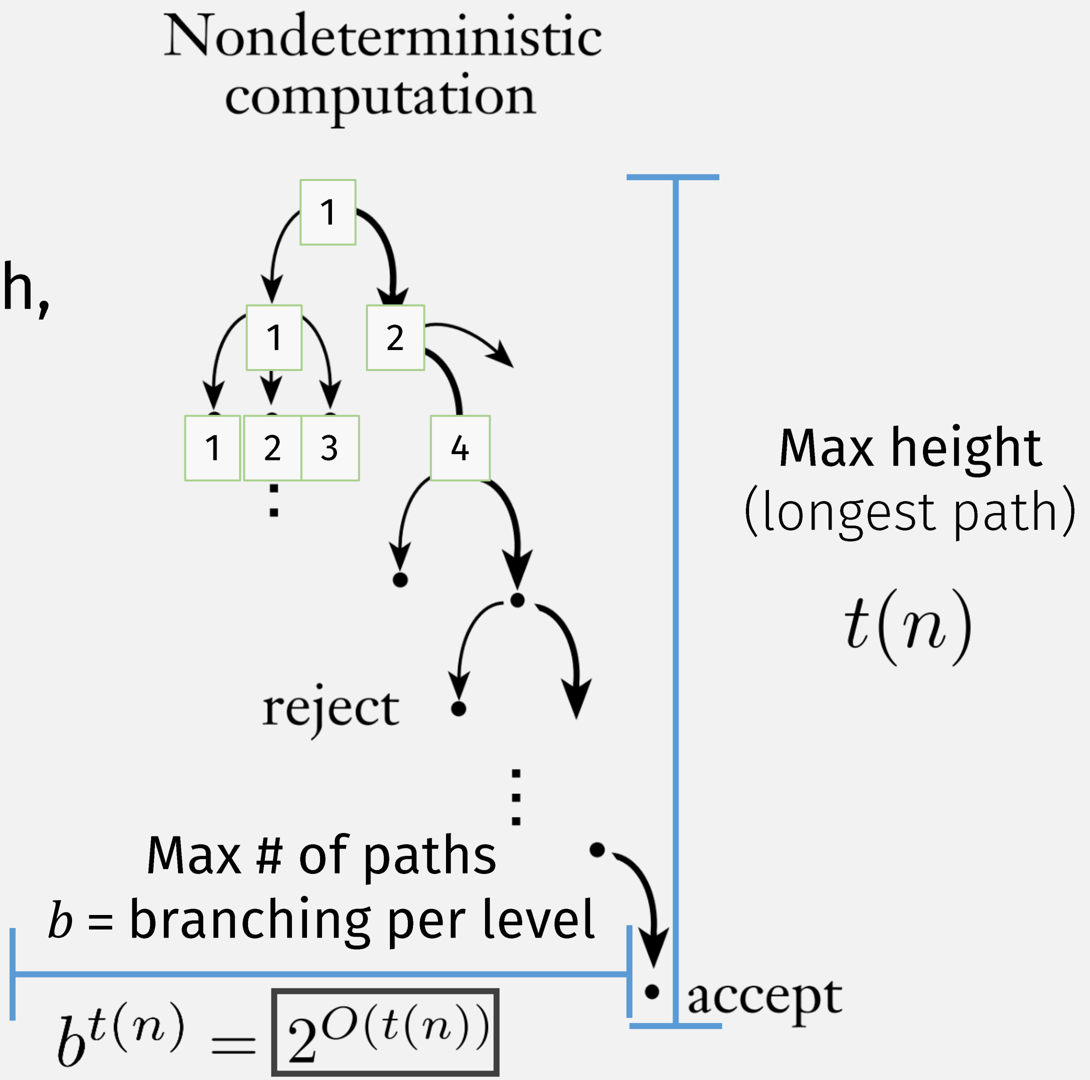
First, we need a formal way to count "# of steps" ...



Flashback: Nondet. TM → Deterministic TM

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



Summary: TM Variations

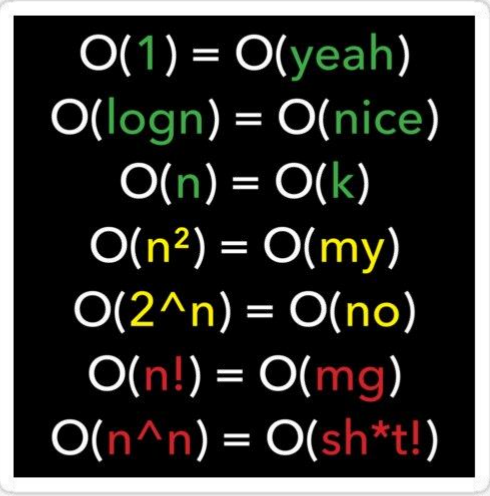
- If **multi-tape TM:** $t(n)$ time
- Then **equivalent single-tape TM:** $O(t^2(n))$
  - **Quadratically** slower

- If **non-deterministic TM:** $t(n)$ time
- Then **equivalent single-tape TM:** $2^{O(t(n))}$
  - **Exponentially** slower

Lecture Participation Question 5/1

On gradescope

Polynomial Time (P)



The Polynomial Time Complexity Class (**P**)

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

- Corresponds to "realistically" solvable problems:
  - Problems in **P**
    - = "**solvable**" or "**tractable**"
  - Problems outside **P**
    - = "**unsolvable**" or "**intractable**"

“Unsolvable” Problems

Mathematicians are weird.

- Unsolvable problems (those outside **P**):
  - usually only have “brute force” solutions
  - i.e., “try all possible inputs”
  - “unsolvable” applies only to large ***n***

Brute-force attack

From Wikipedia, the free encyclopedia

In cryptography, a **brute-force attack** consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing a combination correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found. Alternatively, the attacker can attempt to guess the key which is typically created from the password using a key derivation function. This is known as an **exhaustive key search**.

Amount of Time to Crack Passwords

“abcdefg” 7 characters	.29 milliseconds
“abcdefgh” 8 characters	5 hours
“abcdefghi” 9 characters	5 days
“abcdefghij” 10 characters	4 months
“abcdefghijk” 11 characters	1 decade
“abcdefghijkl” 12 characters	2 centuries

In this class, we’re interested in questions like:

today → How to prove something is “**solvable**” (in **P**)?

How to prove something is “**unsolvable**” (not in **P**)?

3 Problems in **P**

- A Graph Problem:

  $PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

- A Number Problem:

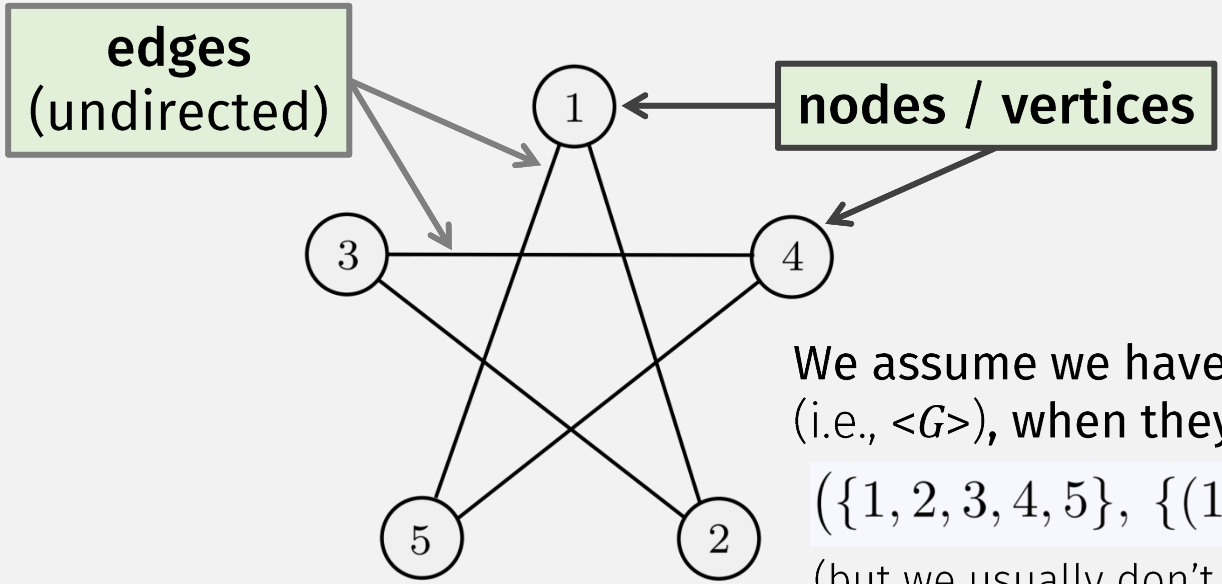
  $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

- A CFL Problem:

  Every context-free language is a member of P

- To prove that a language is "solvable", i.e., in **P** …
  - … construct a polynomial time algorithm deciding the language
- (These may also have nonpolynomial, i.e., brute force, algorithms)
  - Check all possible … paths/numbers/strings …

Interlude: Graphs (see Sipser Chapter 0)



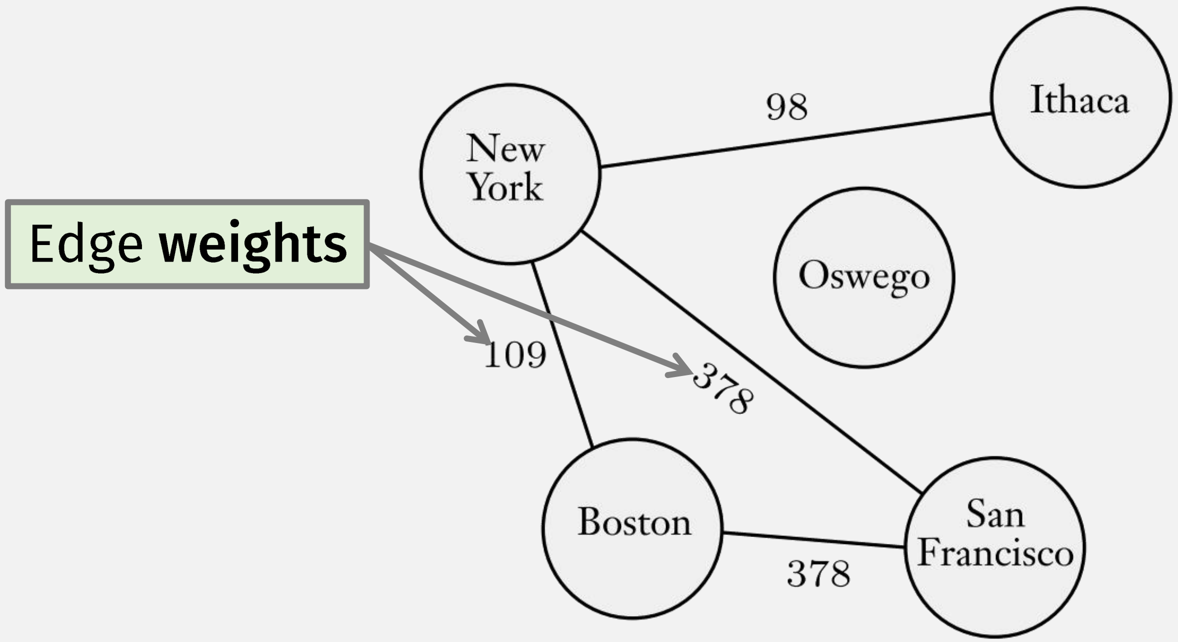
We assume we have *some* string encoding of a graph (i.e., <*G*>), when they are args to TMs, e.g.:

$$(\{1,2,3,4,5\},\ \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,1)\})$$

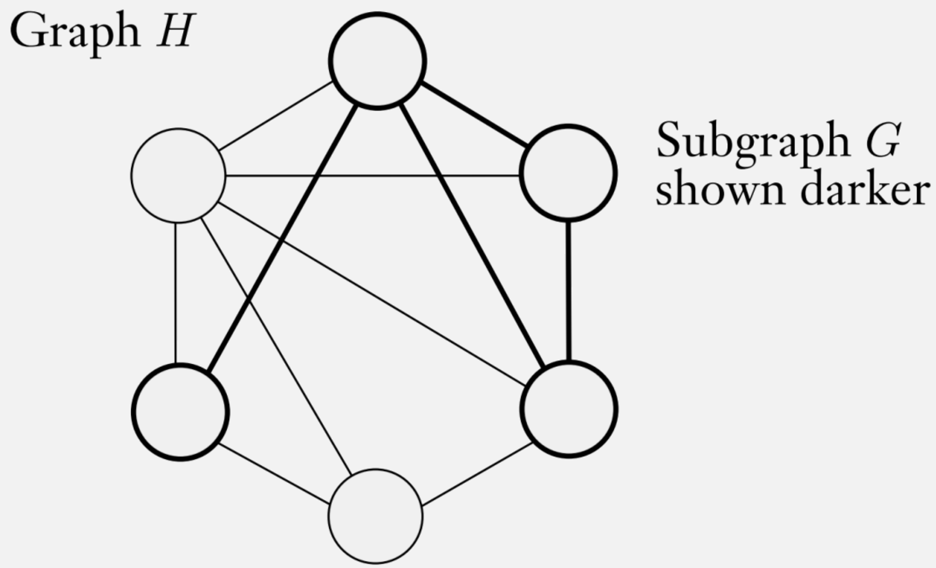
(but we usually don't care about the actual details)

- **Edge** defined by two **nodes** (order doesn't matter)
- Formally, a **graph** = a pair (V, E)
  - Where V = a set of nodes, E = a set of edges

Interlude: Weighted Graphs



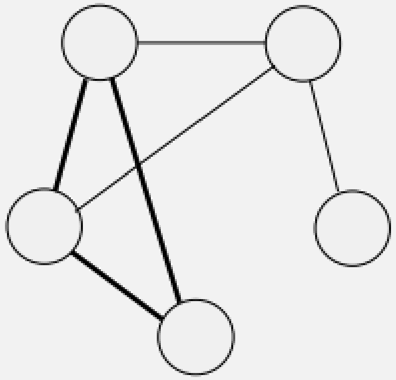
Interlude: Subgraphs

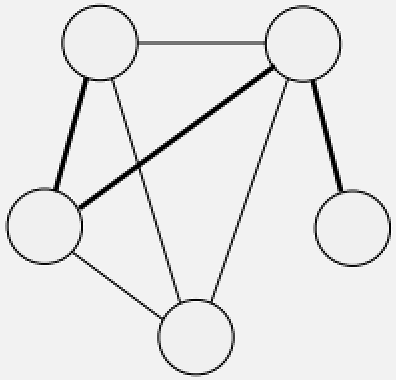


Interlude: Paths and other Graph Things

- **Path**
  - A sequence of nodes connected by edges

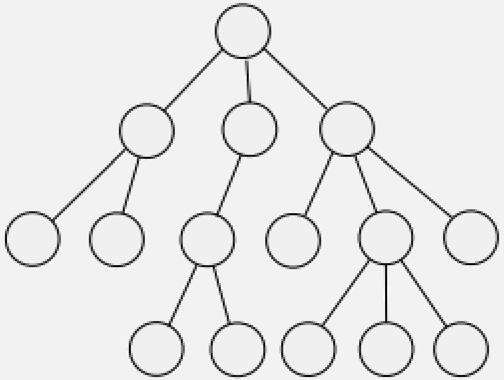
- **Cycle**
  - A path that starts/ends at the same node



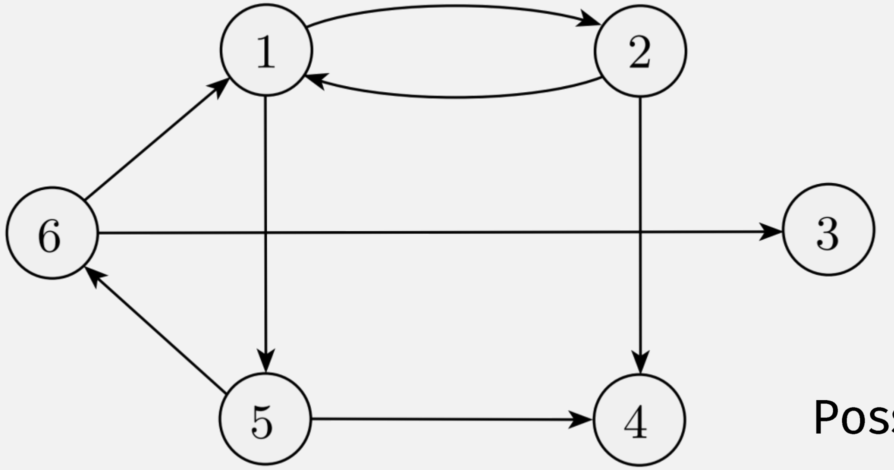


- **Connected graph**
  - Every two nodes has a path

- **Tree**
  - A connected graph with no cycles



Interlude: Directed Graphs



Possible string encoding given to TMs:

$$(\{1,2,3,4,5,6\},\ \{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\})$$

Each pair of nodes included twice

- **Directed graph** = (V, E)
  - V = set of nodes, E = set of edges
- An **edge** is a pair of nodes (u,v), ***order now matters***
  - u = "from" node, v = "to" node
- "degree" of a node: number of edges connected to the node
  - Nodes in a directed graph have both indegree and outdegree

Interlude: Graph Encodings

$(\{1,2,3,4,5\},\ \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,1)\})$

- For graph algorithms, "length of input" n usually = # of vertices
  - (Not number of chars in the encoding)
- So given graph $G = (V, E)$, $n = |V|$
- Max edges?
  - $= O(|V|^2) = O(n^2)$
- So if a set of graphs (call it lang L) is decided by a TM where
  - # steps of the TM = polynomial in the # of vertices

    Or polynomial in the # of edges
  - Then L is in **P**

3 Problems in $\mathbf{P}$

- A Graph Problem:

  $PATH = \{\langle G, s, t\rangle |\ G$ is a directed graph that has a directed path from s to $t\}$

- A Number Problem:

  $RELPRIME = \{\langle x, y\rangle |\ x$ and y are relatively prime$\}$

- A CFL Problem:

  Every context-free language is a member of P

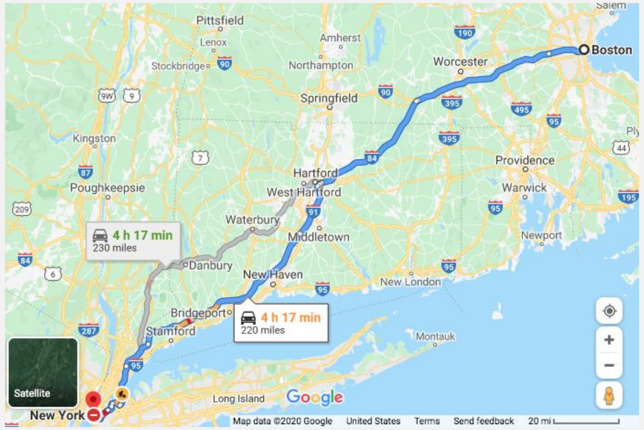
A Graph Theorem: $\mathit{PATH} \in \mathrm{P}$

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

$$\mathit{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

(A **path** is a sequence of nodes connected by edges)



- To prove that a language is in **P** ...
- ... we must construct a polynomial time algorithm deciding the lang
- A non-polynomial (i.e., "brute force") algorithm:
  - check all possible paths,
  - see if any connect s to t
  - If n = # vertices, then # paths $\approx n^n$

A Graph Theorem: $PATH \in \text{P}$

$PATH = \{\langle G, s, t\rangle |\ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

# of steps (worst case) **(*n* = # nodes):**

- Line 1: **1 step**

A Graph Theorem: $PATH \in \mathrm{P}$

$PATH = \{\langle G, s, t \rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

M = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.

2. Repeat the following until no additional nodes are marked:

3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.

4. If t is marked, *accept*. Otherwise, *reject*."

# of steps (worst case) **(*n* = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$

A Graph Theorem: $PATH \in \mathrm{P}$

$PATH = \{\langle G, s, t\rangle |\ G$ is a directed graph that has a directed path from s to $t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.

2. Repeat the following until no additional nodes are marked:

3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.

4. If t is marked, *accept*. Otherwise, *reject*."

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$
  - # iterations (line 2): loop runs at most n times

A Graph Theorem: $PATH \in \mathrm{P}$

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.

2. Repeat the following until no additional nodes are marked:

3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.

4. If t is marked, *accept*. Otherwise, *reject*."

(Breadth-first search)

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$
  - # iterations (line 2): loop runs at most n times
  - ➢ Total: $O(n^3)$

A Graph Theorem: $PATH \in \mathrm{P}$

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - **Steps/iteration** (line 3): max # steps = max # edges = $O(n^2)$
  - **# iterations** (line 2): loop runs at most n times
  - Total: $O(n^3)$

➢ Line 4: **1 step**

A Graph Theorem: $PATH \in \mathrm{P}$

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

$PATH \in \mathrm{TIME}(\boldsymbol{n}^3)$

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

$M =$ "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

$O(\boldsymbol{n}^3)$

# of steps (worst case) **($\boldsymbol{n}$ = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(\boldsymbol{n}^2)$
  - # iterations (line 2): loop runs at most $\boldsymbol{n}$ times
  - Total: $O(\boldsymbol{n}^3)$
- Line 4: **1 step**

➢ Total = 1 + 1 + $O(\boldsymbol{n}^3)$ = $O(\boldsymbol{n}^3)$