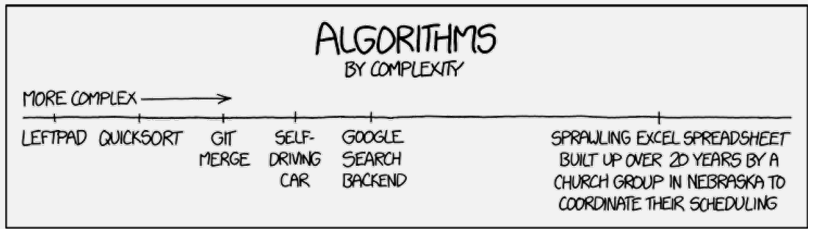
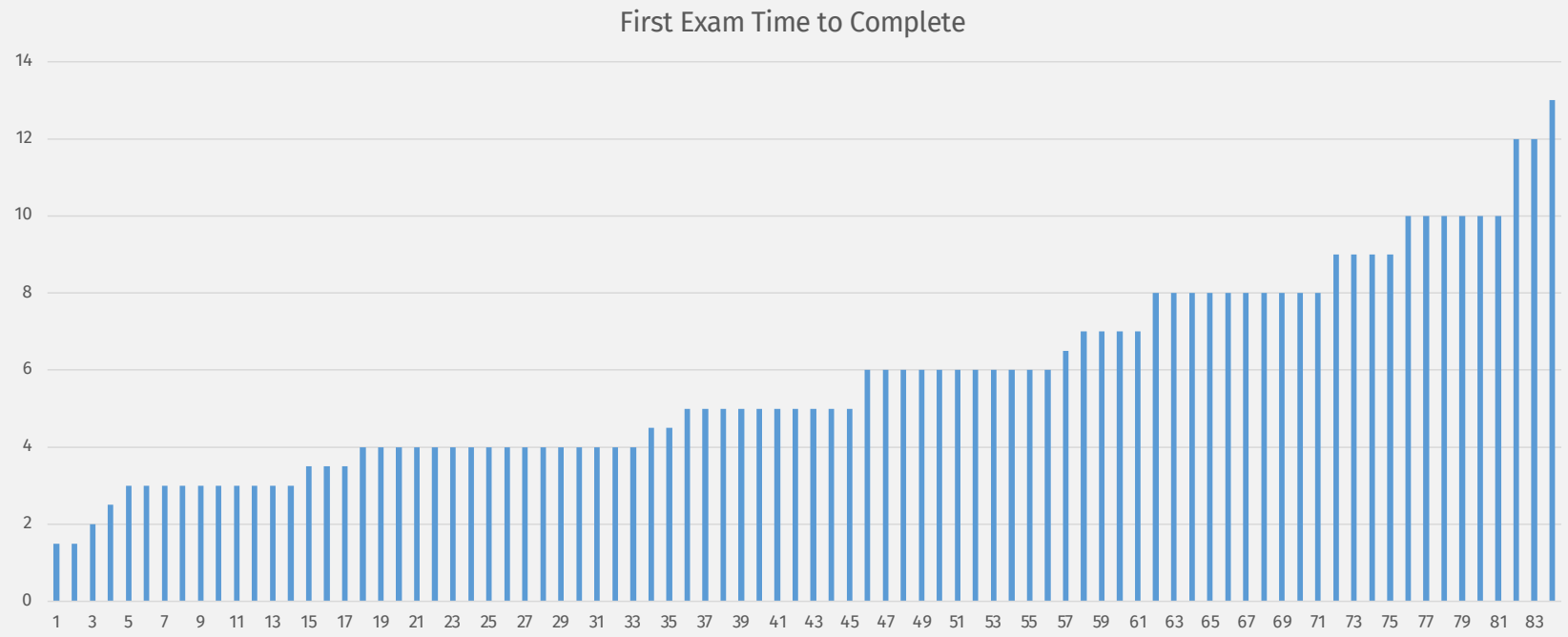
Time Complexity



Time to Complete first exam x # of subs




autograder
HW
line
long
class
hours
went
exam
One
fun
also
code
much
bug
NFA
able
expected
good
fix
2
vs
think
homework
tests
10
lve
old
fine
lot
questions
time
23
test
3
may
idea
Didnt
go
get
thought
way
really
bit
fair
well
like
4
7
took
use
part
25
next
spent
5
felt
dev
bugs
pretty
unit
Overall
run
cases
epsilon
auto
tech
set
build
nice
within
difficult
far
cause
least

Announcements

- Reminder: How's that Exam going?
  - I'm a better programmer. But not _that_ much better.
  - Auto still WIP on that last one. Shouldn't be a show-stopper here for you.
- Someone asked for an extension on HW9. But I was also in
- HW10 coming soon!

IMPLEMENTATION PROJECTS IN A COMPUTING THEORY COURSE

Tammy VanDeGrift
Computer Science, Shiley School of Engineering
University of Portland
Portland, OR 97203
503-943-7256
vandegri@up.edu

INFORMATION PAPER

International Journal of Recent Trends in Engineering, Vol. 1, No. 2, May 2009

Enhancing Theory of Computation Teaching Through Integration with other Courses

Mukta Goyal, Shelly Sachdeva

Department of Computer Science
Jaypee Institute of Information Technology University, Noida, India

mukta.goyal@jiit.ac.in,shelly.sachdeva@jiit.ac.in

Abstract:-Teaching and learning, theory of computation is a challenging task from both the ...tions and ...esentation ...courses is ...nteraction

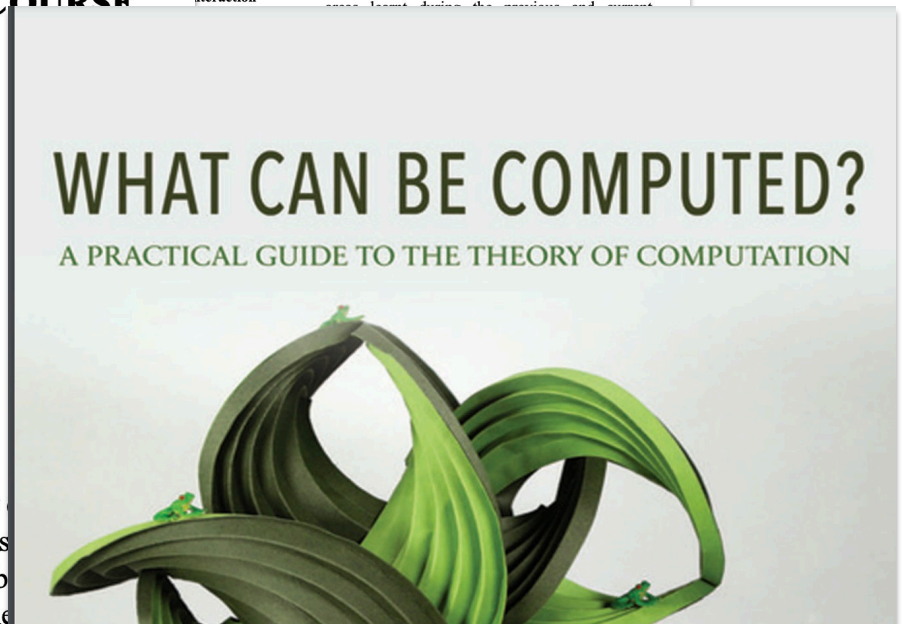
manner. Paper [2] inculcated a constructivist approach and powerful use of technology to achieve significant learning of course.
However, what remains unaddressed is the relationship between automata and other subject

A "Big-Ideas" Computation Theory Course for the Undergraduate

Arnold L. Rosenberg
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
rsnbrg@cs.umass.edu

Abstract

A "big-ideas" approach to an undergraduate Computation Theory
is described. The aim of this approach to the Theory is to focus the s
on those of the Theory's concepts and tools that are more likely to b
vant to a student's non-theoretical endeavors. By explaining why the



John MacCormick. CACM Oct. 2020

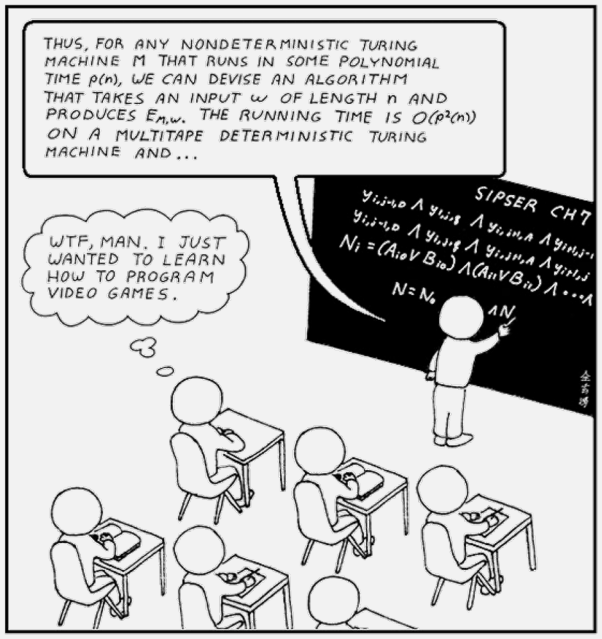
Back to Top

Using Real Computer Programs to Complement Automata

Another technique for increasing student engagement and connections with other parts of the CS curriculum is to employ code in a real programming language. This can provide a beneficial supplement to the automata and grammars that typically dominate a course in theory of computation. Formal models such as Turing machines are of course essential, especially for providing a rigorous definition of computation itself. However, it is possible to teach a mathematically rigorous theory course using a programming language as the primary model of computation. In this approach, the program model is layered over Turing machines as an underlying model, and Turing machines are still employed when required in certain proofs and definitions. A strong majority of CS theory textbooks do not employ a programming language as the primary computational model, but several authors have done so, for example using Python,[6] Ruby,[9] and a variant of LISP.[4,7] As an example of the approach, consider the Python program shown in the figure here.

16. You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.

Flashback: Single-tape TM “equiv to“ Nondet. TM



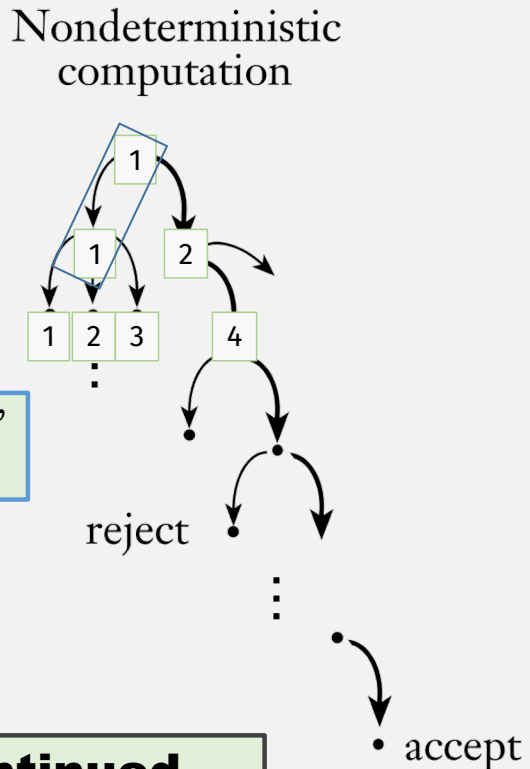
Flashback: Single-tape TM "equiv to" Nondet. TM

- Deterministic TM simulating nondeterministic TM:
  1. Number the nodes at each step
  2. Deterministically check every path, in breadth-first order (restart at top each time)
     - 1
     - 1-1
     - 1-2
     - 1-1-1
     - 1-1-2
     - and so on
  3. Accept if accepting config found

"This is the most inefficient algorithm ever"
--- Anonymous 3800 student

Exactly how inefficient is it???

Now we'll start to count "# of steps"

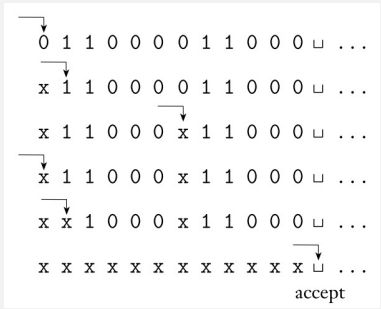


To be continued ...

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*."



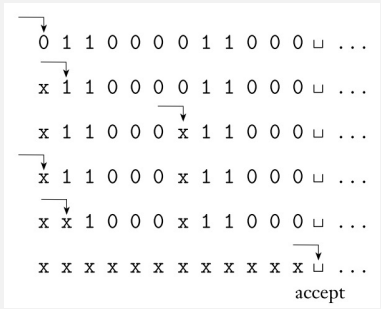
Number of steps (worst case), n = length of input:

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

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*."



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

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

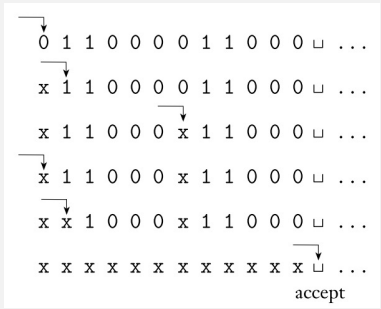
➢Lines 2 and 3 (loop):

- Steps for 1 iter: **n/2** steps to find "1" + **n/2** steps to return = **n** steps
- # iters: Each scan crosses off 2 chars, so at most **n/2** scans
- Total = 1 iter steps * # iters = **n (n/2)** = **$n^2/2$ steps**

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*."



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

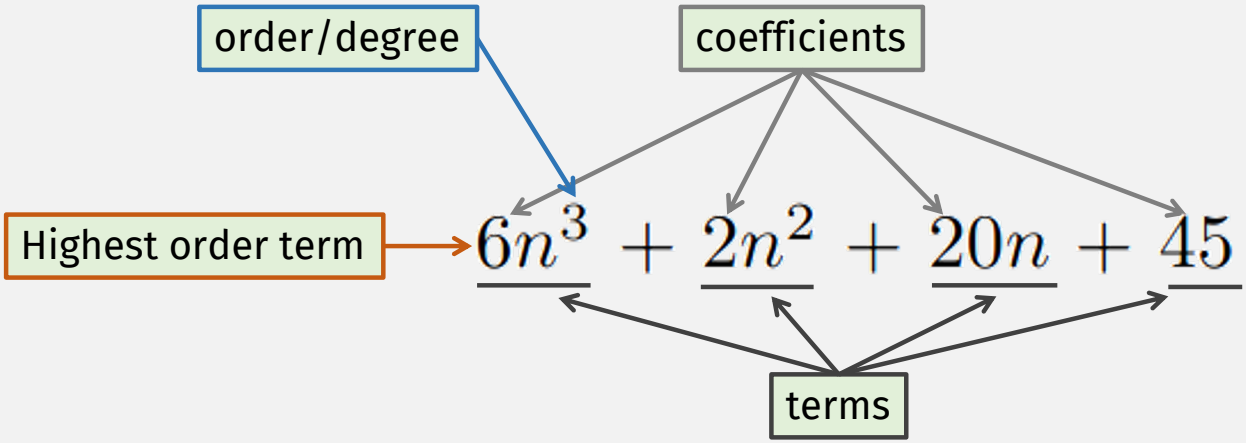
- TM Line 1:
  - **n** steps to scan + **n** steps to return to beginning = **2n steps**
- Lines 2 and 3 (loop):
  - Steps for 1 iter: **n/2** steps to find "1" + **n/2** steps to return = **n** steps
  - # iters: Each scan crosses off 2 chars, so at most **n/2** scans
  - Total = 1 iter steps * # iters = **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

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

n can be #characters, but also #states, #nodes, etc.

We can use any of these for n, bc they're correlated with input length

DEFINITION 7.1

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, say that M runs in time $f(n)$ and that M is an $f(n)$ time Turmachine. Customarily we use n to represent the length of the ut.

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

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: $\mathbf{n^2 + 3n}$
  - If $\mathbf{n} = 1$
    - $\mathbf{n^2} = 1$
    - $\mathbf{3n} = 3$
    - Total = 4
  - If $\mathbf{n} = 10$
    - $\mathbf{n^2} = 100$
    - $\mathbf{3n} = 30$
    - Total = 130
  - If $\mathbf{n} = 100$
    - $\mathbf{n^2} = 10000$
    - $\mathbf{3n} = 300$
    - Total = 10300
  - If $\mathbf{n} = 1000$
    - $\mathbf{n^2} = 1000000$
    - $\mathbf{3n} = 3000$
    - Total = 1003000
- $\mathbf{n^2 + 3n} \approx \mathbf{n^2}$ as n gets large
- asymptotic analysis only cares about large n

Definition: Big-O Notation

DEFINITION 7.2

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. 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.

- In English: Keep only highest order term, drop all coefficients
- Machine M_1 that decides $A = \{0^k 1^k \mid k \geq 0\}$
  - Is an $\mathbf{n^2 + 3n}$ time Turing machine
  - Is an $O(\mathbf{n^2})$ time Turing machine
  - Has asymptotic upper bound $O(\mathbf{n^2})$

Definition: Small-o Notation (less used)

DEFINITION 7.5

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

DEFINITION 7.2

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. 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.

Big-O arithmetic

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

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

Definition: Time Complexity Classes

DEFINITION 7.7

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

TMs have a running time, languages are in a complexity class

- Machine M_1 that decides $A = \{0^k 1^k \mid k \geq 0\}$
  - Is an $O(\mathbf{n^2})$ running time Turing machine
  - So A is in $\mathrm{TIME}(\mathbf{n^2})$

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*."

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*."

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

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 = $O(\mathbf{n})$ 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 = $O(\mathbf{n})$ steps

➢Lines 2, 3, 4 (loop):

- Steps for 1 iter: a scan takes $O(\mathbf{n})$ steps
- # iters: Each iter crosses off *half* the chars, so at most $O(\mathbf{\log n})$ scans
- Total: $O(\mathbf{n}) * O(\mathbf{\log n}) = O(\mathbf{n \log n})$ 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 = $O(\mathbf{n})$ steps
- Lines 2, 3, 4 (loop):
  - Steps for 1 iter: a scan takes $O(\mathbf{n})$ steps
  - # iters: Each iter crosses off *half* the chars, so at most $O(\mathbf{\log n})$ scans
  - Total: $O(\mathbf{n}) * O(\mathbf{\log n}) = O(\mathbf{n \log n})$ steps

➢ Line 5:

- $O(\mathbf{n})$ 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*."

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

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

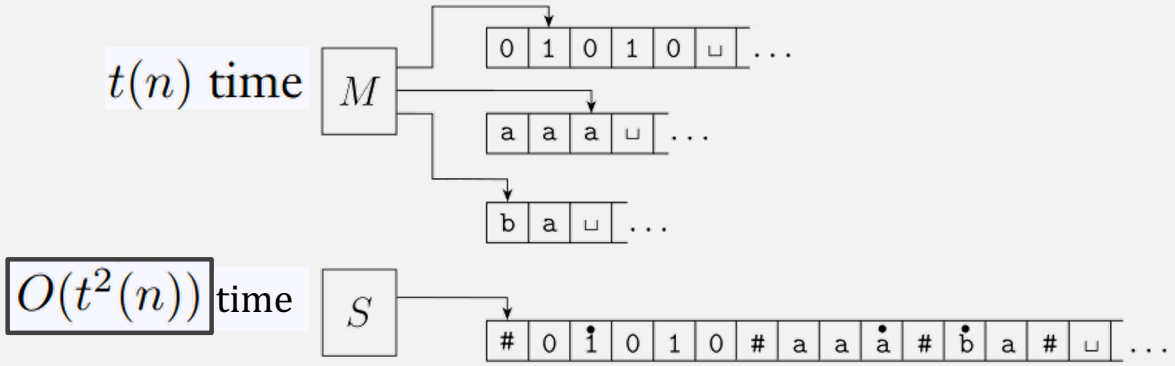
Interlude: Logarithms

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

Terminology: Categories of Bounds

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

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



- For single-tape TM to simulate 1 step of multi-tape:
  - Scan to find all "heads" = O(length of all M's tapes)
  - "Execute" transition at all the heads = O(length of all 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
- Total steps (single tape): $O(t(n))$ per step × $t(n)$ steps = $\boldsymbol{O(t^2(n))}$

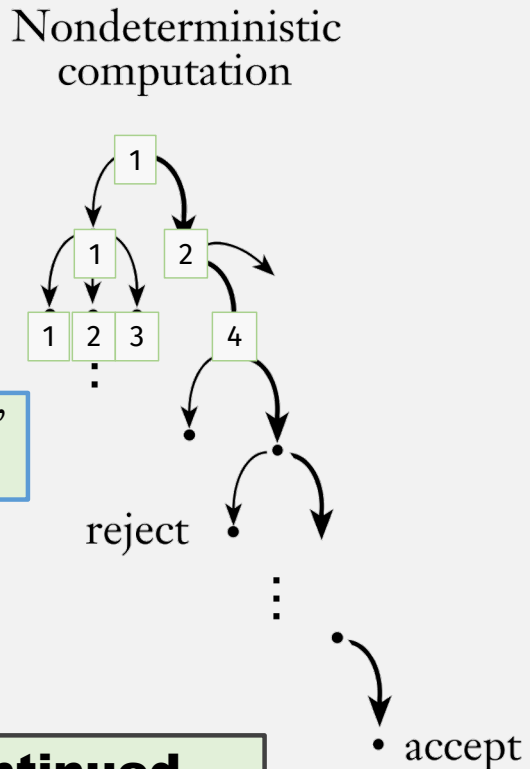
Single-tape TM vs Nondet. TM: # of steps

- Deterministic TM simulating nondeterministic TM:
  - Number the nodes at each step
  - Deterministically check every path, in **breadth-first order** (restart at top each time)
    - 1
    - 1-1
    - 1-2
    - 1-1-1
    - 1-1-2
    - and so on
  - Accept if accepting config found

"This is the most inefficient algorithm ever"
--- CS420 Spring 2021 student

Exactly how inefficient is it???

Now we'll start to count "# of steps"

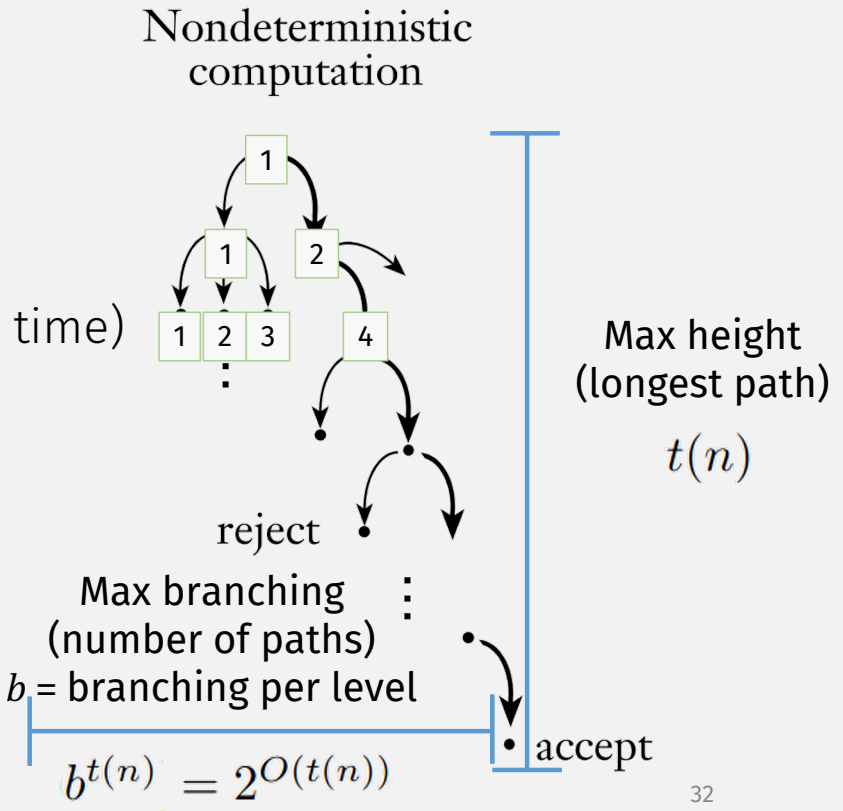


To be continued ...

Single-tape TM vs Nondet. TM: # of steps

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

- Deterministic TM simulating nondeterministic TM: ← $t(n)$ time
  - Number the nodes at each step
  - Deterministically check every path, in breadth-first order (restart at top each time)
    - 1
    - 1-1
    - 1-2
    - 1-1-1
    - 1-1-2
    - and so on
  - Accept if accepting config found



Summary

- 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

Next time: Specific Complexity Classes

DEFINITION 7.12

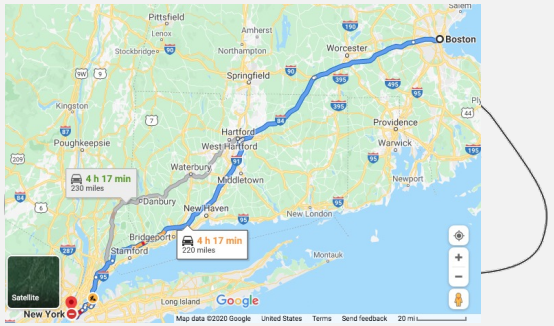
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

- Corresponds to “realistically” solvable problems
- In this class:
  - Problems in **P** = “solvable”
  - Problems outside **P** = “unsolvable”
    - These are usually “brute force” solutions that “try all possible inputs”

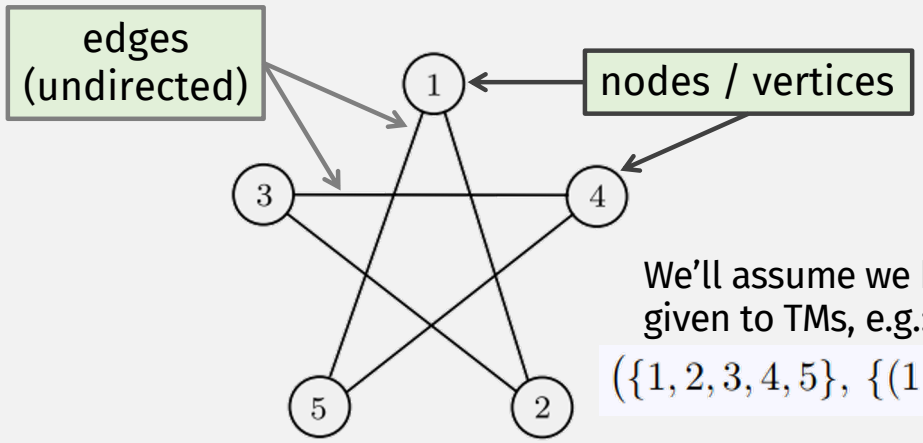
Next time: A Graph Theorem: $PATH \in \mathrm{P}$

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



- To prove that a language is in **P** …
- … we must construct a polynomial time algorithm deciding the lang
- A non-polynomial (i.e., exponential, brute force) algorithm:
  - Check all possible paths, and see if any connect s to t

Interlude: Graphs (see Chapter 0)



We'll assume we have some **string encoding**, <G>, given to TMs, e.g.:

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

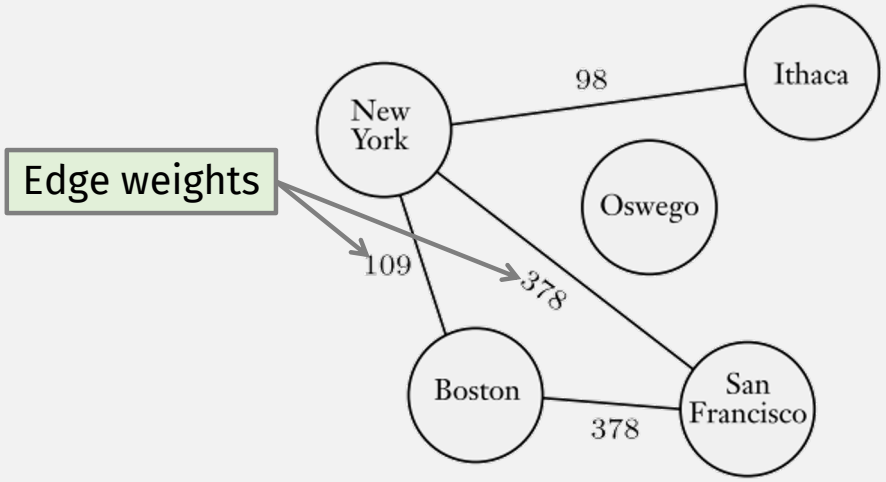
- Edge defined by two nodes (order doesn't matter)
- Formally, a graph = *(V, E)*
  - V = set of nodes, E = set of edges

Interlude: Graph Encodings

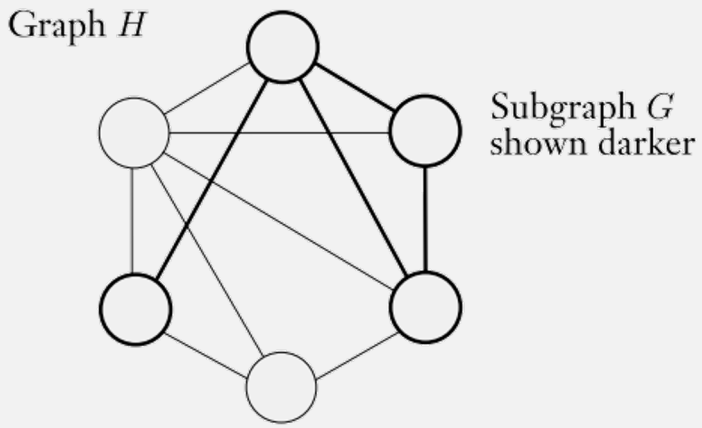
$$(\{1,2,3,4,5\},\ \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,1)\})$$

- In graph algorithms, "length of input" n = number of vertices
  - and sometimes number of edges
  - Not number of chars
  - So steps counted in terms of number of vertices
- Given a graph $G = (V, E)$ with $n = |V|$ vertices
- Max edges = $O(|V|^2) = O(n^2)$
- So # vertices + edges is polynomial in length of input
- Algorithm runs in time polynomial in the number of vertices ⇔ algorithm runs in time polynomial in the length of input

Interlude: Weighted Graphs

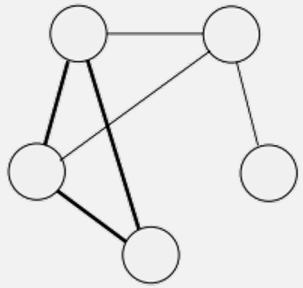


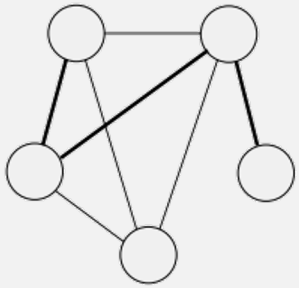
Interlude: Subgraphs

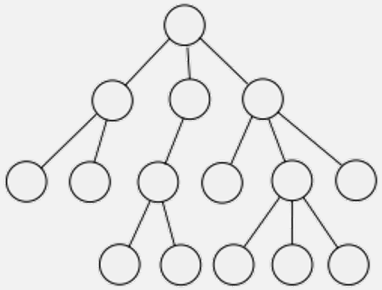


Interlude: Paths and other Graph Things

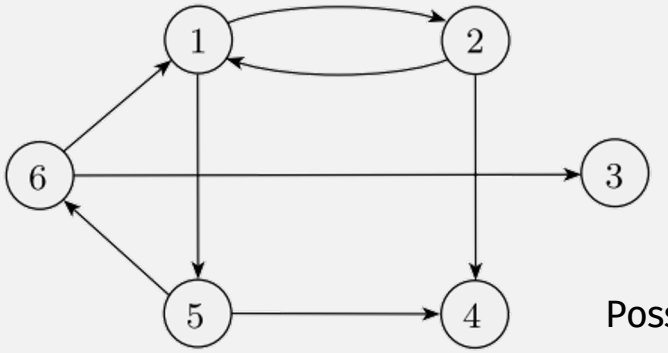
- Path
  - A sequence of nodes connected by edges
- Cycle
  - A path that starts/ends at the same node
- Connected graph
  - Every two nodes has a path
- Tree
  - A connected graph with no cycles







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
- A "degree" of a node is the number of edges connected to the node
  - Nodes in a directed graph have both indegree and outdegree

Check-in Quiz

On gradescope