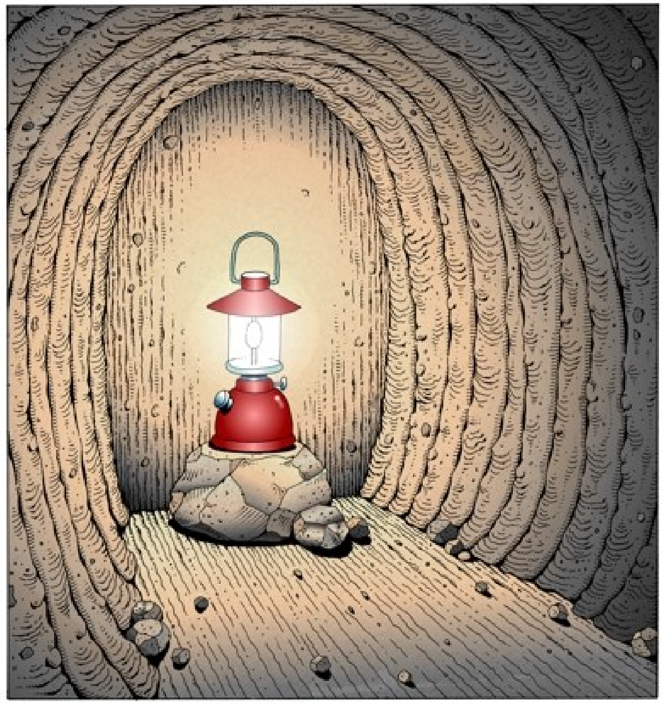
One Last Class…

- Team Challenge
  - Take a look at **feedback**, not just the grade!
  - Use it as guidance on what to **focus** as you prepare for final exam
- Final
  - Comprehensive, weighted towards chapters after midterm-2
- Course Evaluations



The light at the end of the tunnel

CHAPTER 7

Time Complexity

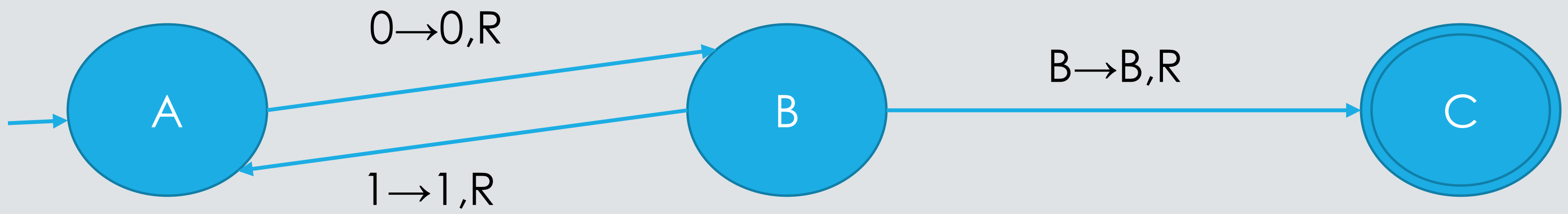
Complexity Theory

- In **computability theory**, we asked the question: Is it possible to *solve* a problem P?
  - To answer the question we explored:
    - What is a computation
    - What is a problem
    - What does it mean to solve a problem
- In **complexity theory**, we ask the question: Is it possible to solve P *efficiently*?
  - To answer that question we will clarify
    - What does complexity mean
    - What is an efficient solution to a problem

Time Complexity

- Definition
  - Let M be a deterministic TM that halts on all inputs. The time complexity of M is a function f: $\mathbb{N} \rightarrow \mathbb{N}$, where f(n) is the *maximum number of steps* that M uses on any *input* of length n.
- Intuitive idea
  - In determining the time complexity of a TM, we analyze the "algorithm" of TM, i.e., the number of steps that the algorithm uses given an input
  - We consider *worst-case*, i.e., the longest running time of all inputs of a particular length

Example



Input	Decision	No. of Steps
00	Reject	1
011	Reject	2
010	Accept	4

$f(n) = n + 1$

Time complexity of TM

An Easier Approach

- In complexity theory, we rarely need an exact value for a TM's time complexity
  - Usually, we are curious with the long-term growth rate of the time complexity
- Example:
  - Assume the time complexity of a TM is $f(n)$ $3n + 5$
    - Doubling the length of the string roughly doubles the worst-case runtime
- The question is….
  - How do we describe the time complexity based on the “information we care about”

Time Complexity and Big-Oh

- We can define **complexity classes**, based on the time complexity of TMs expressed in terms of Big-Oh notation
  - Constant do not matter
  - Only dominant term matters
- The time complexity class **TIME(f(n))** is the set of languages decidable by a TM with runtime **O(f(n))**
  - Examples
    - TIME(n)
      - TM that decides a regular language
    - TIME(n^2)
      - Palindrome

Comparison of Run Times

Size	1	Log n	n	n log n	n^2	n^3	2^n
100	1μs	7μs	100μs	0.7ms	10ms	<1min	40 quadrillion yrs
200	1μs	8μs	200μs	1.5ms	40ms	<1min	More…
…							
500	1μs	9μs	500μs	4.5ms	250ms	4 min	
…							
1000	1μs	10μs	1000μs	10ms	1000ms	22 min	

Polynomial functions scale "well"

Exponential functions scale "terribly"

Take Away…

A language L can be solved **efficiently** if there is a TM that decides it in **polynomial time**.

The Class P

P is the class of languages that are decidable in polynomial time on a *deterministic single-tape* TM.

$P = \bigcup_k TIME\ (n^k)$

Theorem: Let t(n) be a function, where $t(n) \geq n$. Then every **t(n) nondeterministic** single-tape TM has an equivalent $\mathbf{2^{O(t(n))}}$ time **deterministic** single-tape TM

Examples:

Is there a path from $node_A$ to $node_B$?

Search a substring on a given input string?

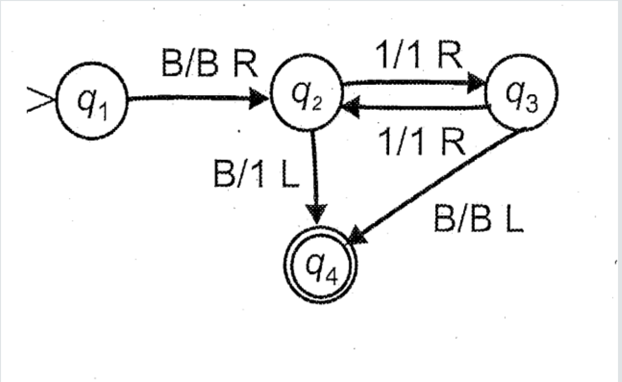
P corresponds to the class of problems that are **realistically solvable** on a computer

How to Prove Languages are in P

- Directly prove the language is in P
  - Build a decider for the language L
  - Prove that the decider runs in time $O(n^k)$
- Use closure properties.
  - Prove that the language can be formed by appropriate transformations of languages in P
- Reduce the language to a language in P
  - Show how a polynomial-time decider for some language L' can be used to decide L

Proof - Directly

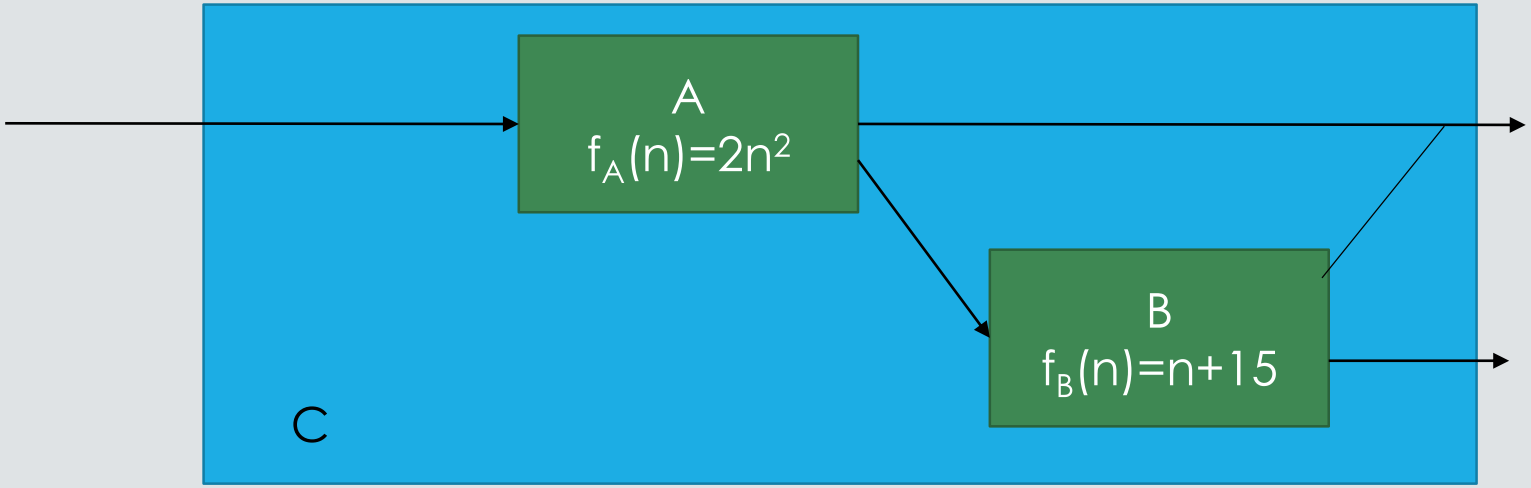
- Output unary odd numbers



$tc_M(n) = n + 2$

Proof - Closure

- Union Operation



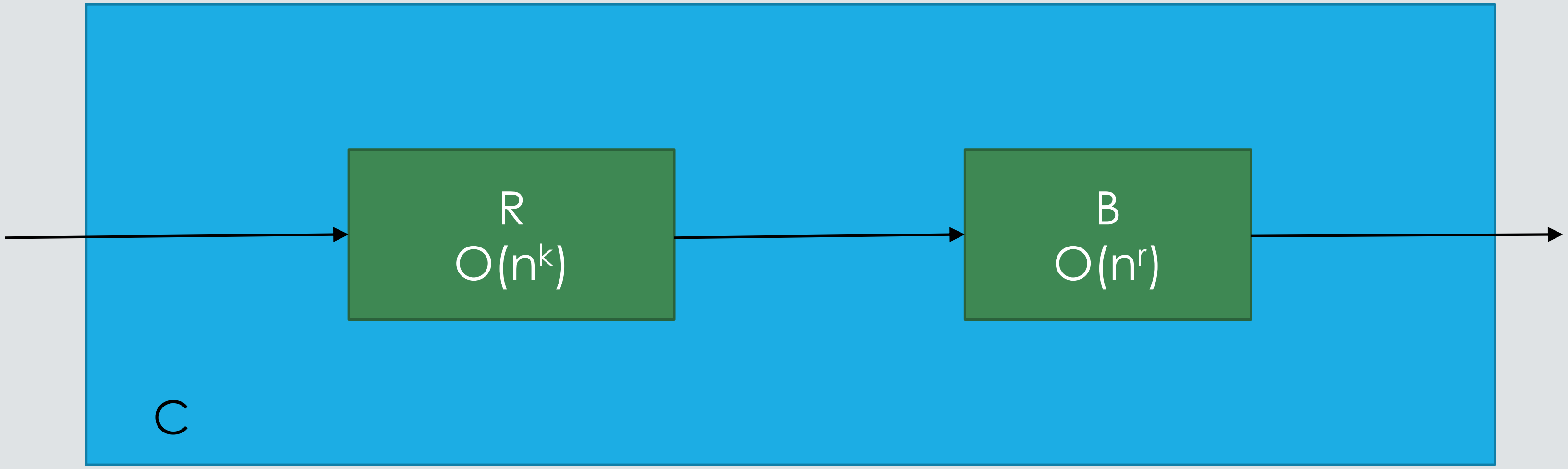
$$\mathbf{f_C(n)} = \max(f_A(n),\ f_B(n)) = \max(O(n^2), O(n)) = \mathbf{O(n^2)}$$

Let M_A and M_B be any two arbitrary TMs such that their complexity is polynomial.
Let $M_C = M_A \cup M_B$.

Given a string of length n, test for membership in one language and then, if the input is not in M_A, test for membership in M_B.

$tc_{M_C}(n) \leq tc_{M_A}(n) + tc_{M_B}(n)$, and since both $tc_{M_A}(n)$ and $tc_{M_B}(n)$ in **P**, $tc_{M_C}(n)$ is in **P**.

Proof - Reduction



$$\mathbf{f_C(n)} = O(n^k) + O((n^k)^r) = O(\mathbf{n^{rk}})$$

Remember! If I do not know the complexity of a new problem, but I can turn it in polynomial time into another problem known to be solved in polynomial time, then I can be sure that my new problem has a solution in polynomial time, i.e., an efficient solution

Take Away…

- P is the complexity class of yes/no questions that can be solved in polynomial time
  - Problems that can be solved in polynomial time using a **deterministic, single-tape** TM
- P is closed under many operations, such as union and intersection
- P is closed under polynomial-time reductions

The Class NP

NP is the class of languages that are decidable in polynomial time on a *nondeterministic single-tape* TM.

NP = $\bigcup_k NTIME\ (n^k)$

NP corresponds to the class of problems that are **realistically verified** on a computer

Examples:

Hamiltonian Path

Composite number

A Problem in NP Class

- Does a Sudoku grid have a solution?

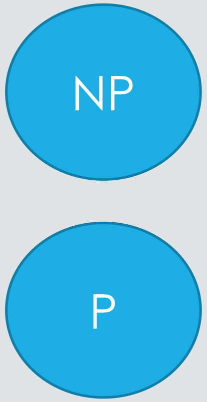
M = “On input <S>, an encoding of a Sudoku puzzle:
Nondeterministically guess how to fill in all the squares.
Deterministically check whether the guess is correct.
If so, accept; if not, reject.”

		6		1			2	
4	3			7		6		
		9		4		8	7	3
				8	4	1		
3				5				9
		5	9	3				
2	8	1		6		5		
		7		9			6	2
	6			2		4		

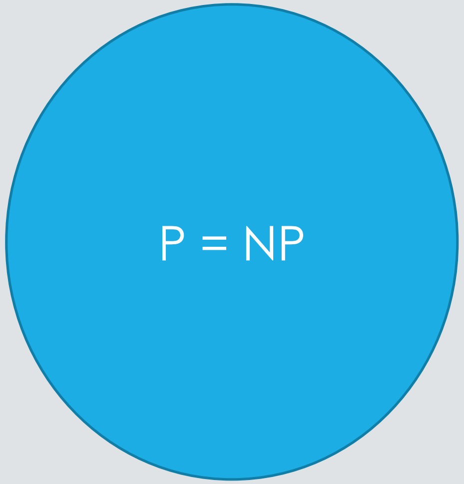
If we allow for a generalized Sudoku board of arbitrary size:
There are polynomially many grid cells to fill in
Checking the grid takes polynomial time

Take Away…

- Basically,
  - NP problems are known to be efficiently verifiable, but have no known efficient solutions
- P= the class of languages for which membership can be **decided** quickly
- NP = the class of languages for which the membership can be **verified** quickly
- Relationship between P and NP



P ≠ NP ?

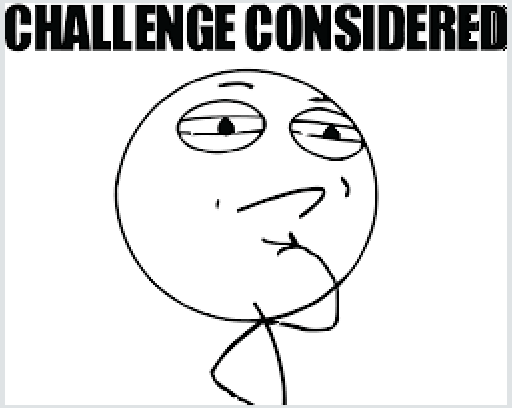


P = NP ?

Why Care if P=NP?

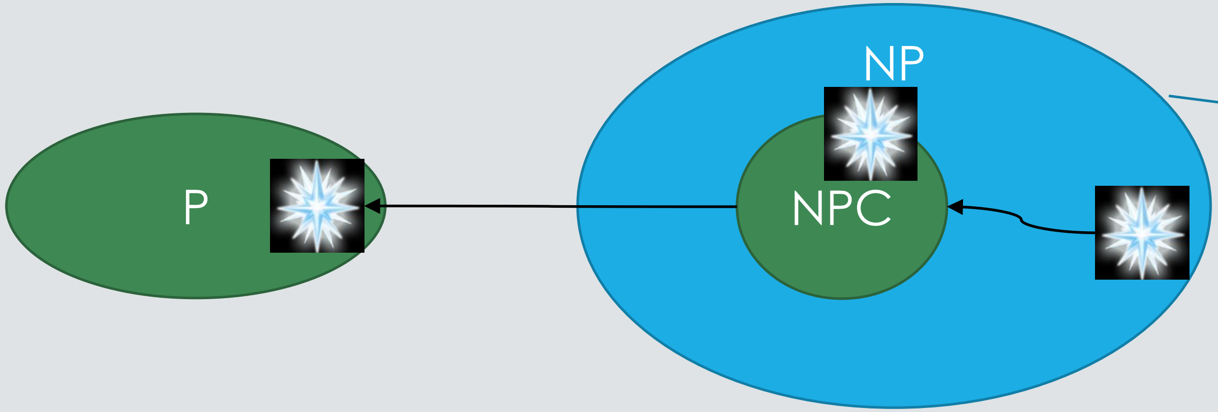
- If P = NP:
  - A huge number of seemingly difficult problems could be solved efficiently
  - Our capacity to solve many problems will scale well with the size of the problems we want to solve
- If P ≠ NP:
  - Enormous computational power would be required to solve many seemingly easy tasks
  - Our capacity to solve many problems will fail to keep up with our curiosity

The Clay Mathematics Institute has offered a $1,000,000 prize to anyone who proves or disproves P = NP



NP-Completeness

- A language Q is called **NP-Hard** if for every language L ∈ NP, L is reducible to Q in polynomial time.
- An NP-hard language that is also in NP is called **NP-Complete**



*This is **not** set notation, just a simple illustrations on how we can (potentially) connect NPC and P*

To prove that P = NP we only need to find a polynomial algorithm for an NP-Complete problem to achieve this goal


are we
there yet?