MODULE V

1. **Chomsky Hierarchy Languages**
2. **Turing Reducibility**
3. **The Class P**

1. Chomsky Hierarchy of Languages

- A containment hierarchy (strictly nested sets) of classes of formal grammars

The Hierarchy

Class	Grammars	Languages	Automaton
Type-0	Unrestricted	Recursively enumerable (Turing-recognizable)	Turing machine
	none	Recursive (Turing-decidable)	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

Type 0 Unrestricted:

Languages defined by Type-0 grammars are accepted by Turing machines .

Rules are of the form: $\alpha \rightarrow \beta$, where α and β are arbitrary strings over a vocabulary V and $\alpha \neq \varepsilon$

Type 1 Context-sensitive:

Languages defined by Type-1 grammars are accepted by linear-bounded automata.

Syntax of some natural languages (Germanic)

Rules are of the form:

$\alpha A\beta \rightarrow \alpha B\beta$

$S \rightarrow \varepsilon$

where

Type 2 Context-free:

Languages defined by Type-2 grammars are accepted by push-down automata.

Natural language is almost entirely definable by type-2 tree structures

Rules are of the form:

$A \rightarrow \alpha$

Where

$A \in N$

$\alpha \in (N \cup \Sigma)*$

Type 3 Regular:

Languages defined by Type-3 grammars are accepted by finite state automata

Most syntax of some informal spoken dialog

Rules are of the form:

$A \rightarrow \varepsilon$

$A \rightarrow \alpha$

$A \rightarrow \alpha B$

where

$A, B \in N$ and $\alpha \in \Sigma$

The Universal Turing Machine

- If Tm's are so damned powerful, can't we build one that simulates the behavior of any Tm on any tape that it is given?

- Yes. This machine is called the ***Universal Turing machine***.
- How would we build a Universal Turing machine?
  - We place an encoding of any Turing machine on the input tape of the Universal Tm.
  - The tape consists entirely of zeros and ones (and, of course, blanks)
  - Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
- Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
- Instructions are separated by **00**.
- We use unary notation to represent components of an instruction, with
  - 0 = **1**,
  - 1 = **11**,
  - 2 = **111**,
  - 3 = **1111**,
  - ***n*** = **111...111** (***n***+1 1's).
- We encode q_n as $\underline{n+1}$ 1's
- We encode symbol a_n as $n + 1$ 1's
- We encode move left as 1, and move right as 11

1111011101111101110100101101101101100

q_3, a_2, q_4, a_2, L q_0, a_1, q_1, a_1, R

- Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
- Let T_n be the Turing machine whose encoding is the number n.

2. Turing Reducibility

- A language A is Turing reducible to a language B, written $A \leq_T B$, if A is decidable relative to B
- Below it is shown that E_{TM} is Turing reducible to EQ_{TM}
- Whenever A is mapping reducible to B, then A is Turing reducible to B
  - The function in the mapping reducibility could be replaced by an oracle
- An oracle Turing machine with an oracle for EQ_{TM} can decide E_{TM}

$T^{EQ\text{-}TM}$ = "On input <M>

1. Create TM M_1 such that $L(M_1) = \varnothing$

   M_1 has a transition from start state to reject state for every element of Σ

1. Call the EQ_{TM} oracle on input $\langle M, M_2\rangle$
2. If it accepts, accept; if it rejects, reject"

- $T^{EQ\text{-}TM}$ decides E_{TM}
- E_{TM} is decidable relative to EQ_{TM}
- **Applications**
  - If $A \leq_T B$ and B is decidable, then A is decidable
  - If $A \leq_T B$ and A is undecidable, then B is undecidable
  - If $A \leq_T B$ and B is Turing-recognizable, then A is Turing-recognizable
  - If $A \leq_T B$ and A is non-Turing-recognizable, then B is non-Turing-recognizable

3. The class P

A decision problem *D* is *solvable in polynomial time* or *in the class P,* if there exists an algorithm *A* such that

- *A Takes instances* of *D* as inputs.
- *A* always outputs the correct answer "Yes" or "No".
- There exists a polynomial *p* such that the execution of *A* on inputs of size *n* always terminates in *p(n)* or fewer steps.
- **EXAMPLE**: The Minimum Spanning Tree Problem is in the class P.

  The class *P* is often considered as synonymous with the class of computationally feasible problems, although in practice this is somewhat unrealistic.

The class NP

A decision problem is *nondeterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm *A* such that

- *A* takes as inputs potential witnesses for "yes" answers to problem *D.*
- *A* correctly distinguishes true witnesses from false witnesses.

- There exists a polynomial *p* such that for each potential witnesses of each instance of size *n* of *D,* the execution of the algorithm *A* takes at most *p(n)* steps.
- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - If a solution exists, computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds it has a working solution
  - So: **NP** = problems *verifiable* in polynomial time
  - Unknown whether **P = NP** (most suspect not)

NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time.
  - Then *every* NP-Complete problem can be solved in polynomial time.
  - And in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve hamiltonian-cycle in O(n^{100}) time, you've proved that **P = NP**. Retire rich & famous.
- The crux of NP-Completeness is *reducibility*
  - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
    - *What do you suppose "easily" means?*
    - This rephrasing is called *transformation*
  - Intuitively: If P reduces to Q, P is "no harder to solve" than Q
- An example:
  - P: Given a set of Booleans, is at least one TRUE?
  - Q: Given a set of integers, is their sum positive?

- Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if x_i = TRUE, $y_i = 0$ if x_i = FALSE

- Another example:
  - Solving linear equations is reducible to solving quadratic equations
    - *How can we easily use a quadratic-equation solver to solve linear equations?*

- Given one NP-Complete problem, we can prove many interesting problems NP-Complete
  - Graph coloring (= register allocation)
  - Hamiltonian cycle
  - Hamiltonian path
  - Knapsack problem
  - Traveling salesman
  - Job scheduling with penalties, etc.

NP Hard

- **Definition:** Optimization problems whose decision versions are NP- complete are called *NP-hard.*

- **Theorem:** If there exists a polynomial-time algorithm for finding the optimum in any *NP*-hard problem, then $P = NP$.
  Proof: Let *E* be an *NP*-hard optimization (let us say minimization) problem, and let *A* be a polynomial-time algorithm for solving it. Now an instance *J* of the corresponding decision problem *D* is of the form *(I, c),* where *I* is an instance of *E,* and *c* is a number. Then the answer to *D* for instance *J* can be obtained by running *A* on *I* and checking whether the cost of the optimal solution exceeds *c.* Thus there exists a polynomial-time algorithm for *D,* and *NP*-completeness of the latter implies $P = NP$.