San José State University
Department of Computer Science

Ahmad Yazdankhah
ahmad.yazdankhah@sjsu.edu
www.cs.sjsu.edu/~yazdankhah

Computation Complexity

Lecture 27
Day 31/31

CS 154
Formal Languages and Computability
Spring 2019

Agenda of Day 31

- About Final Exam
- Summary of Lecture 26
- Lecture 27: Teaching ...
  - Computation Complexity

About Final Exam

Reminder 2

- **Value**: 20%
- **Topics**: Almost everything covered from the beginning of the semester
- **Type**: Closed all materials

Section	Date	Time	Venue
01 (TR 3:00)	Tuesday, May 21	2:45 – 5:00 pm	DH 450
02 (TR 4:30)	Monday, May 20	2:45 – 5:00 pm	DH 450
03 (TR 6:00)	Thursday, May 16	5:15 – 7:30 pm	DH 450

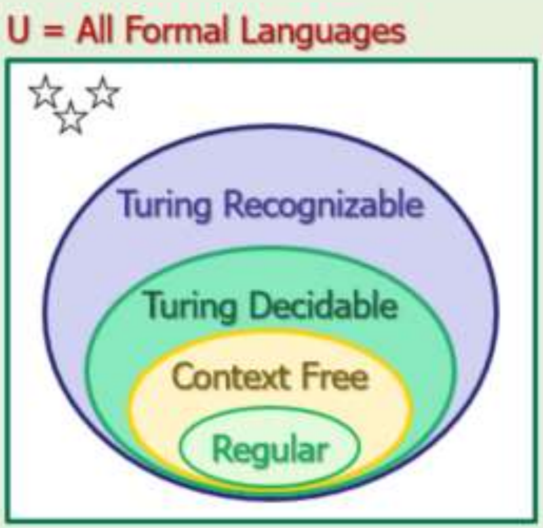
- We won't need whole 2:15 hours.
- As usual, I'll announce officially the type and number of questions via Canvas. (study guide)

Summary of Lecture 27: We learned ...

Computability

- Turing Thesis
  - Any computation carried out by a mechanical procedure can be performed by a TM.
  - We cannot prove it and we could not refute it yet.
- For Turing-recognizable languages, we have problem with the rejecting of the strings of $\overline{L}$.
  - Because the TM might get stuck in a forever loop.
- We prefer TMs that always halt.
- We called these TMs as deciders.

- A language is called Turing-decidable (or just decidable) if there is a decider for it.

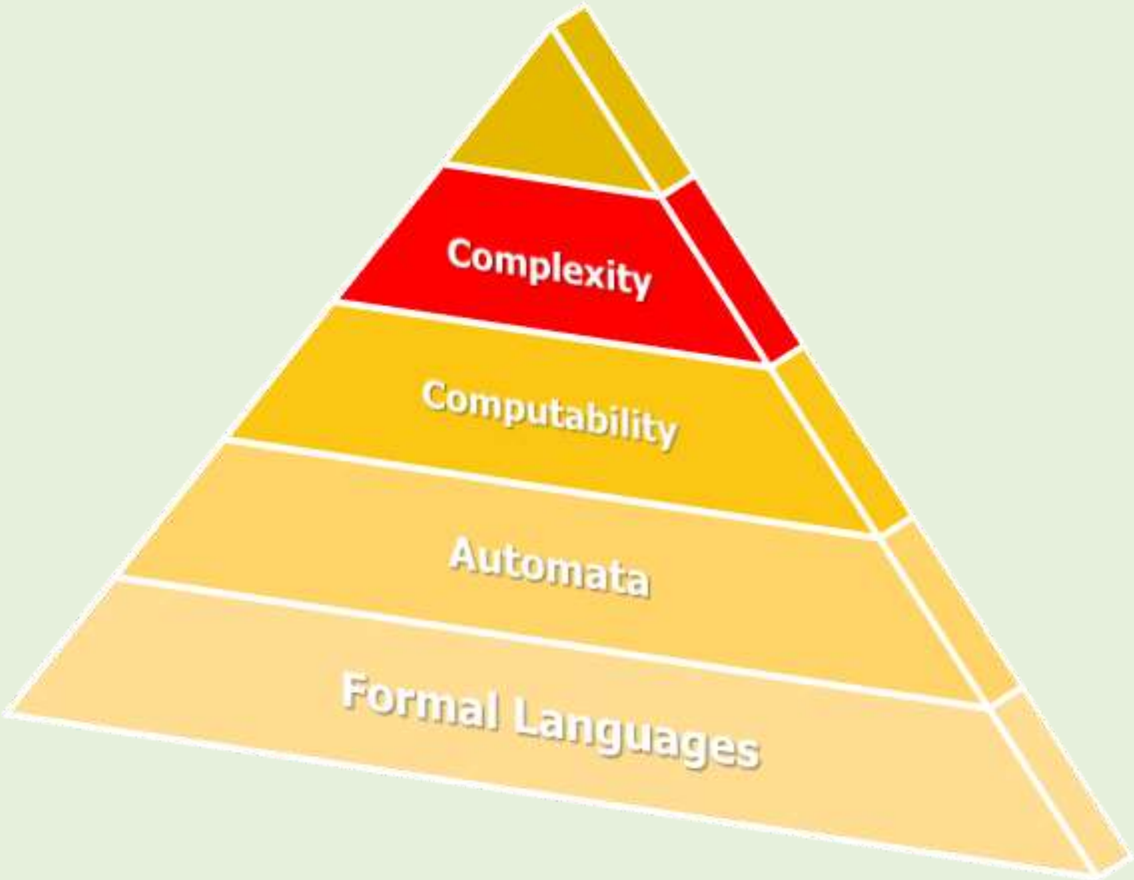


- Universal TM is a TM that simulates other TMs.
- Halting problem shows the limitation of the theory of the computation.

Any question?

The Big Picture of the Course

Recap



Objective of This Lecture

- What is complexity?
- What do we mean when we say:

  Computation A is more complex than computation B.

- How do we classify the problems based on their complexity?
- What classes of complexities do we have?

Computation Complexity

Introduction

- So far, "efficiency" was not our concern!

- Recall that in our designs, ...
- ... especially when we're dealing with nondeterministic machines, ...
- we said:

  It doesn't matter how much resources we are consuming!

- But in real world, we do care about it.
  - In fact, it's one of the most essential concerns in computer science.

Introduction

- In this lecture, we'll deal with this concern briefly.
- But "CS146: Data Structure and Algorithms" course is the place to talk about this concern in detail.
- Let's start with this question:

"What is computation complexity?"

What is Computation Complexity?

- Specifically, what do we mean when we say:

  Computation A is more complex than computation B?

- It means, computation A needs more RESOURCES.

- So, we measure the computation complexity by the amount of required resources.

Definition

- "Computation complexity" (aka efficiency) is the amount of required resources.

What is Computation Complexity?

What are the resources?

- The resources could be:
  - Time
  - Space
  - Number of CPUs
  - Energy
  - etc..

- But time and space are usually our main concerns.

What is the Computation Complexity?

- So, we can talk about two types of complexities:

    1. Time-complexity
    2. Space-complexity

- Storage is getting cheaper and cheaper but time is always "Gold"!

- In this lecture we'll focus on "time-complexity" that is usually of more concerns.

- Space-complexity is handled pretty much the same way as time-complexity.

Time-Complexity

Time-Complexity: Required Background

- To understand this topic, we'd need the following backgrounds:

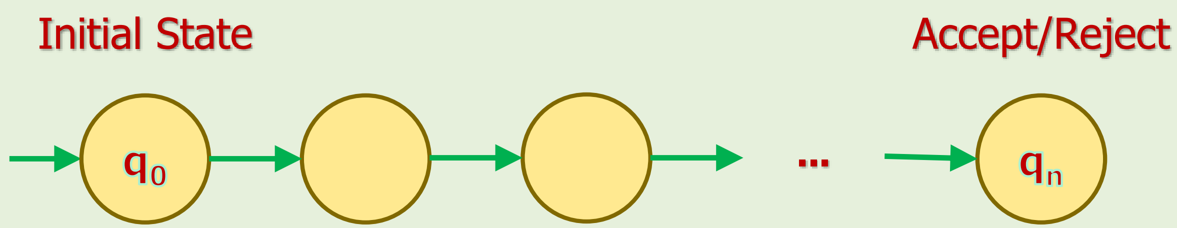
  1. The concept of deterministic (standard) and nondeterministic TMs (we are so familiar with these two concept!)
  2. Growth rate (will be reviewed quickly!)
  3. Asymptotic notations: Big-O (will be reviewed quickly!)

- Before going further, we need to define the "computation time".

What is Computation Time?

- For any computation, the machine makes some transitions starting from the initial state until it halts.
  - Recall that if it doesn't halt, there won't be any computation!

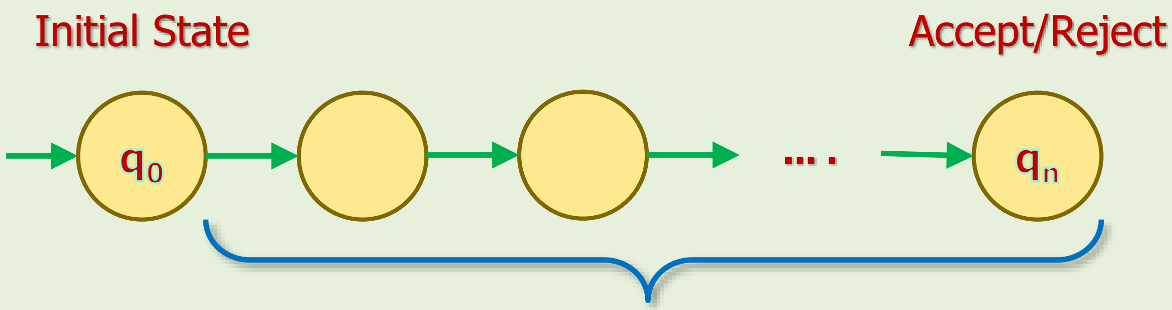
- For example, the following one-dimensional projection shows the computation of a process.



Computation Time of Standard TMs

Definition

- The computation time of a standard TM (single process) is the number of transitions from when the process starts until it halts.



Computation Time = Number of Transitions

- What would be the computation time when a machine gets stuck in an infinite-loop?

Computation Time of Nondeterministic TMs

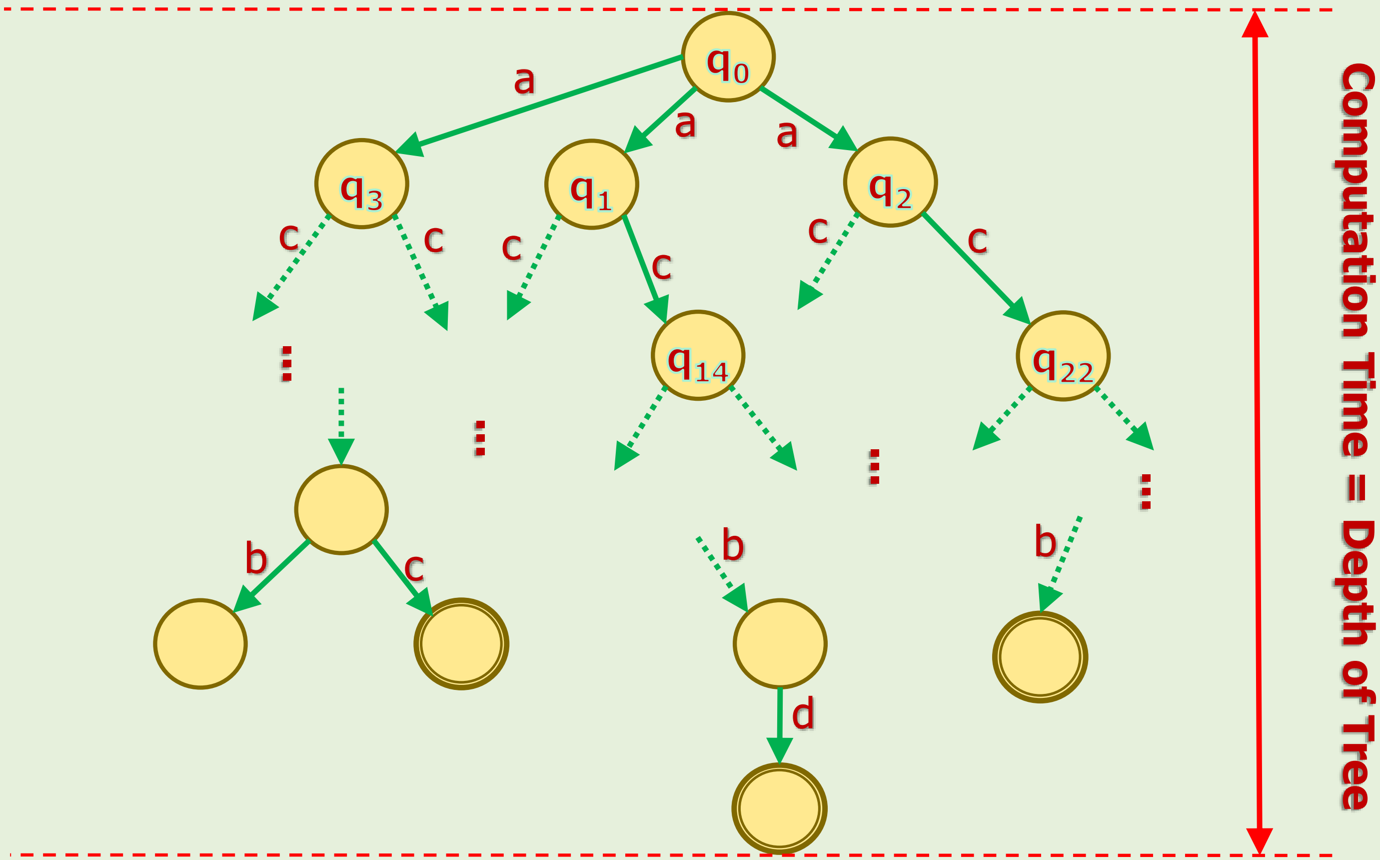
- Recall that a nondeterministic TM is ...
- ... a collection of some standard TMs.
- And all processes run concurrently.

Definition

- The computation time of a nondeterministic TM is the computation time of the longest process.

- In the next slide, we combined all processes of a nondeterministic TM in a tree that we called "processes tree".
- The computation time would be the depth of the tree.

Computation Time of Nondeterministic TMs



- Note that just input symbols of the labels are shown for readability purpose.

Growth Rate

Definition

- How fast the required resources grow when the input size becomes larger.
- This is called "growth rate of resources".
- Definitely, slower growth of the required resources is desirable.
- To understand this concept deeply, let's be more precise!

Growth Rate of Resources

- Consider the following deterministic automaton:

- We define the computation time of this machine by f(n) ...
  - ... that is a function of the input size n.

- What does f(n) look like for different types of automata?
- For example, if M is a DFA, how f(n) looks like?

$$f(n) = n \text{ (linear function)}$$

- What if the automaton is a TM?

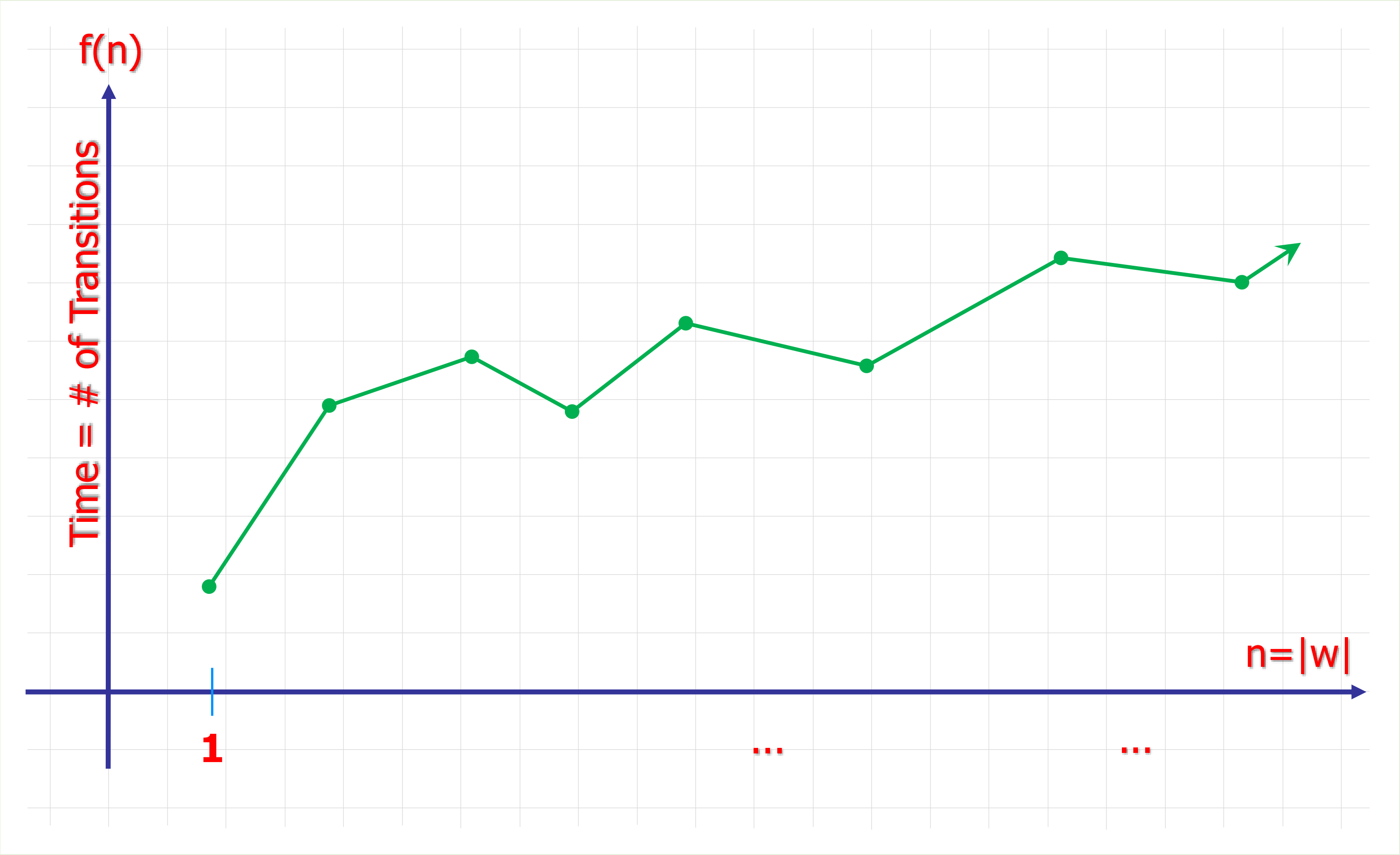
Growth Rate of Resources

- For TMs, the computation time for w's with the same size might vary.
- For example, a TM might have the following values for input size 3:

$$f(3) = \begin{cases} 3 & if\ w = aaa \\ 5 & if\ w = aba \\ 5 & if\ w = baa \\ \dots & \dots \\ 4 & if\ w = bbb \end{cases}$$

- In this case, we pick the worst case that is the longest one, 5, for f(3).
- If we examine for all sizes of w, we'd get a function f(n).
- Next slide shows an example of f(n) for a sample TM.

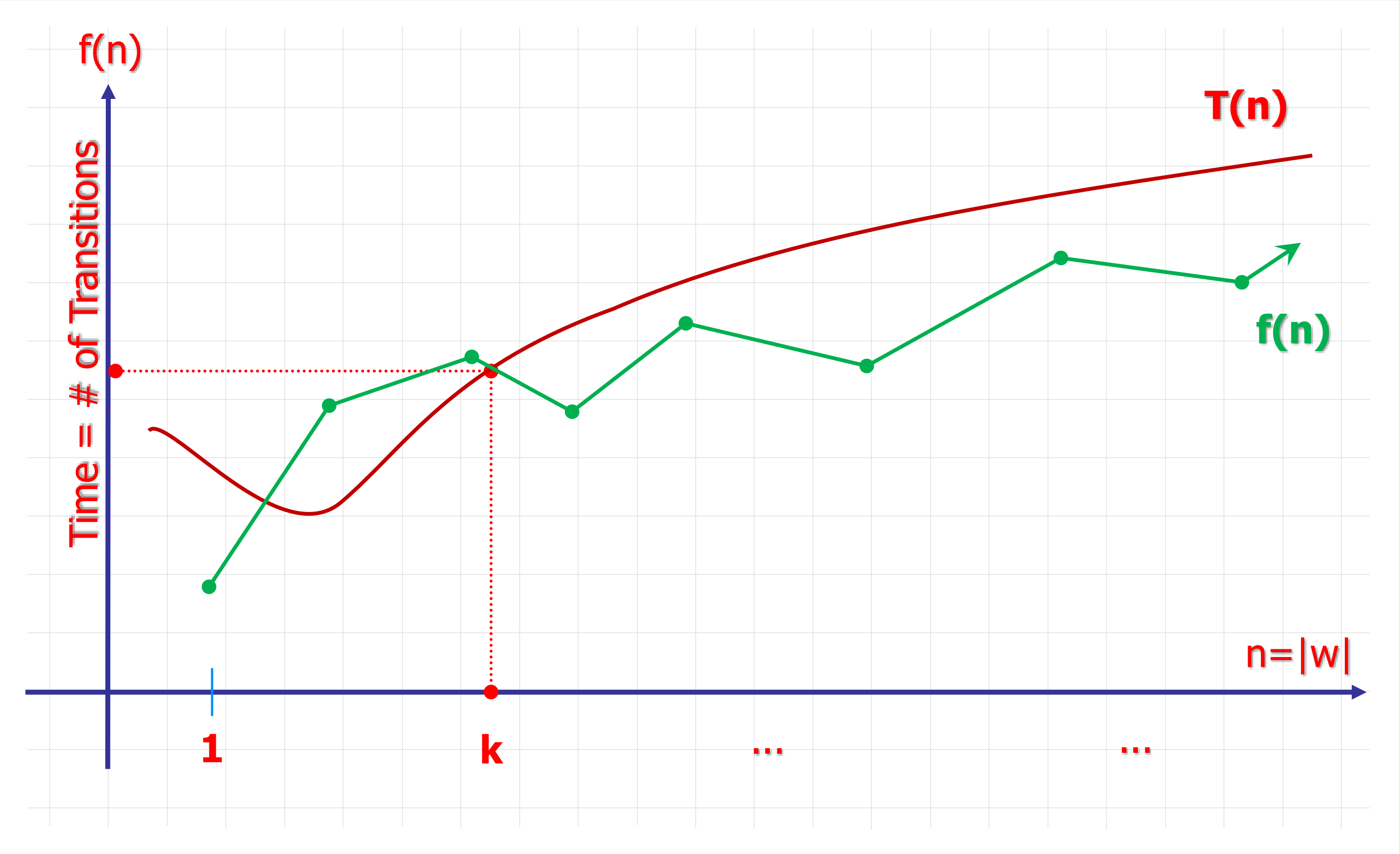
Example of a TM's Computation Time Graph



Example of a TM's Computation Time Graph

- Almost always, the function f(n) is an unknown function.
- With known functions, we mean those that are famous and we know about their behavior. For example:
  - $f(n) = n$
  - $f(n) = n^2$
  - $f(n) = n^3$
  - $f(n) = \log n$
  - $f(n) = 2^n$
  - etc. ...
- But most of times, we can approximate it with a known function as the next slide shows.

Example of a TM's Computation Time Graph



- The approximate function T(n) should have the same growth rate, from a point afterward (e.g. k).

Big-O Notation

Recap

- If we can find such function, then we use a special notation called "Big-O" (aka "Order of magnitude") to represent it.

$$f(n) = O(T(n))$$

  - In math, growth rate of function is also called "order of the function".
- The meaning of the above notation is:

  c * T(n) is an upper-bound for the growth rate of f(n).

  Where c is a positive real number.

- The above equal sign is an "asymptotic notation", not the regular equal sign.
  - That's why it is a very confusing notation.

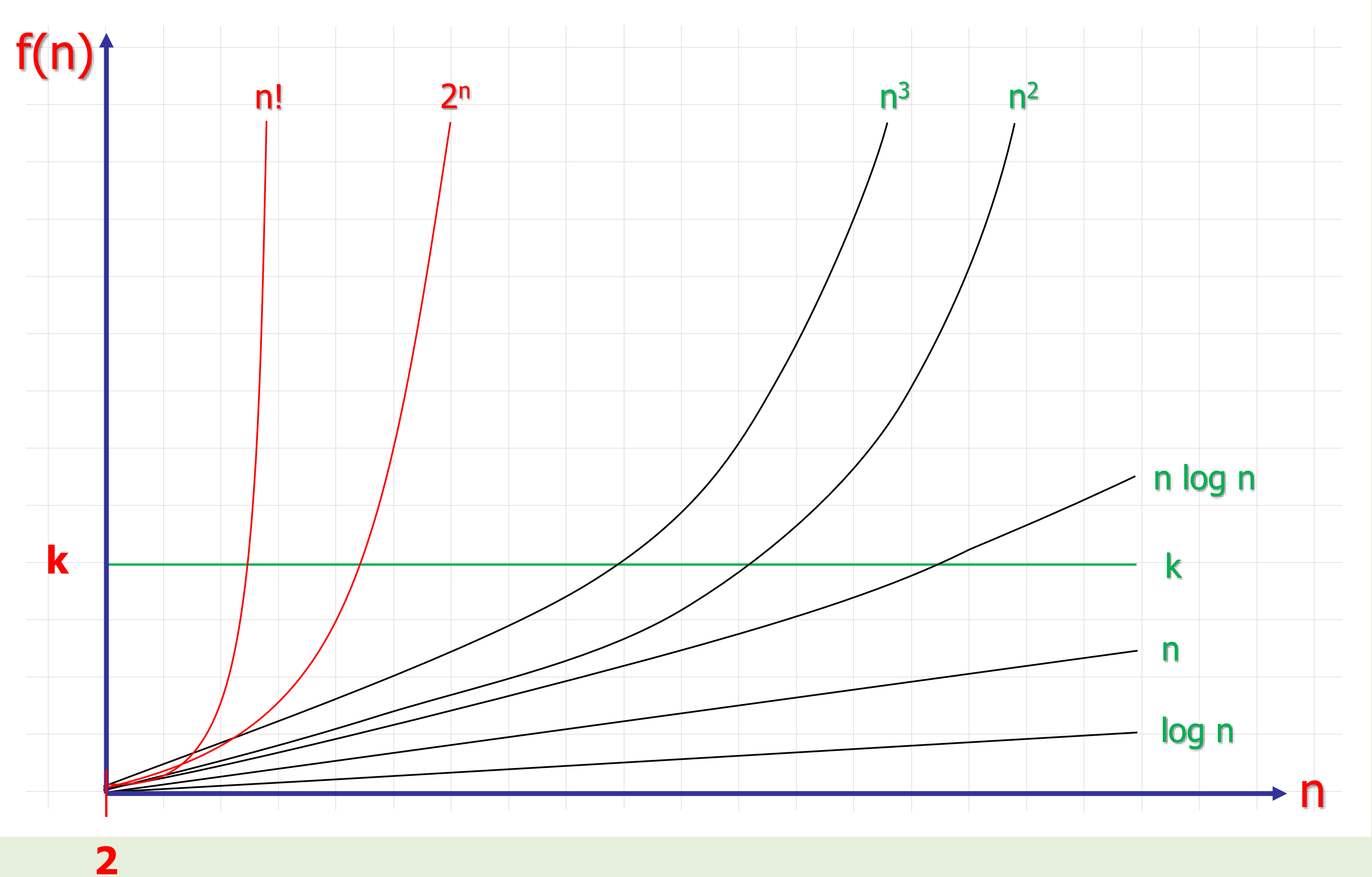
Growth Rate of Some Functions

- The following table shows how different functions grow when the input size grows.

n	k	n	n^2	n^3	2^n
1	k	1	1	1	2
2	k	2	4	8	4
3	k	3	9	27	8
…	…	…	…	…	…
10	k	10	100	1000	1024
…	…	…	…	…	…
100	k	100	10,000	1,000,000	2^{100} = ???

- Next slide shows the graphs of some known functions.

Growth Rate of Some Functions



Computation Complexity Comparison

- To be able to compare complexities, we need to quantify them.
- In computer science, it's been proven that Big-O is the best notation to quantify complexities.

Example 1

- Problem A needs $O(n^2)$ resources.
- Problem B needs $O(n)$ resources.
- Which problem is more complex?

- Problem A ...
- ... because the resource requirement of problem A grows faster than problem B.

Time-Complexity Classes

Time-Complexity Classes

- In this section, we'll classify problems (languages) based on their complexities.

- The goal of this classification is:

  To have an engineering feeling about the types of problems that we encounter.

- To solve problems, we can use standard (deterministic) TM or nondeterministic TM.
  - As we'll see later, there is a huge difference between them.

- Let's start with deterministic TM.

Assumptions

- For the next slides, here are our assumptions:

1. The TMs are single-tape.
2. We are interested in the "worst-cases" because it needs the highest resources.
3. We define ...

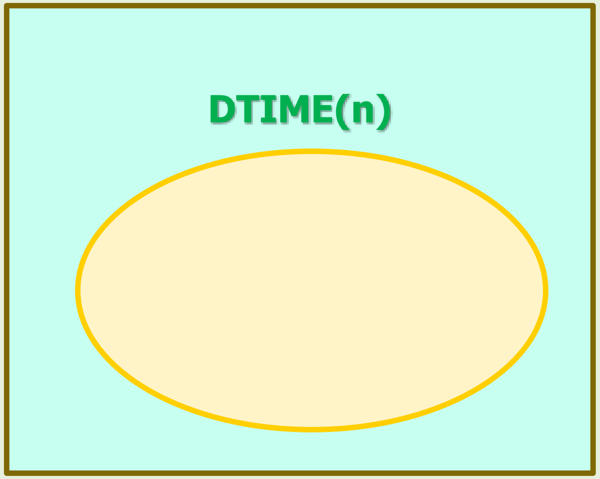
   The complexity of a computation = Efficiency of its algorithm

Complexity Class DTIME(n)

- Our first complexity class is called DTIME(n).

- It contains all problems that can be decided in O(n) time.
  - The "D" at the beginning of DTIME shows that we are using deterministic TMs.
- Let's see what problems we can put in this class.

U = All Formal Languages



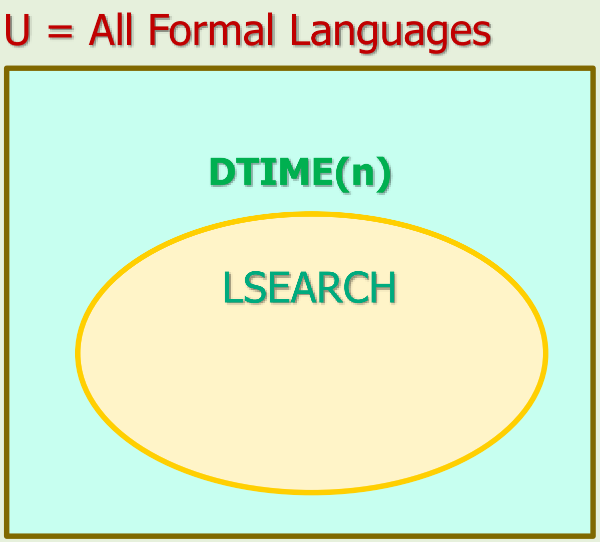
Complexity Class DTIME(n)

Example 2

- Given an unsorted list of numbers $x_1, x_2, ..., x_n$ and a key number k.
- Search in the list and determine if it contains k (LSEARCH).

- In the worst-case, we need n comparisons.
- So, the time-complexity of this problem is O(n).

- Note that we assume each comparison needs constant amount of time k.
- So, total time needed is n * k.
- In big-O notation, we eliminate constants.

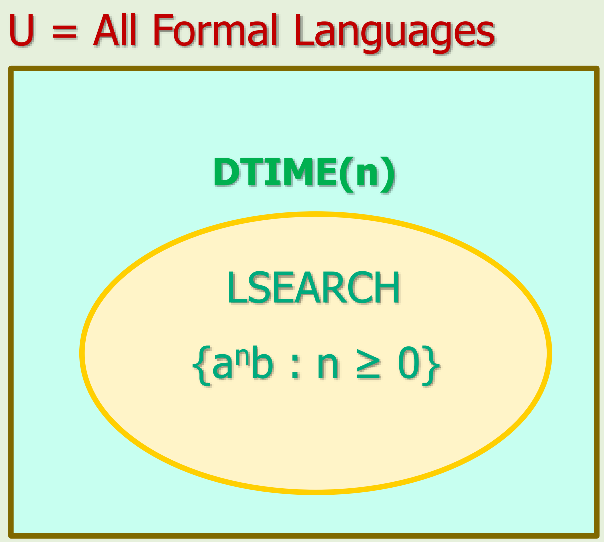


Complexity Class DTIME(n)

Example 3

- Given $L = \{a^n b : n \geq 0\}$
- What is the time-complexity of accepting this language?

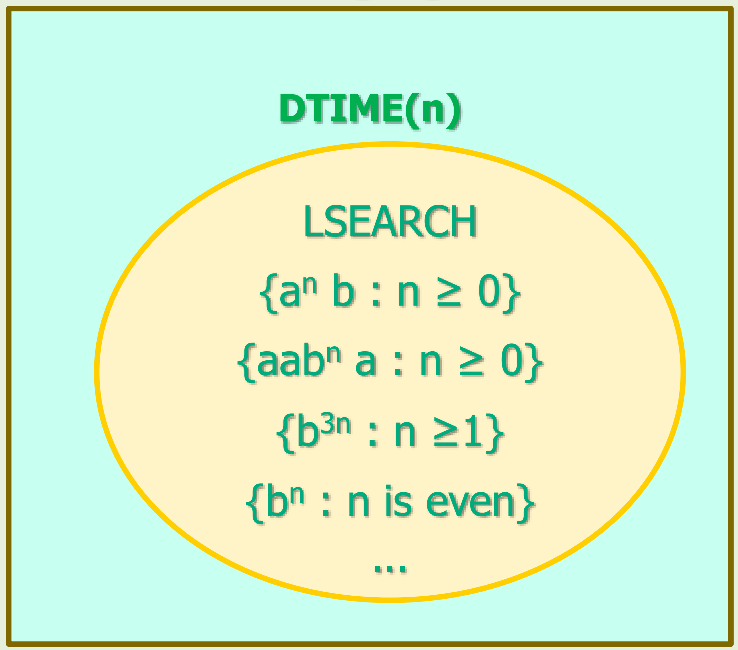
- L can be decided in O(n) by using a deterministic TM.



Complexity Class DTIME(n)

- Also, the following languages can be decided in $O(n)$ by using a deterministic TM.

U = Al Formal Languages



Time-Complexity Classes

- What are the complexities of the following languages?

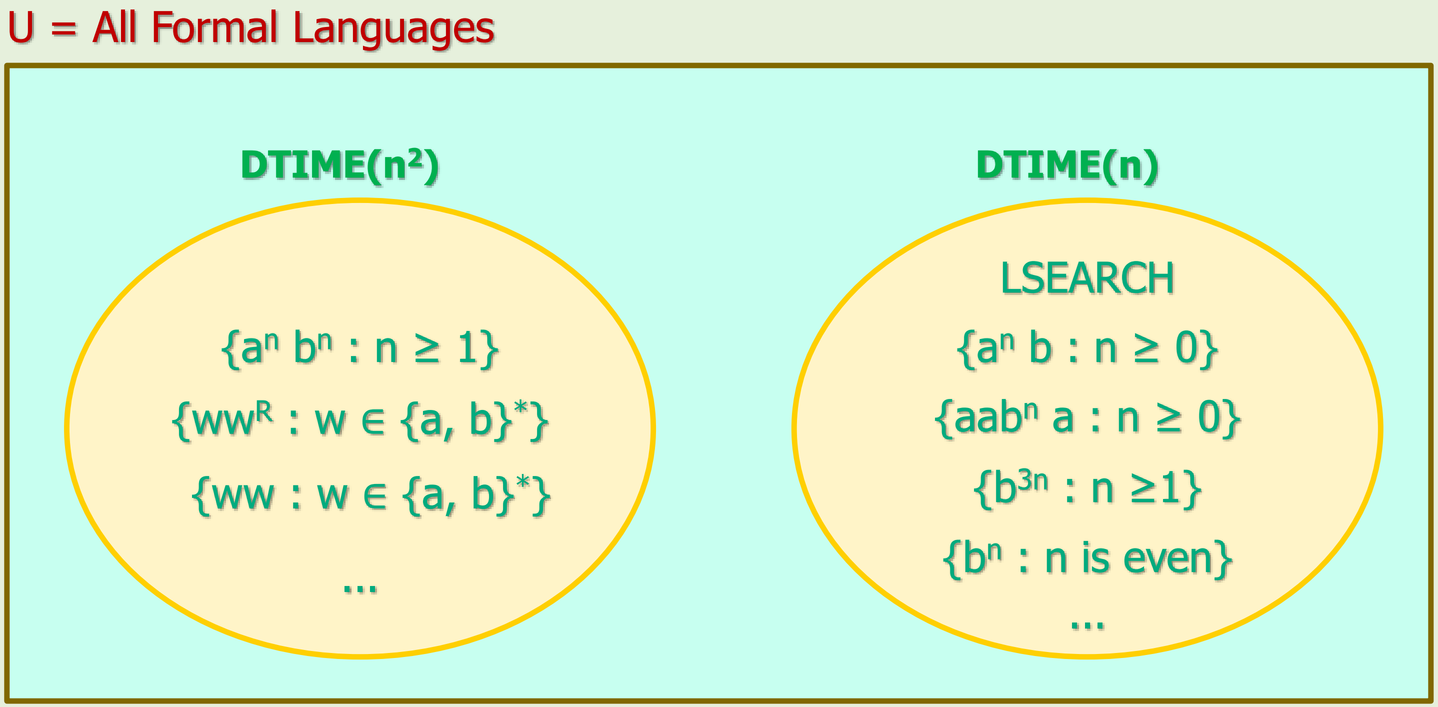
- $L_1 = \{a^n b^n : n \geq 0\}$
- $L_2 = \{ww^R : w \in \{a, b\}^*\}$
- $L_3 = \{ww : w \in \{a, b\}^*\}$

- If we add up all of the back-and-forth of the head that we needed to accept L_1, we get $O(n^2)$.
- For the other languages, it is the same.

- So, we need a new class of complexity.

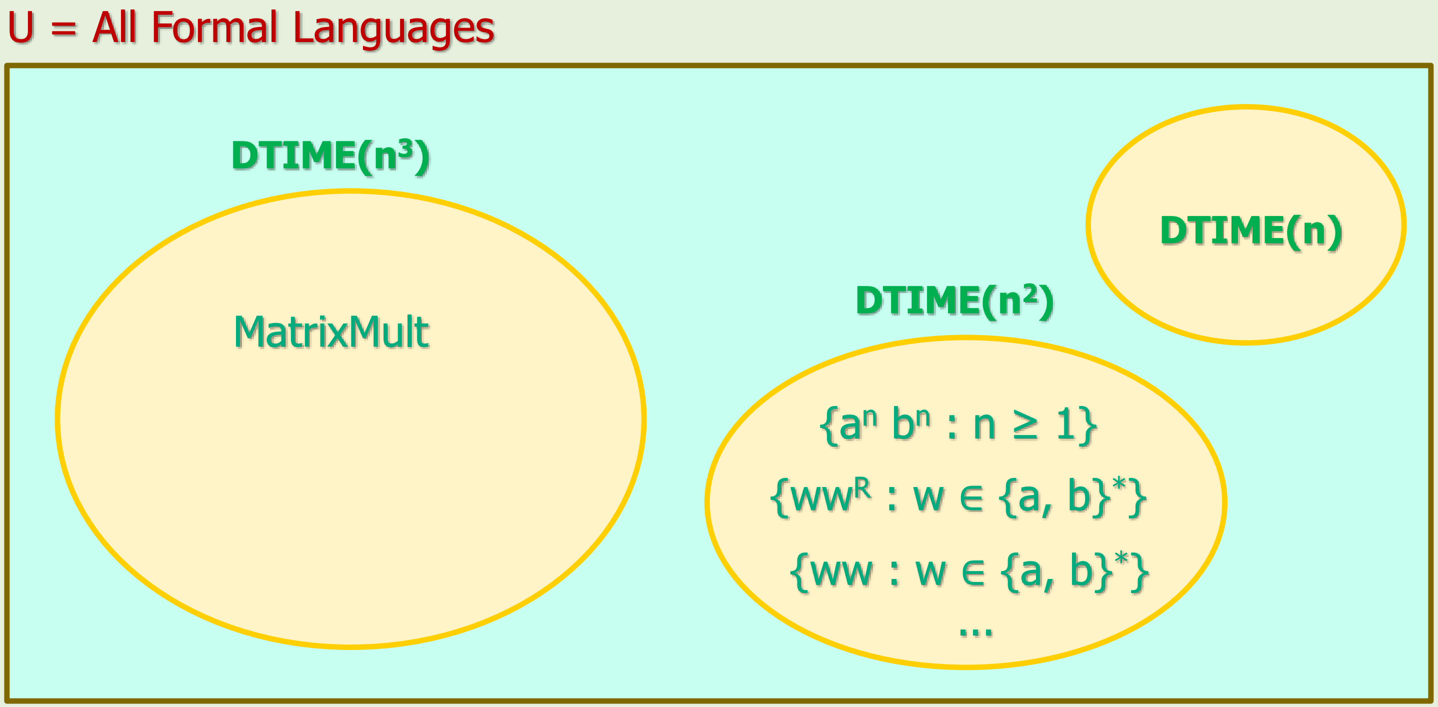
Complexity Class DTIME(n^2)

- We create a new class called DTIME(n^2) and put the languages in the previous slide in this new class.



Complexity Class DTIME(n^3)

- Matrix multiplication problem can be decided in $O(n^3)$ by using a deterministic TM.
- So, we need another class for $O(n^3)$.

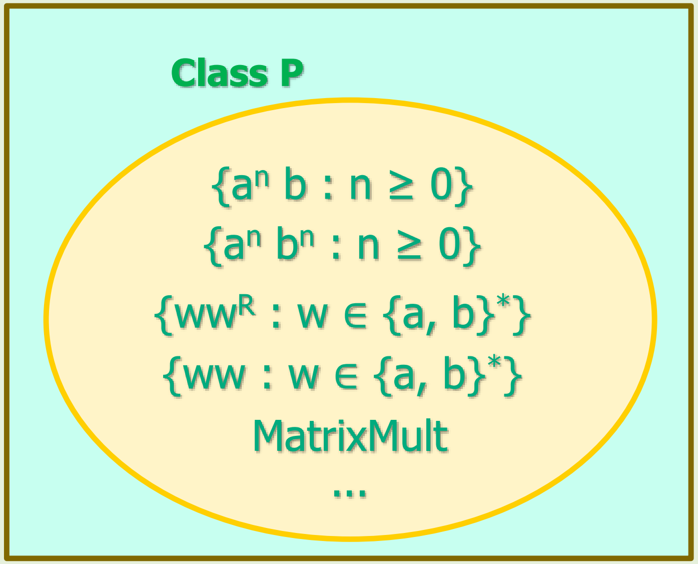


- We can continue this process for $O(n^4)$, $O(n^5)$, ..., $O(n^k)$.

Class P

- Classifying languages based on the degree of n has less practical benefit.
- We define a general class called "polynomial time-complexity" or just "class P".
- Class P is the set of problems that can be decided in polynomial time $O(n^k)$ by using deterministic TMs.
  - Where $k \geq 0$
- These problems are also known as "easy" or "tractable".
- We'll see within a few minutes why they are called "easy"!

U = All Formal Languages



Introduction

- We continue our study about the classification of problems based on their complexities by focusing on "exponential algorithms".
- But first, we need to get familiar with some of those problems.
- In the next slides we'll see some problems that need exponential time to be decided.

Satisfiability Problem (SAT)

- As an example, consider the following logical expression:

$$X = (p \lor r) \land (\sim q \lor \sim r)$$

- For what values of p, q, and r, the expression X is satisfied (= true)?

Solution

- Using "truth table" is the most reliable way to find all solutions.
- The expression has three variable p, q, and r.
- Therefore, there are $2^3 = 8$ rows in the truth table.
- The algorithm should evaluate X for all rows to find all possible solutions.

Satisfiability Problem (SAT)

$X = (p \lor r) \land (\sim q \lor \sim r)$

1. $X = (T \lor T) \land (\sim T \lor \sim T) = F$
2. $X = (T \lor F) \land (\sim T \lor \sim F) = T$
3. $X = (T \lor T) \land (\sim F \lor \sim T) = T$
4. $X = (T \lor F) \land (\sim F \lor \sim F) = T$
5. $X = (F \lor T) \land (\sim T \lor \sim T) = F$
6. $X = (F \lor F) \land (\sim T \lor \sim F) = F$
7. $X = (F \lor T) \land (\sim F \lor \sim T) = T$
8. $X = (F \lor F) \land (\sim F \lor \sim F) = F$

	p	q	r
1	T	T	T
2	T	T	F
3	T	F	T
4	T	F	F
5	F	T	T
6	F	T	F
7	F	F	T
8	F	F	F

Satisfiability Problem (SAT)

- In the previous example, we used an exhaustive algorithm.

Algorithm

- Construct a truth table for n variables.
- Evaluate X for every row.
- Pick those rows that X = true.

- So, theoretically this problem is computable.

Efficiency of Satisfiability Problem (SAT)

- If the number of variables is n, the truth table would have 2^n rows .
- We assume the evaluation of one row needs constant time k.
- Total time required = $k * 2^n$
- But, we ignore the constant coefficients in big-O notation.
- Therefore, the efficiency of SAT problem is $O(2^n)$.

Efficiency of Satisfiability Problem (SAT)

- Is this algorithm practically feasible?

- What would happen if we had 100 variables?
- In this case, we'd need evaluate a table with 2^{100} rows.

- Do you have any idea how big is this number?

- To answer this question, let's "do some math".

Let's Do Some Math!

Example 4: A Practical Calculation for 2^{100}

- Consider a truth table with 100 variables (2^{100} rows).

- If a computer processes each row in 1 Nano sec (10^{-9} sec), how long does it take for this computer to process entire table?

Solution

Let's Do Some Math!

Exhaustive Parsing Algorithm

$S \rightarrow SS \mid a\,S\,b \mid b\,S\,a \mid \lambda$
$w = abba...b;\ |w| = 50$

- Efficiency of exhaustive search parsing algorithm: $O(|P|^{2|w|+1})$

- We have a deterministic computer that can process each substitution in 1 Nano sec (10^{-9} sec).
- How long does it take to parse a string of length 50?

Let's Do Some Math Again!

- Let's take another look at the table of growth rate of functions.
- Compare, for example, one million rows of n^3 and the number that we just calculated for 2^{100}.
- One million rows can be processed within less than a second while 2^{100} needs ....

n	k	n	n^2	n^3	2^n
1	k	1	1	1	2
2	k	2	4	8	4
3	k	3	9	27	8
...	...	...	...	...	...
10	k	10	100	1000	1024
...	...	...	...	...	...
100	k	100	10,000	1,000,000	2^{100} = ???

Using Nondeterministic TMs

Using Nondeterministic TM

Theorem

- If a deterministic TM solves a problem in exponential time $O(k^{an})$, a nondeterministic TM solves it in a polynomial time $O(n^p)$.

- Where p, a, and k are constants.

Using Nondeterministic TM

Example 7

- The SAT problem complexity = $O(2^n)$ (by using deterministic TM)

- If we solve this problem by using a nondeterministic TM, the complexity would be $O(n^p)$ where p is a constant.

- Do the math again!

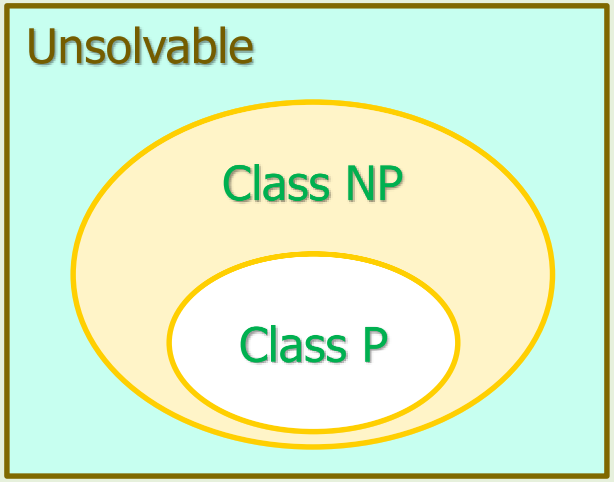
Class NP

- Class NP is the set of problems that can be decided in polynomial time by using nondeterministic TMs.
- NP stands for Nondeterministic Polynomial Time-Complexity
- These problems are also known as "hard" or "intractable".

Relationship between class P and NP

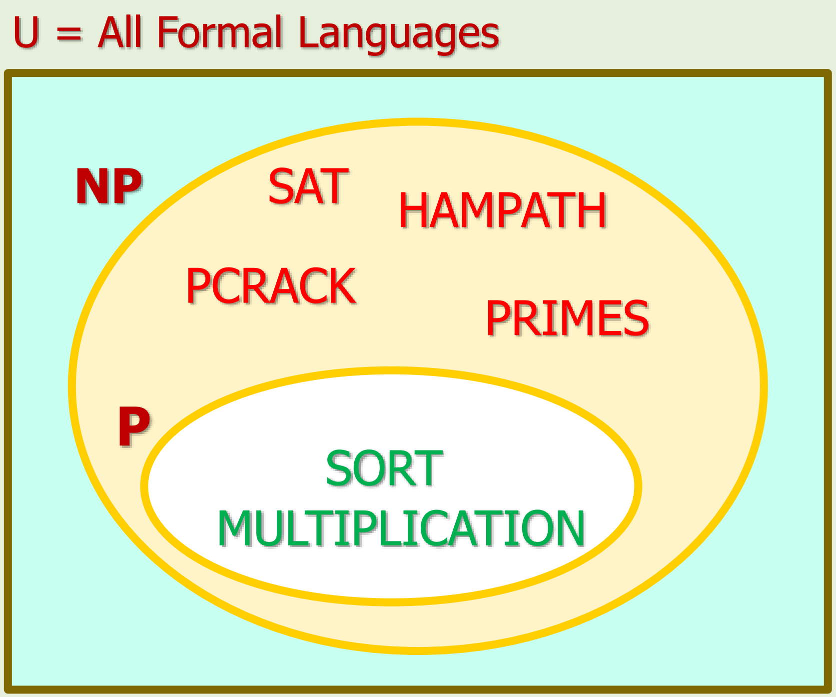
- All problems in class P can also be decided in polynomial time by using nondeterministic TM.
- So, $P \subseteq NP$

U = All Formal Languages



P vs. NP

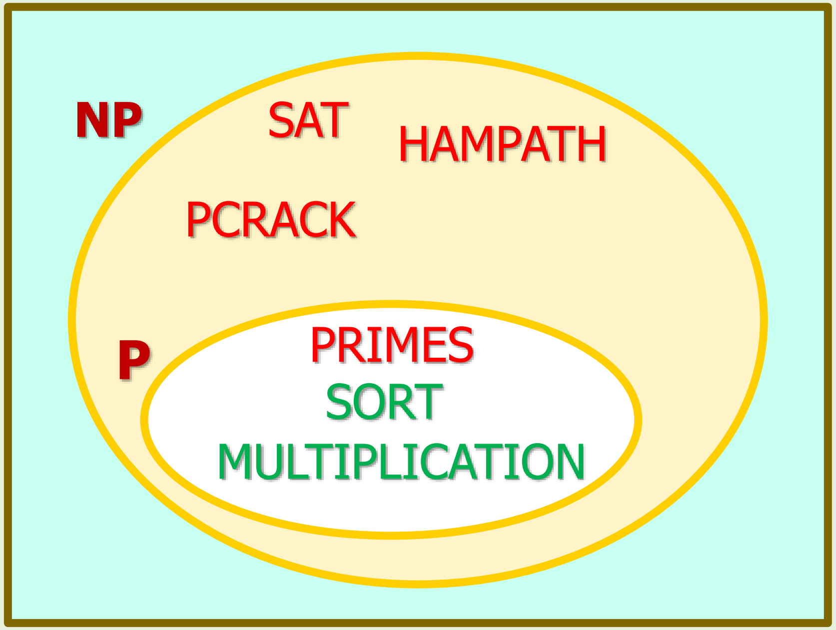
- Computer scientists found polynomial time algorithms for some problems such as sorting, multiplication, etc..
- They found exponential algorithms for some others, such as SAT, HAMPATH (Hamilton path), PRIMES (finding prime numbers), PCRACK (password cracking), ...



P vs. NP

- We were lucky to find a polynomial time algorithm for some of them like PRIMES. (By Agrawal, Kayal, Saxena / 2004) known as AKS alg.
  - Before that, Miller-Rabin algorithm was used that produces probabilistic result. (= Not deterministic algorithm)
- So, we moved PRIMES problem to class P.

U = All Formal Languages



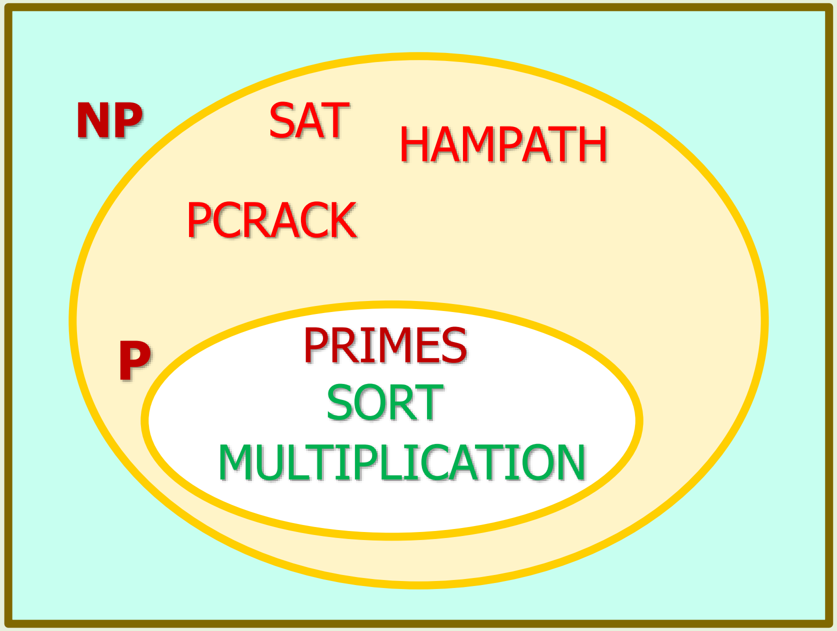
P vs. NP: An Open Question

- Now the question is:

  Can we find polynomial time algorithms for the rest of them?

- In other words, can we expect one day P = NP?
- Nobody's proved "yes" or "no" to this question.
- So, we don't know yet!

U = All Formal Languages



- This is another "open question" of computer science.
- $1,000,000 for the solution!
- http://www.claymath.org

The End

I wish you, all the Bests!

References

1. Linz, Peter, "An Introduction to Formal Languages and Automata, 5th ed.," Jones & Bartlett Learning, LLC, Canada, 2012
2. Michael Sipser, "Introduction to the Theory of Computation, 3rd ed.," CENGAGE Learning, United States, 2013 ISBN-13: 978-1133187790