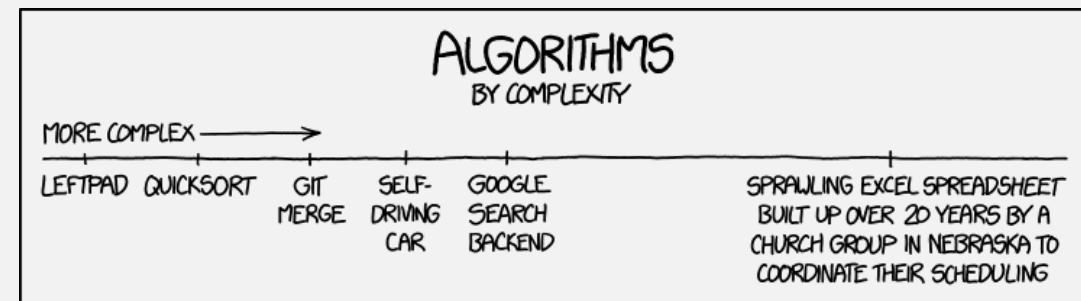


CS 420 / CS 620

Time Complexity

Wednesday, November 26, 2025
UMass Boston Computer Science

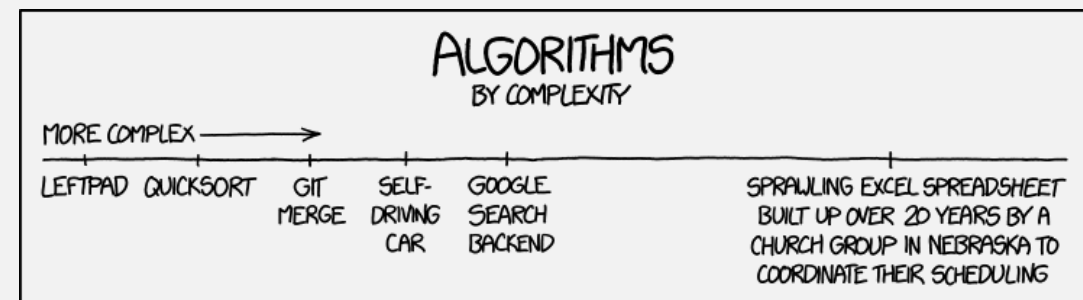


Announcements

- HW 12
 - Out: Mon 11/24 12pm (noon)
 - **Thanksgiving: 11/26-11/30**
 - Due: Fri 12/5 12pm (noon)

Last HW

- HW 13
 - Out: Fri 12/5 12pm (noon)
 - Due: Fri 12/12 12pm (noon) (classes end)
 - Late due: Mon 12/15 12pm (noon) (exams start)
 - Nothing accepted after this (please don't ask)



Class participation question (in Gradescope)

Q1 What is the worst case number of steps of a deterministic single-tape Turing machine called?

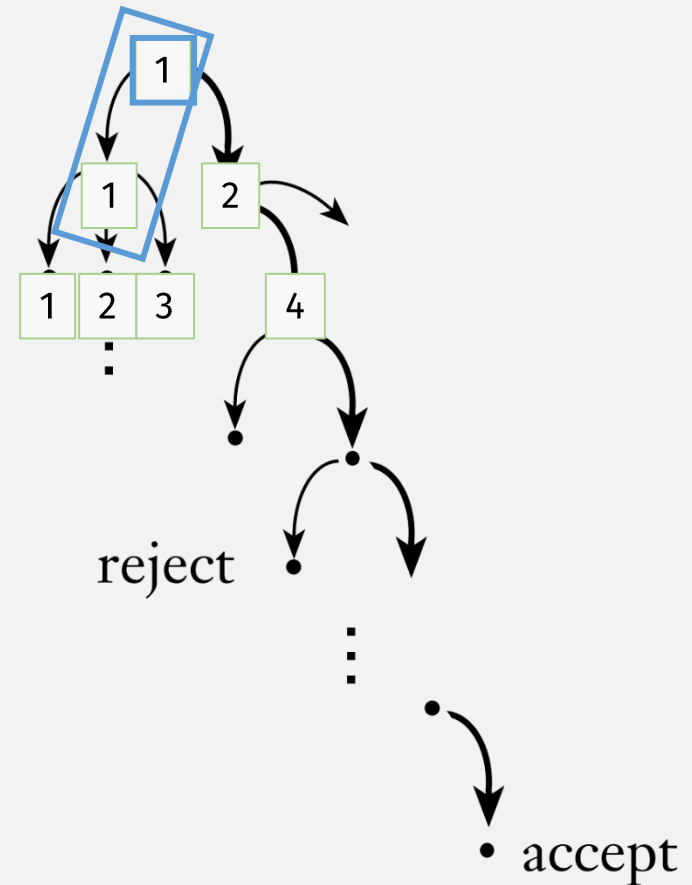
1 Point

(select the one best answer)

Flashback: Nondet. TM \rightarrow 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

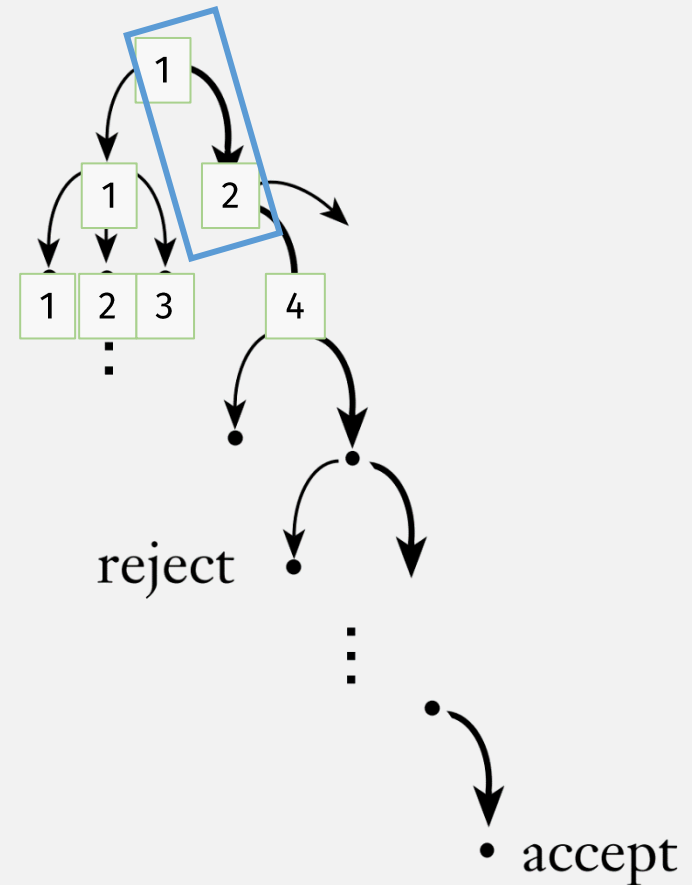
Nondeterministic computation



Flashback: Nondet. TM \rightarrow 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

Nondeterministic computation



Flashback: Nondet. TM \rightarrow 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

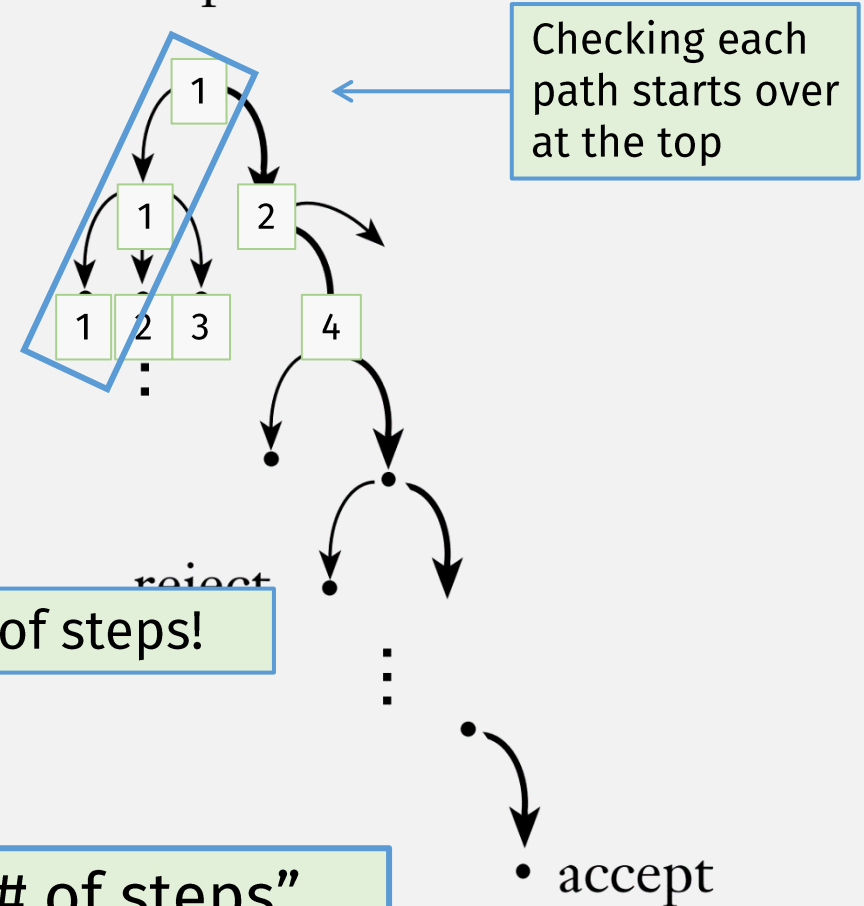
A TM and an NTM are “equivalent” ...

.. but NOT if we take into account the # of steps!

So how inefficient is it?

First, we need: a formal way to count “# of steps” ...

Nondeterministic
computation



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

$M_1 =$ “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: $A = \{0^k 1^k \mid k \geq 0\}$

$M_1 =$ “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 steps to find “1” + n steps to return = $2n$ steps
- # iterations (line 2): Each scan crosses off 2 chars, so at most $n/2$ scans
- Total = steps/iteration * # iterations = $2n (n/2) =$ n^2 steps

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

$M_1 =$ “On input string w :

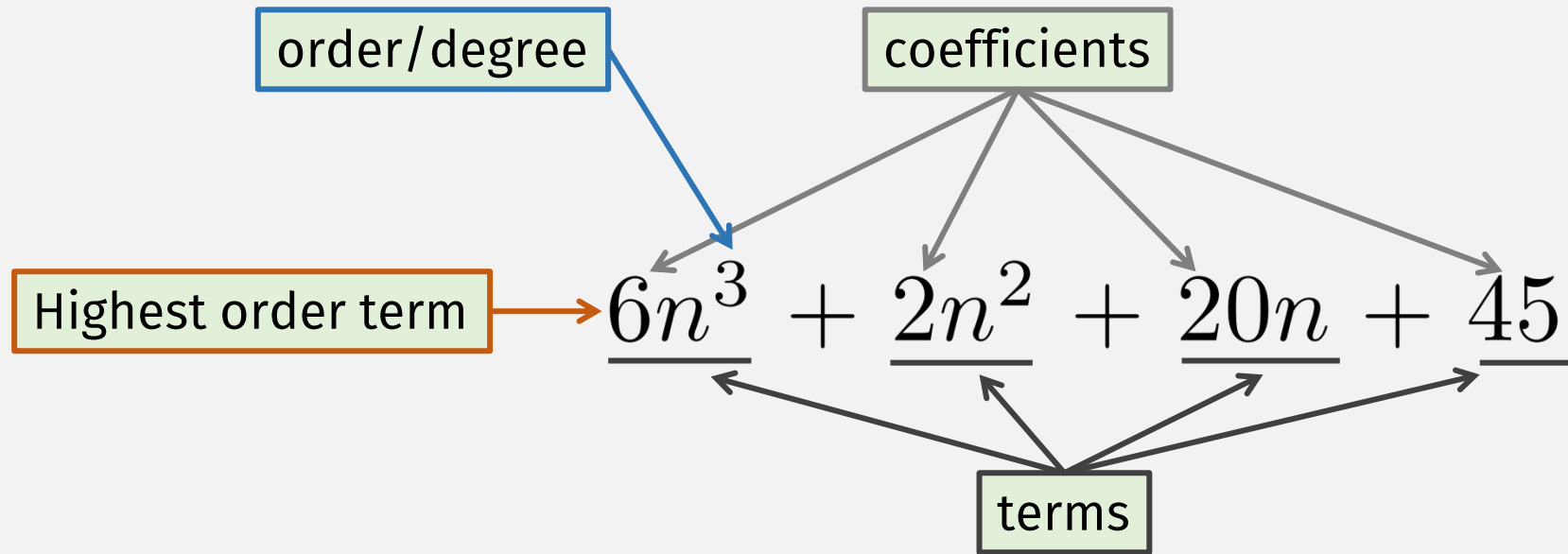
1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed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

$$n^2 + 3n$$

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

Interlude: Polynomials



Definition: Time Complexity

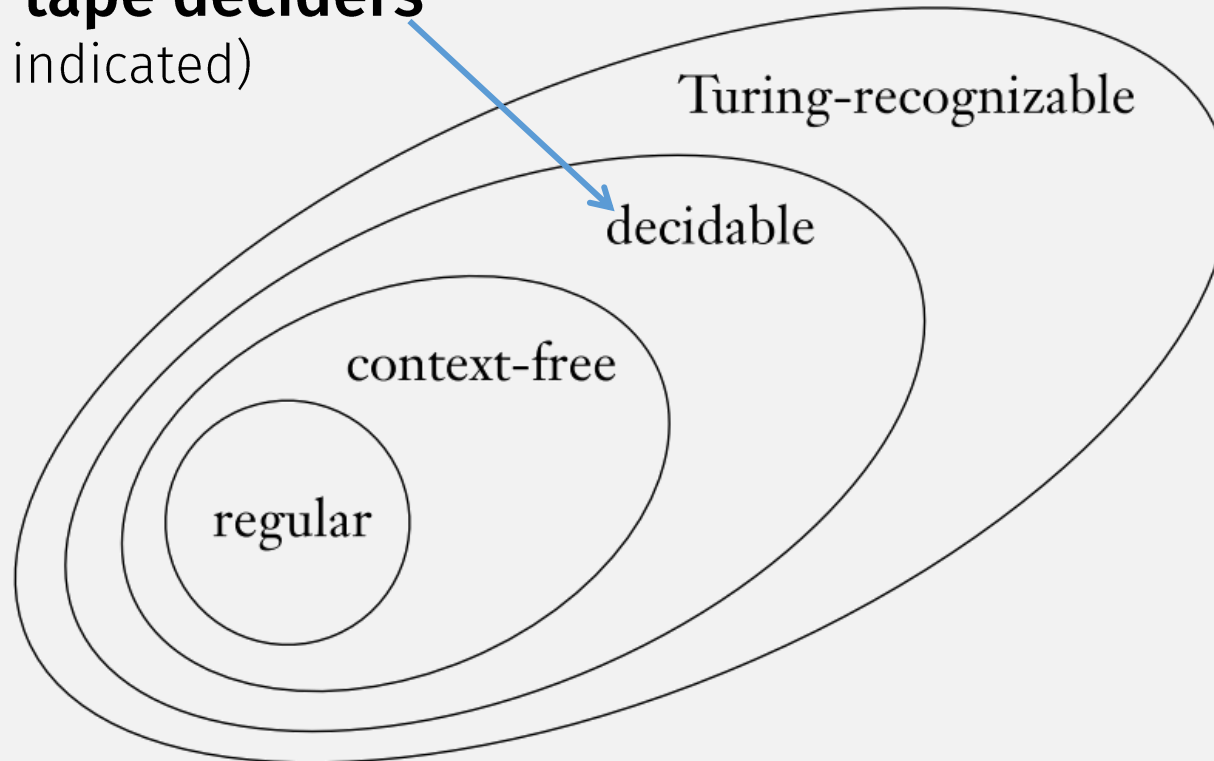
i.e., a decider (algorithm)

Let M be a deterministic Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $f: \mathcal{N} \rightarrow \mathcal{N}$, 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 "constant" size

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 $f: \mathcal{N} \rightarrow \mathcal{N}$, 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.

Depends on size of input

Worst case

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

$M_1 =$ "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: $n^2 + 3n$

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

$n^2 + 3n \approx n^2$ as n gets large

asymptotic analysis only cares about large n

Definition: Big- O Notation

Let f and g be functions $f, g: \mathcal{N} \rightarrow \mathcal{R}^+$. Say that $f(n) = O(g(n))$ 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.

notation

“only care about large n ”

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 $n^2 + 3n$ time Turing machine
 - is an $O(n^2)$ time Turing machine, i.e., $n^2 + 3n = O(n^2)$
 - has asymptotic upper bound $O(n^2)$

Definition: Small- o Notation (less used)

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

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

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 : \leq :: small- o : $<$

Let f and g be functions $f, g: \mathcal{N} \rightarrow \mathcal{R}^+$. Say that $f(n) = O(g(n))$ 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 exact definitions!

Big- O arithmetic

- $O(n^2) + O(n^2)$
= $O(n^2)$

- $O(n^2) + O(n)$
= $O(n^2)$

- $2n = O(n)$?
 - TRUE

- $2n = O(n^2)$?
 - TRUE

- $1 = O(n^2)$?
 - TRUE

- $2^n = O(n^2)$?
 - FALSE

NOTE: Other courses might use Big- θ notation (which is a tighter bound) where some of these equalities won't be true, e.g., $2n \neq \theta(n^2)$

NOTE: In this course, we use Big- O only, not Big- θ (so do not confuse the two)

Definition: Time Complexity Classes

Let $t: \mathcal{N} \rightarrow \mathcal{R}^+$ 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

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

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

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

Previously:

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*.”

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 (“opposite” of exponentiation)

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

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

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: $O(n) * O(\log n) = \underline{O(n \log n)}$ 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: $n^2 + 3n = O(n^2)$

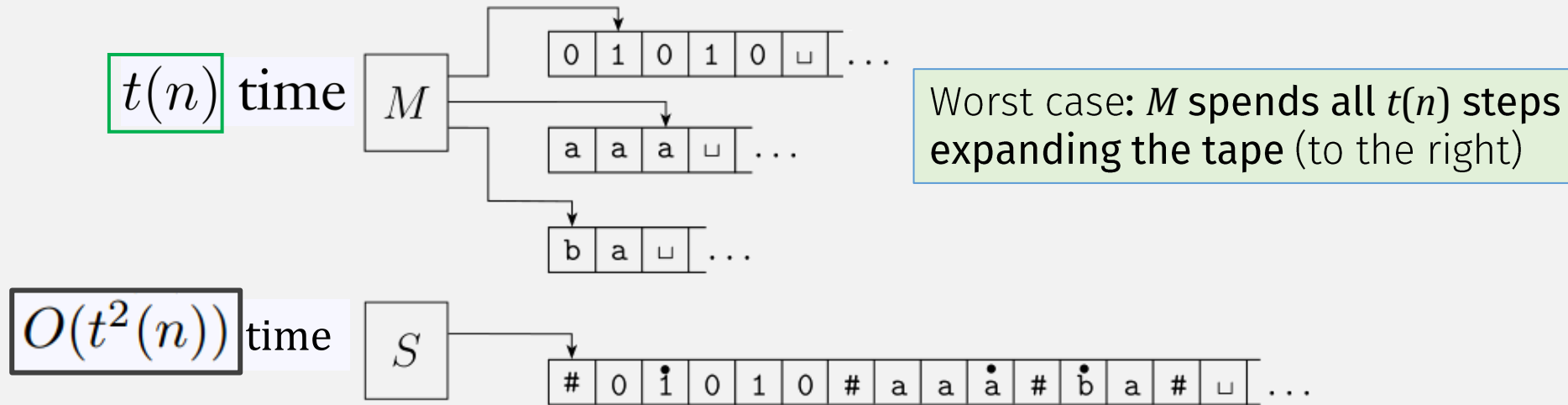
Number of steps (worst case), n = length of input:

- 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**
 - $O(2^{n^c})$, 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**
 - $O(\log n)$
- **Constant time**
 - $O(1)$

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



- For single-tape TM to simulate 1 step of multi-tape:
 1. Scan to find all “heads” = $O(\text{length of all } M\text{'s tapes})$
 2. “Execute” transition at all the heads = $O(\text{length of all } M\text{'s tapes})$
- # single-tape steps to simulate 1 multitape step (worst case)
 - = $O(\text{length of all } M\text{'s tapes})$
 - = $O(t(n))$, If M spends all its steps expanding its tapes
- Total steps (single tape): $O(t(n))$ per step $\times t(n)$ steps =

Flashback: Nondet. TM \rightarrow 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

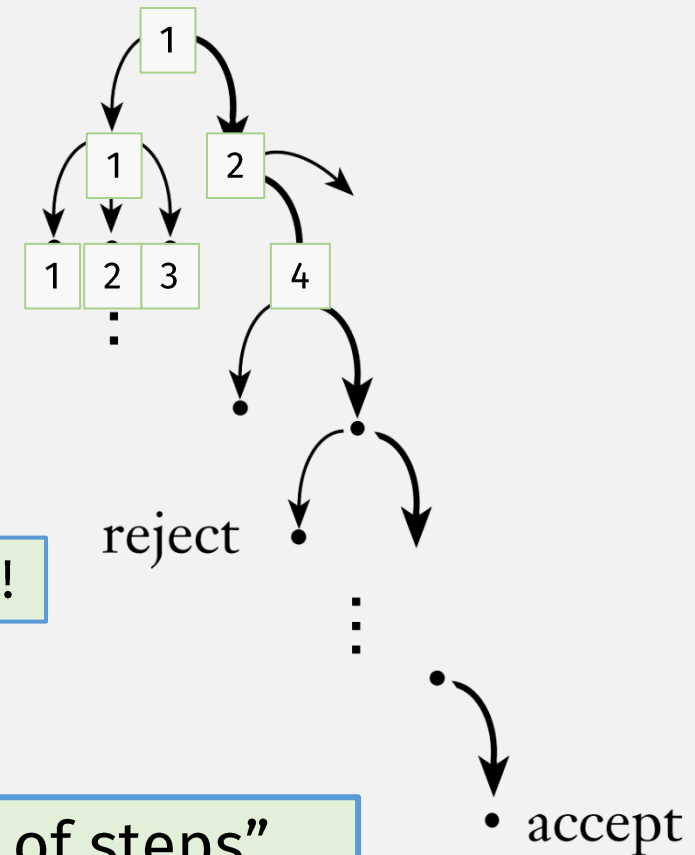
A **TM** and an **NTM** are “equivalent” ...

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

So how inefficient is it?

First, we need: a formal way to count “# of steps” ...

Nondeterministic computation

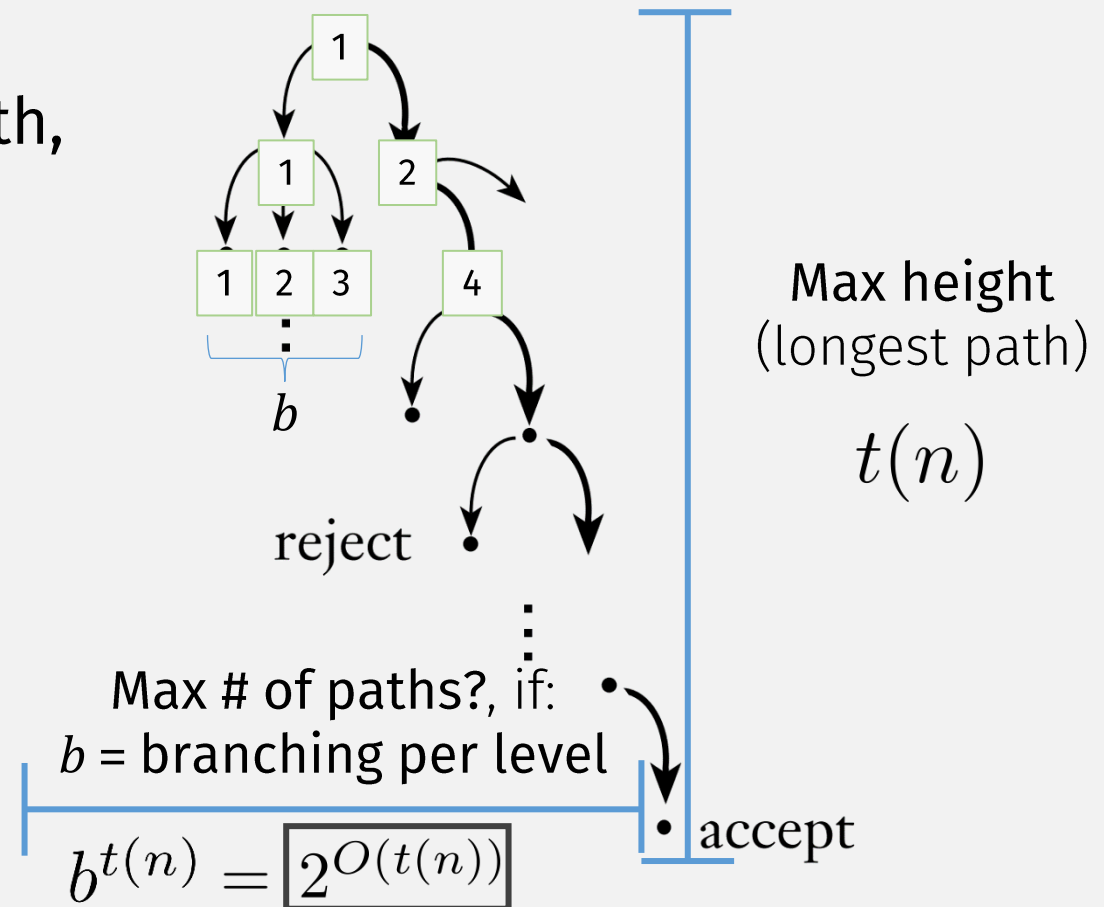


Flashback: Nondet. TM \rightarrow Deterministic TM

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

Nondeterministic computation



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

Polynomial Time (P)

Caveat:

This class: **polynomial time** = “good” (won’t take forever)

Real programmers: **polynomial time** = “eh” (pretty slow)

$O(1) = O(\text{yeah})$

$O(\log n) = O(\text{nice})$

$O(n) = O(k)$

$O(n^2) = O(\text{my})$

$O(2^n) = O(\text{no})$

$O(n!) = O(\text{mg})$

$O(n^n) = O(\text{sh*t!})$

Previously: Time Complexity

Running Time or Time Complexity is a property of decider TMs (algorithms)

Let M be a deterministic Turing machine that halts on all inputs. The *running time* or *time complexity* of M is the function $f: \mathcal{N} \rightarrow \mathcal{N}$, 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.

Depends on size of input

Worst case

Last Time: Time Complexity Classes

Big- O = asymptotic upper bound,
i.e., “only care about large n ”

Let $t: \mathcal{N} \rightarrow \mathcal{R}^+$ 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**

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

A language can have multiple deciding TMs, so could be in multiple time complexity classes

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

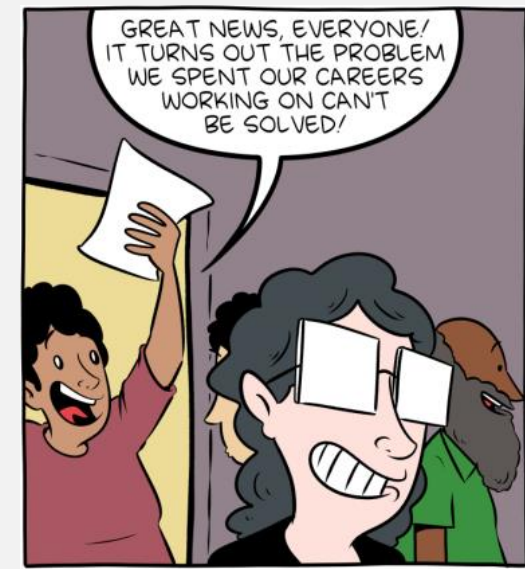
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

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

“Unsolvable” Problems

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



Mathematicians are weird.

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**.

Time it takes a hacker to brute force your password in 2025

Hardware: 12 x RTX 5090 | Password hash: bcrypt (10)

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	57 minutes	2 hours	4 hours
6	Instantly	46 minutes	2 days	6 days	2 weeks
7	Instantly	20 hours	4 months	1 year	2 years
8	Instantly	3 weeks	15 years	62 years	164 years
9	2 hours	2 years	791 years	3k years	11k years
10	1 day	40 years	41k years	238k years	803k years
11	1 weeks	1k years	2m years	14m years	56m years
12	3 months	27k years	111m years	917m years	3bn years
13	3 years	705k years	5bn years	56bn years	275bn years
14	28 years	18m years	300bn years	3tn years	19tn years
15	284 years	477m years	15tn years	218tn years	1qd years
16	2k years	12bn years	812tn years	13qd years	94qd years
17	28k years	322bn years	42qd years	840qd years	6qn years
18	284k years	8tn years	2qn years	52qn years	463qn years

As usual, 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:

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

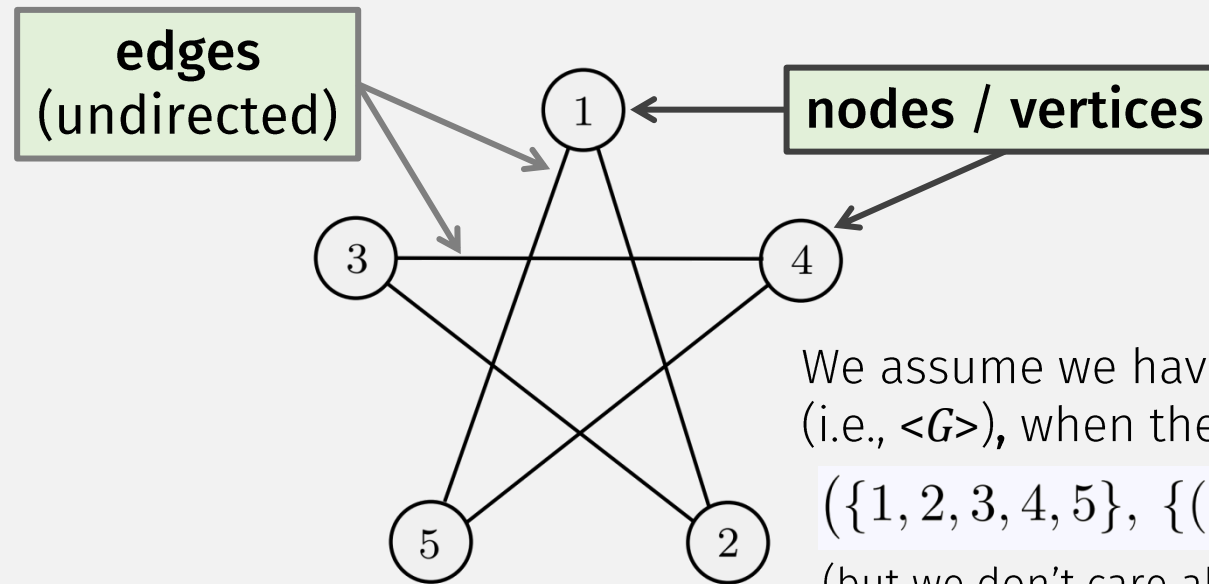
- A CFL Problem:

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 ...

A decider!

Interlude: Graphs (see Sipser Chapter 0)



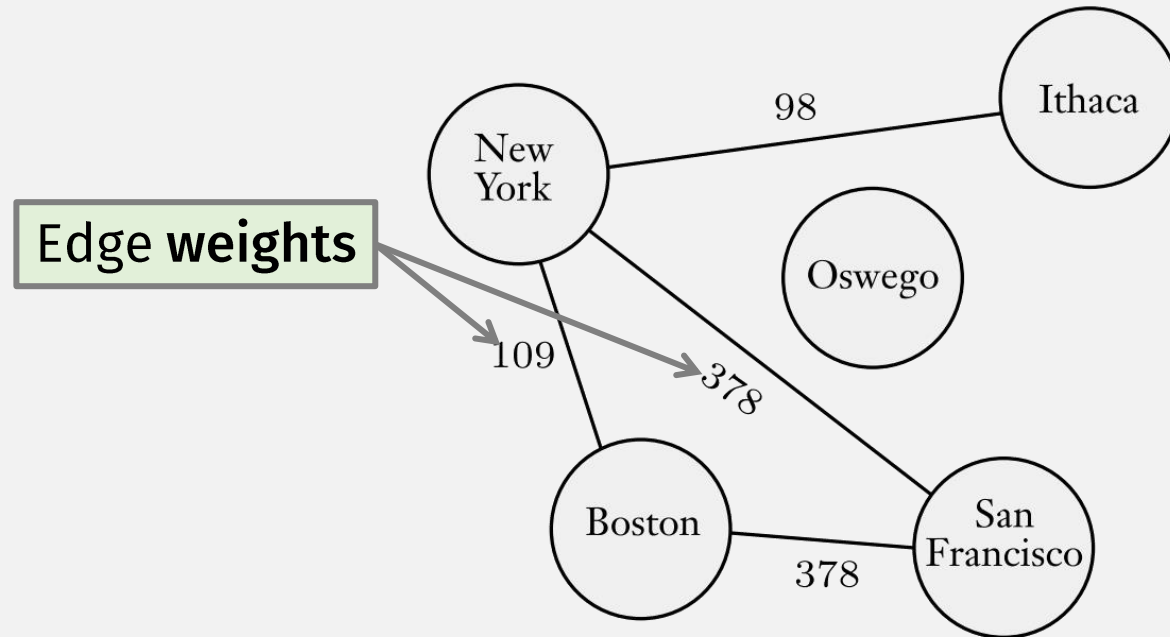
We assume we have some string encoding of a graph (i.e., $\langle G \rangle$), when they are args to TMs, e.g.:

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

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

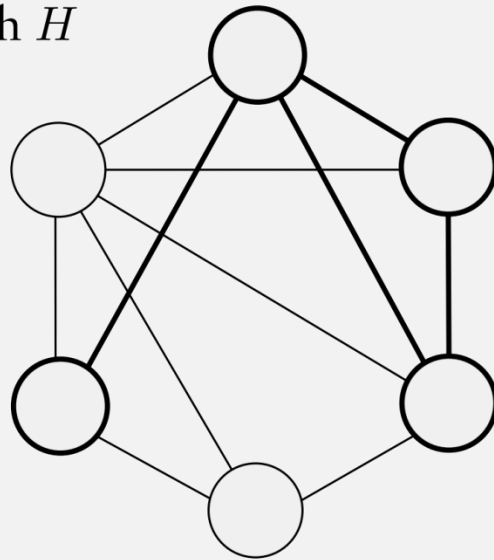
- **Edge** defined by two **nodes** (order doesn't matter)
- Formally, a **graph** = a pair (V, E)
 - Where V = a set of nodes, E = a set of edges

Interlude: Weighted Graphs



Interlude: Subgraphs

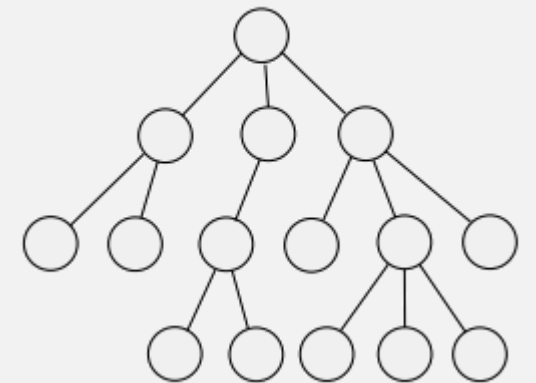
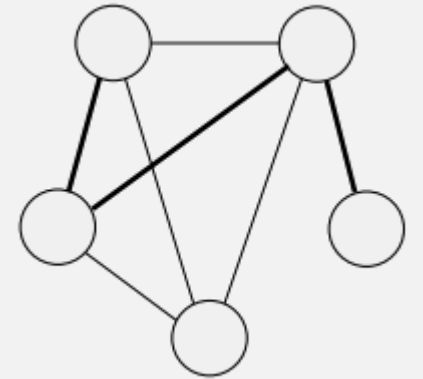
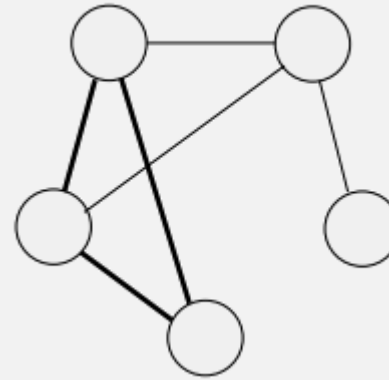
Graph H



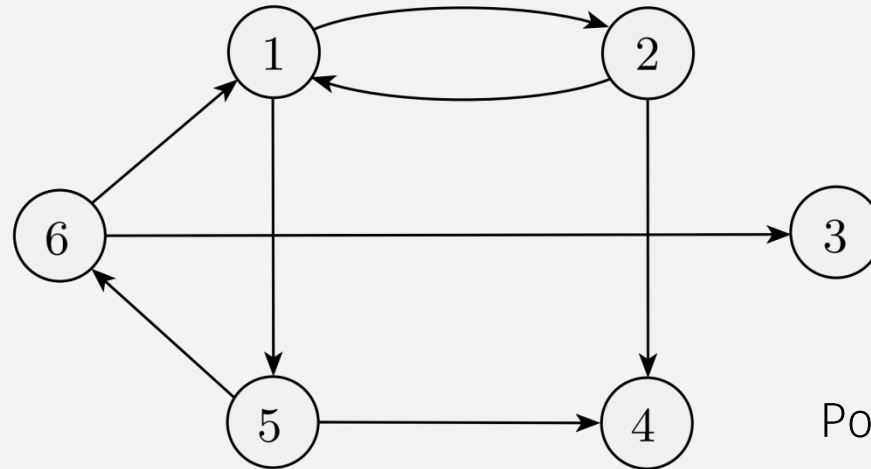
Subgraph G
shown darker

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:

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

- **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

Each pair of nodes included twice

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 **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

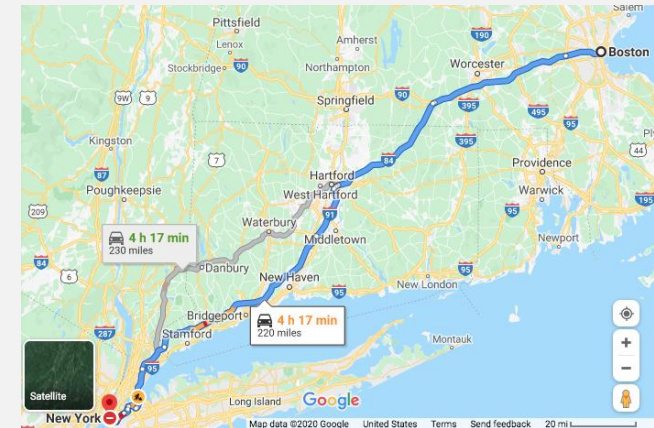
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

A Graph Theorem: $PATH \in P$

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

(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 combination (ordering) of all vertices,
 - see if any connect s to t
 - If $n = \#$ vertices, then $\#$ paths $\approx n^n$ or $n!$ (worse than $2^{O(n)}$)

A decider!

A Graph Theorem: $PATH \in P$

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

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: $PATH \in P$

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

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

A Graph Theorem: $PATH \in P$

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

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**
- 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: $PATH \in P$

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

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*.”

(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: $PATH \in P$

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

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
- 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

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

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

A Graph Theorem: $PATH \in P$

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

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*.”

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

$O(n^3)$

(For practical purposes, not a great algorithm, but it's in P ! i.e., “solvable”)

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
- Total = $1 + 1 + O(n^3) = O(n^3)$

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

A Number Theorem: $RELPRIME \in P$

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

- Two numbers are **relatively prime**: if their $\text{gcd} = 1$
 - $\text{gcd}(x, y)$ = largest number that divides both x and y
 - E.g., $\text{gcd}(8, 12) = \boxed{??}$
- Brute force (**exponential**) algorithm deciding $RELPRIME$:
 - Try all of numbers (up to x or y), see if it can divide both numbers
 - Q: Why is this exponential?
 - HINT: What is a typical “representation” of numbers?
 - A: binary numbers
(if $x = 2^n$, then trying x numbers is exponential in $n =$ the number of digits)
- A gcd algorithm that runs in **polynomial** time:
 - Euclid’s algorithm

A GCD Algorithm for: $RELPRIME \in P$

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

Modulo
(i.e., remainder)

$$\begin{aligned} 15 \bmod 8 &= 7 \\ 17 \bmod 8 &= 1 \end{aligned}$$

cuts x (at least) in half
every loop, requires:

$\log x$ loops

The Euclidean algorithm E is as follows.

$E =$ "On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y .
4. Output x ."

$O(n)$

Each number is
cut in half every
other iteration

Total run time (assume $x > y$): $2 \log x = 2 \log 2^n = O(n)$,
where $n =$ number of binary digits in (ie length of) x