Complexity Theory Introduction

Charles E. Hughes

COT6410 – Spring 2020 Notes

Who, What, Where and When

- **Instructor: Charles Hughes;**
  HEC-247C
  charles.hughes@ucf.edu
  (e-mail is a good way to get me)
  Use Subject: COT6410
  Office Hours: TR 10:45AM-12:00PM
- **Web Page: http://www.cs.ucf.edu/courses/cot6410/Spring2020**
- **Meetings: TR 1:30PM-2:45PM, HEC-103;**
  28 periods, each 75 minutes long.
  Final Exam (Tuesday, April 21 from 1:00PM to 3:50PM) is separate from class meetings
- **GTA: ???**
  Use Subject: COT6410
  Office Hours: ???

Text Material

- References:
- Cooper, Computability Theory 2nd Ed., Chapman-Hall/CRC Mathematics Series, 2003.
- Garey&Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1979.
- Davis, Sigal&Weyuker, Computability, Complexity and Languages 2nd Ed., Acad. Press (Morgan Kaufmann), 1994.
- Papadimitriou & Lewis, Elements of the Theory of Computation, Prentice-Hall, 1997.
- Bernard Moret, The Theory of Computation, Addison-Wesley, 1998.
- Hopcroft, Motwani&Ullman, Intro to Automata Theory, Languages and Computation 3rd Ed., Prentice-Hall, 2006.
- Oded Goldreich, Computational Complexity: A Conceptual Approach, Cambridge University Press, 2008.
- Draft available at http://www.wisdom.weizmann.ac.il/~/oded/cc-drafts.html
- Oded Goldreich, P, NP, and NP-Completeness: The Basics of Complexity Theory, Cambridge University Press, 2010.
- Draft available at http://www.wisdom.weizmann.ac.il/~/oded/bc-drafts.html
- Arora&Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.
- Draft available at http://www.cs.princeton.edu/theory/complexity/
- Sipser, Introduction to the Theory of Computation 3rd Ed., Cengage Learning, 2013.

Goals of Course

- Introduce Computability and Complexity Theory, including
  - Review background on automata and formal languages
  - Basic notions in theory of computation
    - Algorithms and effective procedures
    - Decision and optimization problems
    - Decision problems have yes/no answer to each instance
  - Limits of computation
    - Turing Machines and other equivalent models
    - Determinism and non-determinism
    - Undecidable problems
    - The technique of reducibility; The ubiquity of undecidability (Rice's Theorem)
    - The notions of semi-decidable (re) and of co-re sets
  - Complexity theory
    - Order notation (quick review)
    - Polynomial reducibility
    - Time complexity, the sets P, NP, co-NP, NP-complete, NP-hard, etc., and the question does P=NP? Sets in NP and NP-Complete.
    - Gadgets and other reduction techniques

Expected Outcomes

- You will gain a solid understanding of various types of computational models and their relations to one another.
- You will have a strong sense of the limits that are imposed by the very nature of computation, and the ubiquity of unsolvable problems throughout CS.
- You will understand the notion of computational complexity and especially of the classes of problems known as P, NP, co-NP, NP-complete and NP-Hard.
- You will (hopefully) come away with stronger formal proof skills and a better appreciation of the importance of discrete mathematics to all aspects of CS.

Keeping Up

- I expect you to visit the course web site regularly (preferably daily) to see if changes have been made or material has been added.
- Attendance is preferred, although I do not take roll.
- I do, however, ask lots of questions in class and give lots of hints about the kinds of questions I will ask on exams. It would be a shame to miss the hints, or to fail to impress me with your insightful in-class answers.
- You are responsible for all material covered in class, whether in the notes or not.

Rules to Abide By

- Do Your Own Work
  - When you turn in an assignment, you are implicitly telling me that these are the fruits of your labor. Do not copy anyone else's homework or let anyone else copy yours. In contrast, working together to understand lecture material and solutions to problems not posed as assignments is encouraged.
- Late Assignments
  - I will accept no late assignments, except under very unusual conditions, and those exceptions must be arranged with me in advance unless associated with some tragic event.
- Exams
  - No communication during exams, except with me or a designated proctor, will be tolerated. A single offense will lead to termination of your participation in the class, and the assignment of a failing grade.

Grading

- Grading of Assignments and Exams
  - I will endeavor to return the midterm exam within a week of its taking place and each assignment within a week of its due date.
- Exam Weights
  - The weights of exams will be adjusted to your personal benefits, as I weigh the exam you do well in more than one in which you do less well.

Important Dates

- Midterm – Thursday, March 5 (tentative)
- Spring Break – March 9 – 14
- Withdraw Deadline – Friday, March 20
- Final – Tues., April 21, 1:00PM – 3:50PM

Evaluation (tentative)

- Mid Term – 125 points ; Final – 200 points
- Assignments – 75 points;
  Paper and Presentation – 75 points
- Extra – 25 points used to increase weight of exams or maybe paper/presentation, always to your benefit
- Total Available: 500 points
- Grading will be A >= 90%, B+ >= 85%,
  B >= 80%, C+ >= 75%, C >= 70%,
  D >= 50%, F < 50% (Minuses might be used)

Decision Problems

- A set of input data items (input "instances" or domain)
- Each input data item defines a question with an answer Yes/No or True/False or 1/0.
- A decision problem can be viewed as a relation between its domain and its binary range
- A decision problem can also be viewed as a partition of the input domain into those that give rise to true instances and those that give rise to false instances.
- In each case, we seek an algorithmic solution (in the form of a predicate) or a proof that none exists
- When an algorithmic solution exists, we seek an efficient algorithm, or proofs of the problem's inherent complexity

Assignment # 1 Includes Financial Aid Related Activity

Complete questionnaire (in quizzes category) on Webcourses.

Complete all questions on time for a few free points out of total points for all assignments.

Complete and submit by one minute before Midnight Friday, 1/10.

UNIVERSE OF DISCOURSE
USUALLY STRINGS OR NATURAL NUMBERS

DECISION PROBLEMS

S
Subset of interest, maybe with ordered elements

For some element, x, is x in S?

Question: How many subsets of Natural Numbers are there? How many languages are there over some finite alphabet?

Example 1: S is set of Primes and x is a natural number; is x in S (is x a prime)?
Example 2: S is an undirected graph (pairs for neighbors); is S 3-colorable?
Example 3: S is a program in C; is S syntactically correct?
Example 4: S is program in C; does S halt on all input?
Example 5: S is a set of strings; is the language S Regular, Context-Free, … ?

Recognizer and Generators

1. When we discuss languages and classes of languages, we discuss recognizers and generators
2. A recognizer for a specific language is a program or computational model that differentiates members from non-members of the given language
3. A portion of the job of a compiler is to check to see if an input is a legitimate member of some specific programming language – we refer to this as a syntactic recognizer
4. A generator for a specific language is a program that generates all and only members of the given language
5. In general, it is not individual languages that interest us, but rather classes of languages that are definable by some specific class of recognizers or generators
6. One type of recognizer is called an automata and there are multiple classes of automata
7. One type of generator is called a grammar and there are multiple classes of grammars
8. Our first journey will be a review of automata and grammars

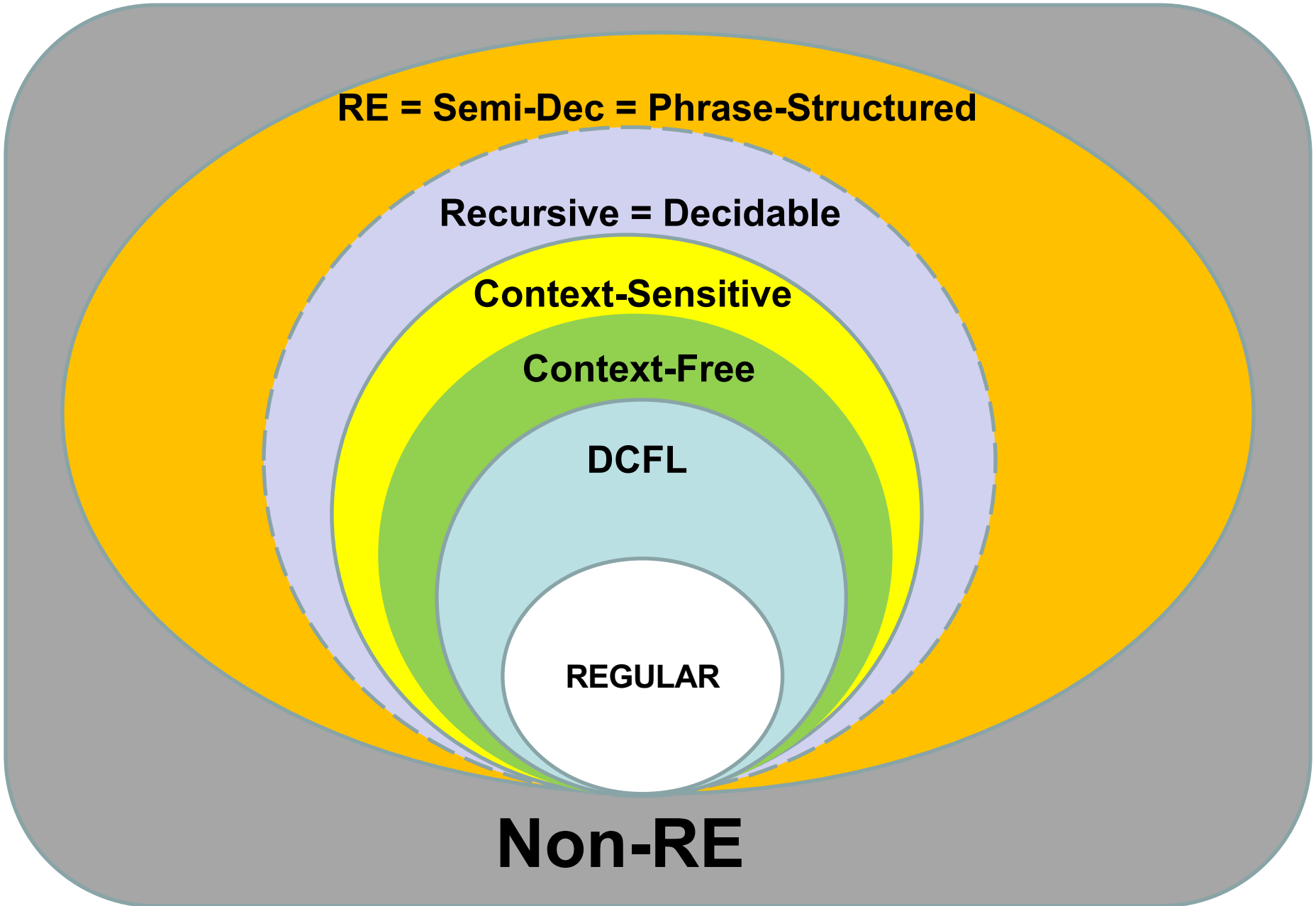
Alphabets and Strings

- DEFINITION 1. An *alphabet* Σ is a finite, non-empty set of abstract symbols.
- DEFINITION 2. Σ^*, the set of all strings over the alphabet, S, is given inductively as follows.
  - Basis: $\lambda \in \Sigma^*$ (the *null string* is denoted by λ, it is the string of length 0, that is $|\lambda| = 0$) [text uses ε but I avoid that as hate saying $\varepsilon \in A$; it's really confusing when manually written] $\forall a \in \Sigma$, $a \in \Sigma^*$ (the members of S are strings of length 1, $|a| = 1$)
  - Induction rule: If $x \in \Sigma^*$, and $a \in \Sigma$, then $a \cdot x \in \Sigma^*$ and $x \cdot a \in \Sigma^*$. Furthermore, $\lambda \cdot x = x \cdot \lambda = x$, and $|a \cdot x| = |x \cdot a| = 1 + |x|$.
  - *NOTE:* "$a \cdot x$" denotes "a *concatenated to* x" and is formed by appending the symbol a to the left end of x. Similarly, $x \cdot a$, denotes appending a to the right end of x. In either case, if x is the null string (λ), then the resultant string is "a".
  - We could have skipped saying $\forall a \in \Sigma$, $a \in \Sigma^*$, as this is covered by the induction step.

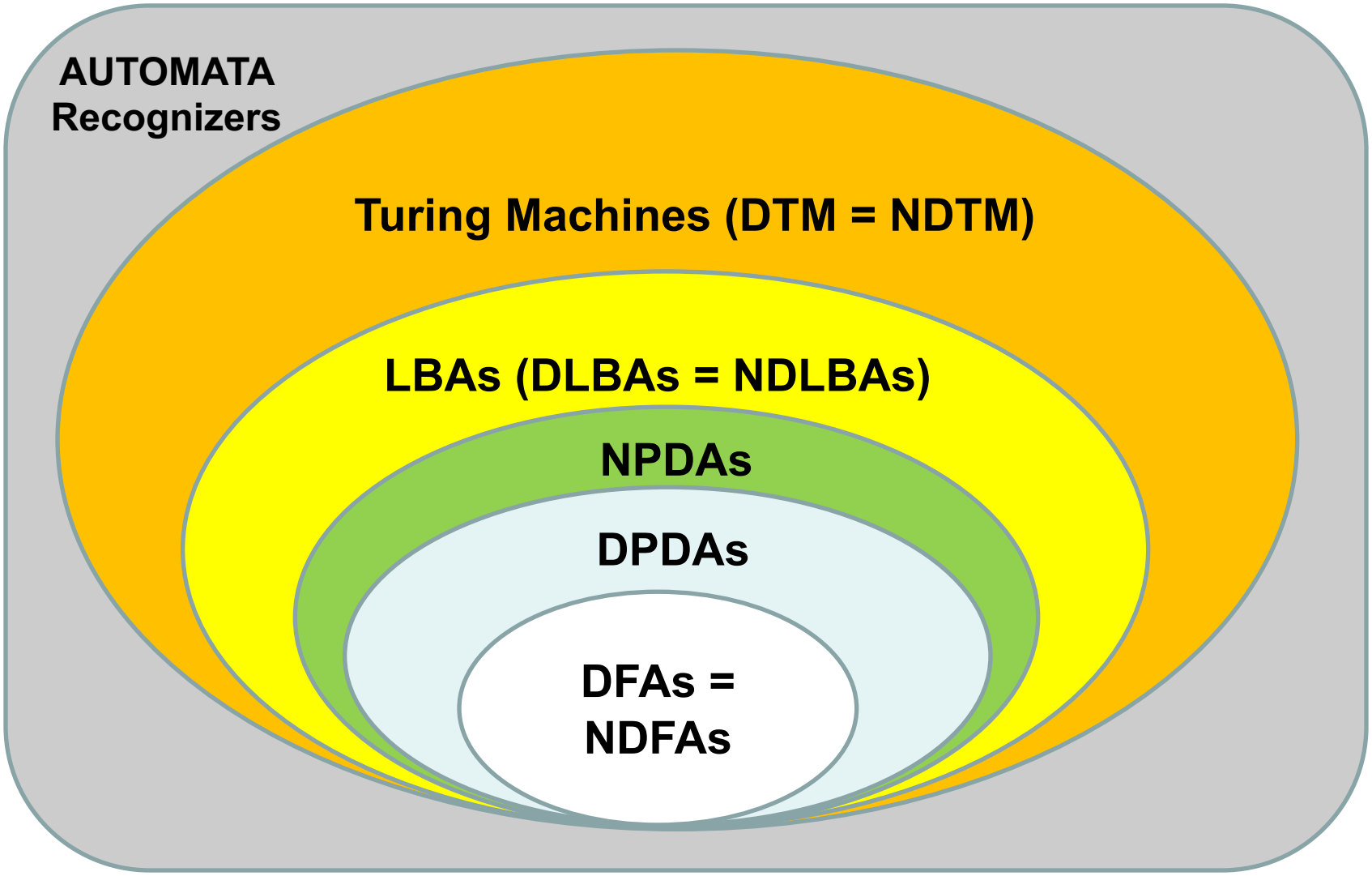
Languages

- DEFINITION 3. Let Σ be an alphabet. A *language over* Σ is a subset, L, of Σ^*.
- Example. Languages over the alphabet Σ = {a, b}.
  - Ø (the empty set) is a language over Σ
  - Σ^* (the universal set) is a language over Σ
  - {a, bb, aba } (a finite subset of Σ^*) is a language over Σ.
  - { ab^na^m | $n = m^2$, n, m ≥ 0 } (infinite subset) is a language over Σ.
- DEFINITION 4. Let L and M be two languages over Σ. Then the *concatenation of L with M*, denoted L·M is the set,
  L·M = { x·y | x $\in$ L and y $\in$ M }
  The concatenation of arbitrary strings x and y is defined inductively as follows.
  Basis: When |x| $\leq$ 1 or |y| $\leq$ 1, then x·y is defined as in Definition 2.
  Inductive rule: when |x| > 1 and |y| > 1, then x = x' · a for some a $\in \Sigma$ and x' $\in \Sigma^*$, where |x'| = |x|-1. Then x·y = x'·(a·y).

UNIVERSE OF LANGUAGES

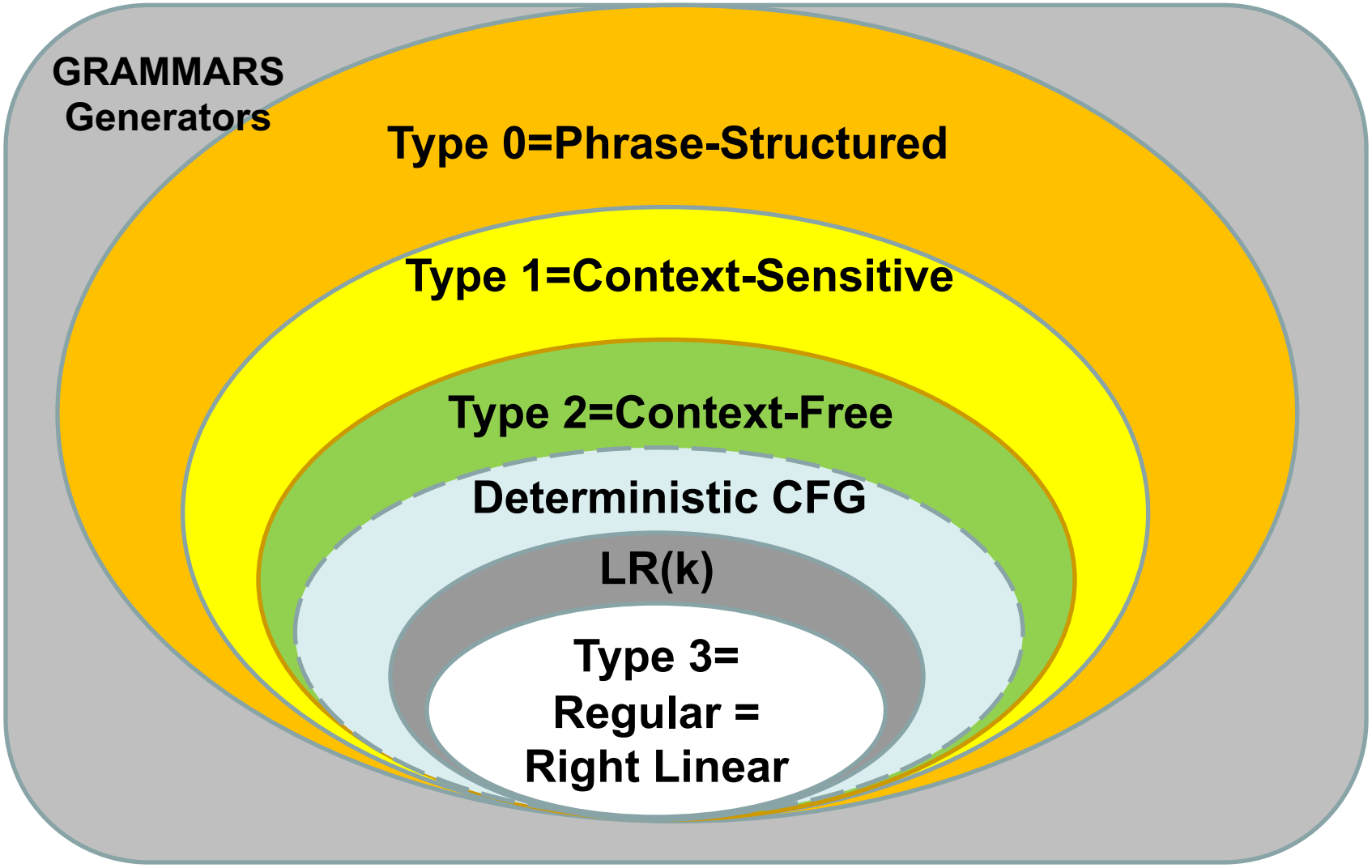


MODELS OF COMPUTATION



Of these models, only TMs can do general computation

REWRITING SYSTEMS



What We are Studying

Computability Theory

The study of what can/cannot be done via purely computational means.

Complexity Theory

The study of what can/cannot be done well via purely computational means.

Graph Coloring

- Instance: A graph **G = (V, E)** and an integer **k**.
- Question: Can **G** be "properly colored" with at most **k** colors?

- Proper Coloring: a color is assigned to each vertex so that adjacent vertices have different colors.

- Suppose we have two instances of this problem (1) is True (Yes) and the other (2) is False (No).

- AND, you know (1) is Yes and (2) is No. (Maybe you have a secret program that has analyzed the two instance.)

Checking a “Yes” Answer

- Without showing how your program works (you may not even know), how can you convince someone else that instance (1) is, in fact, a Yes instance?
- We can assume the output of the program was an actual coloring of **G**. Just give that to a doubter who can easily check that no adjacent vertices are colored the same, and that no more than **k** colors were used.
- How about the No instance?
- What could the program have given that allows us to quickly "verify" (2) is a No instance?
    - No One Knows!!
- For all seems to be harder than there exists in many contexts

Checking a "No" Answer

- The only thing anyone has thought of is to have it test all possible ways to **k**-color the graph – all of which fail, of course, if "No" is the correct answer.
- There are an exponential number of things (colorings) to check.
- For some problems, there seems to be a big difference between verifying Yes and No instances.
- To solve a problem efficiently, we must be able to solve both Yes and No instances efficiently.

Hard and Easy

- True Conjecture: If a problem is easy to solve, then it is easy to verify (just solve it and compare).
- Contrapositive: If a problem is hard to verify, then it is (probably) hard to solve.
- There is nothing magical about Yes and No instances – sometimes the Yes instances are hard to verify and No instances are easy to verify.
- And, of course, sometimes both are hard to verify.

Easy Verification

- Are there problems in which both Yes and No instances are easy to verify?

- Yes. For example: Search a list **L** of **n** values for a key **x**.
- Question: Is **x** in the list **L**?

- Yes and No instances are both easy to verify.

- In fact, the entire problem is easy to solve!!

Verify vs Solve

- Conjecture: If both Yes and No instances are easy to verify, then the problem is easy to solve.
- No one has yet proven this claim, but most researchers believe it to be true.
- Note: It is usually relatively easy to prove something is easy – just write an algorithm for it and prove it is correct and that it is fast (usually, we mean polynomial).
- But, it is usually very difficult to prove something is hard – we may not be clever enough yet. So, you will often see "appears to be hard."

Instances vs Problems

- Each instance has an *'answer*.‘
  - An instance’s answer is the solution of the instance - it is *not* the solution of the problem.
  - A solution of the problem is a computational procedure that finds the answer of any instance given to it – the procedure must halt on all instances – it must be an *'algorithm*.'

Three Classes of Problems

Problems are often classified in one of three groups (classes):

Undecidable (impossible), Exponential (hard), and Polynomial (easy).

Theoretically, all problems belong to exactly one of these three classes and our job is often to find which one.

Why do we Care?

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution and, better yet, the lowest degree polynomial solution.

You should know something about how hard a problem is before you try to solve it.

Effective Procedure

- *A process whose execution is clearly specified to the smallest detail*
- Such procedures have, among other properties, the following:
  - Processes must be finitely describable, and the language used to describe them must be over a finite alphabet.
  - The current state of the machine model must be finitely presentable.
  - Given the current state, the choice of actions (steps) to move to the next state must be easily determinable from the procedure's description.
  - Each action (step) of the process must be capable of being carried out in a finite amount of time.
  - The semantics associated with each step must be clear and unambiguous.

Algorithm

- *An effective procedure that halts on all input*
- The key term here is "*halts on all input*"
- By contrast, an effective procedure may halt on all, none or some of its input.
- The *domain* of an algorithm is its entire universe of possible inputs
- The *domain* of a procedure is the inputs on which it converges (stops).

Sample Algorithm/Procedure

{ Example algorithm:

Linear search of a finite list for a key;

If key is found, answer “Yes”;

If key is not found, answer “No”; }

{ Example procedure:

Linear search of a finite list for a key;

If key is found, answer “Yes”;

If key is not found, try this strategy again; }

Note: Latter is not unreasonable if the list can be increased in size by some properly synchronized concurrent thread.

Procedure vs Algorithm

Looking back at our approaches to “find a key in a finite list,” we see that the algorithm always halts and always reports the correct answer. In contrast, the procedure does not halt in some cases, but never lies.

What this illustrates is the essential distinction between an algorithm and a procedure – algorithms always halt in some finite number of steps, whereas procedures may run on forever for certain inputs. A particularly silly procedure that never lies is a program that never halts for any input.

Notion of Solvable

- A problem is *solvable* if there exists an algorithm that solves it (provides the correct answer for each instance).
- The fact that a problem is solvable or, equivalently, *decidable* or, equivalently, *recursive* does not mean it is *solved*. To be solved, someone must have produced a correct algorithm.
- The distinction between solvable and solved is subtle. Solvable is an innate property – an unsolvable problem can never become solved, but a solvable one may or may not be solved in an individual's lifetime.

An Old Solvable Problem

Does there exist a set of positive whole numbers, a, b, c and an n>2 such that $a^n+b^n = c^n$?

In 1637, the French mathematician, Pierre de Fermat, claimed that the answer to this question is “No”. This was called Fermat’s Last Theorem, even though he never produced a proof of its correctness.

While this problem remained *unsolved* until Fermat’s claim was verified in 1995 by Andrew Wiles, the problem was always *solvable*, since it had just one question, so the solution was either “Yes” or “No”, and an algorithm *exists* for each of these candidate solutions.

Research Territory

Decidable – vs – Undecidable

(area of Computability Theory)

Exponential – vs – polynomial

(area of Computational Complexity)

For "easy" problems, we want to determine lower and upper bounds on complexity and develop best Algorithms

(area of Algorithm Design/Analysis)

A CS Grand Challenge

Does *P=NP*?

There are many equivalent ways to describe *P* and *NP*. For now, we will use the following.

P is the set of decision problems (those whose instances have "Yes"/ "No" answers) that can be solved in polynomial time on a deterministic computer (no concurrency or guesses allowed).

NP is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn an unbounded number of parallel threads; equivalently one that can be verified in polynomial time on a deterministic computer).

Again, as "Does ***P=NP?***" has just one question, it is solvable, we just don't yet know which solution, "Yes" or "No", is the correct one.

Computability vs Complexity

Computability focuses on the distinction between solvable and unsolvable problems, providing tools that may be used to identify unsolvable problems – ones that can never be solved by mechanical (computational) means. Interestingly, unsolvable problems are everywhere as you will see.

In contrast, complexity theory focuses on how hard it is to solve problems that are known to be solvable. Hard solvable problems abound in the real world. We will address computability theory for the first part of this course, returning to complexity theory later in the semester.

HISTORY

The Quest for Mechanizing Mathematics

Hilbert, Russell and Whitehead

- Until 1800's there were no formal systems to reason about mathematical properties
- Major advances in late 1800's/early 1900's
- Axiomatic schemes
  - Axioms plus sound rules of inference
  - Much of focus on number theory
- First Order Predicate Calculus
  - $\forall \mathbf{x} \exists \mathbf{y}\ [\mathbf{y} > \mathbf{x}]$
- Second Order (Peano's Axiom)
  - $\forall \mathbf{P}\ [[\mathbf{P(0)}\ \&\&\ \forall \mathbf{x}[\mathbf{P(x)} \Rightarrow \mathbf{P(x+1)}]] \Rightarrow \forall \mathbf{x}\mathbf{P}(\mathbf{x})]$

Hilbert

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math's complexity.
- Hilbert's Tenth asks for an algorithm to find the integral zeros of polynomial equations with integral coefficients. This is now known to be impossible (In 1970, Matiyacevič showed this undecidable).
- Davis based on prior work by Julia Robinson, him and Hilary Putnam provided more readable proof in 1972.

Hilbert's Belief

- All mathematics could be developed within a formal system that allowed the mechanical creation and checking of proofs.

Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.

Turing (Post, Church, Kleene)

- In 1936, each presented a formalism for computability.
  - Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.
  - Church developed the notion of lambda-computability from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model.
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
- Church's notation was the lambda calculus, which later gave birth to Lisp.

More on Emil Post

- In the 1920's, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus (all tautologies can be deduced from a finite set of tautologies and a finite set of rules of inference – substitution and modus ponens).
- In the late 1930's and the 1940's, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky's grammars). He showed their equivalence to Turing machines.
- In 1940s, Post showed the complexity (undecidability) of determining what is derivable from an arbitrary set of propositional axioms.