CS351, Mock
Introduction – Automata: The Methods and the Madness

What is this course about? Automata theory is the study of abstract computing devices, or "machines." This topic goes back to the days before digital computers and describes *what is possible to compute using an abstract machine*. Much of this work was completed by Alan Turing, and we will describe the abstract Turing Machine later in the course. Why is this useful? These ideas directly apply to creating compilers, programming languages, and designing applications. They also provide a formal framework to analyze new types of computing devices, e.g. biocomputers or quantum computers. Finally, the course should help to turn you into mathematically mature computer scientists capable of precise and formal reasoning.

More precisely, we'll focus primarily on the following topics. Don't worry about what all the terms mean yet, we'll cover the definitions as we go:

1. Finite state automata: Deterministic and non-deterministic finite state machines; regular expressions and languages. Techniques for identifying and describing regular languages; techniques for showing that a language is not regular. Properties of such languages.

2. Context-free languages: Context-free grammars, parse trees, derivations and ambiguity. Relation to pushdown automata. Properties of such languages and techniques for showing that a language is not context-free.

3. Turing Machines: Basic definitions and relation to the notion of an algorithm or program. Power of Turing Machines.

4. Undecidability: Recursive and recursively enumerable languages. Universal Turing Machines. Limitations on our ability to compute; undecidable problems.

5. Computational Complexity: Decidable problems for which no sufficient algorithms are known. Polynomial time computability. The notion of NP-completeness and problem reductions. Examples of hard problems.

Let's start with a big-picture overview of these five topics and hopefully give you some motivation of why we should study them and where knowledge of these topics can be useful.

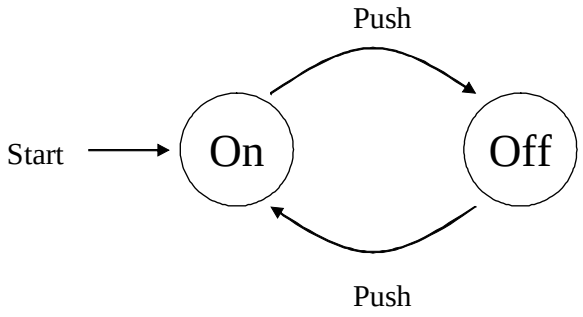
Finite State Automata

Finite automata serve as a useful model for hardware, software, algorithms, and processes. Automata is the plural of "automaton", i.e. a robot, so a finite state automata is essentially a "robot composed of a finite number of states." Here are a few examples where finite automata can be used:

- Software to design and verify circuit behavior
- Lexical analyzer of a typical compiler
- Parser for natural language processing
- An efficient scanner for patterns in large bodies of text (e.g. text search on the web)
- Verification of protocols (e.g. communications, security).

We'll precisely define an automata later, but informally a finite state automata is a finite enumerated list of states, with transitions between those states. The advantage of a finite number of states is that the model can be implemented in hardware as a circuit or perhaps written as a program.

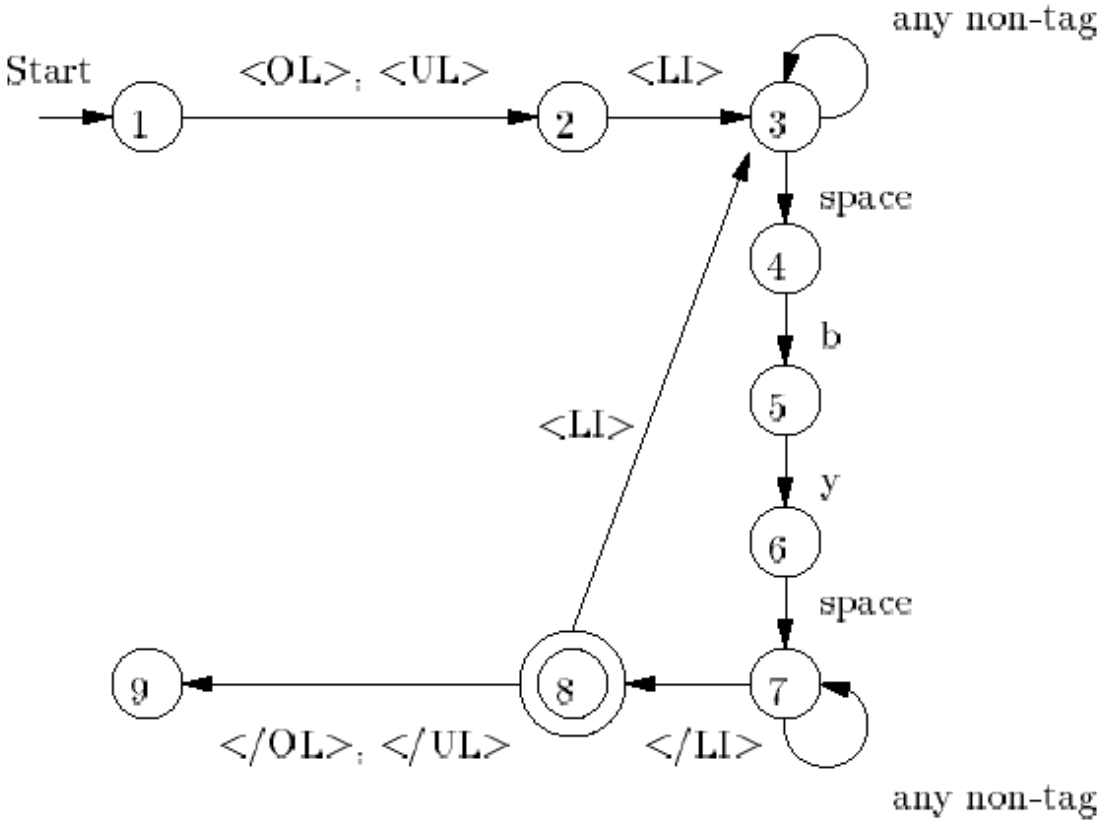
Here is perhaps one of the simplest finite automaton, an on-off switch:



States are represented by circles. In this case, the states are named "on" and "off" but generally we will use much more generic names for states (e.g. q1, q2). Edges or arcs between states indicate transitions or inputs to the system. The "start" edge indicates which state we start in. Initially the switch is on. When the switch encounters a "push" we move to the Off state.. When we encounter another push we now move to the On state.

Sometimes it is necessary to indicate a "final" or "accepting" state. We'll do this by drawing the state in double circles, as shown in the next example below.

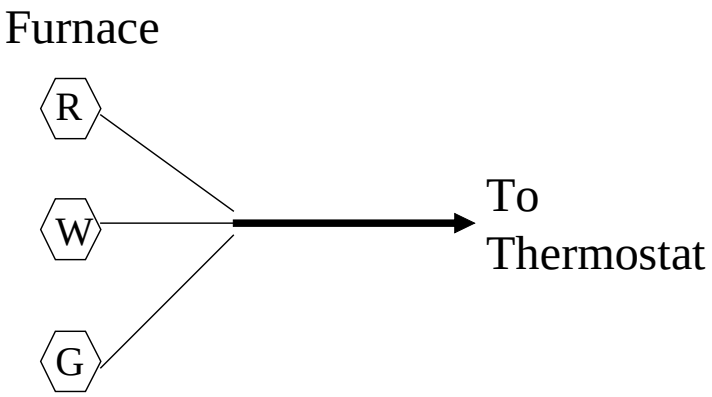
Consider an automaton to parse an HTML document that attempts to identify title-author pairs in a bulleted or ordered list. This might be useful to generate a reading list of some sort automatically. A hypothetical automaton to address this task is shown below that scans for the letters "by" inside a list item:



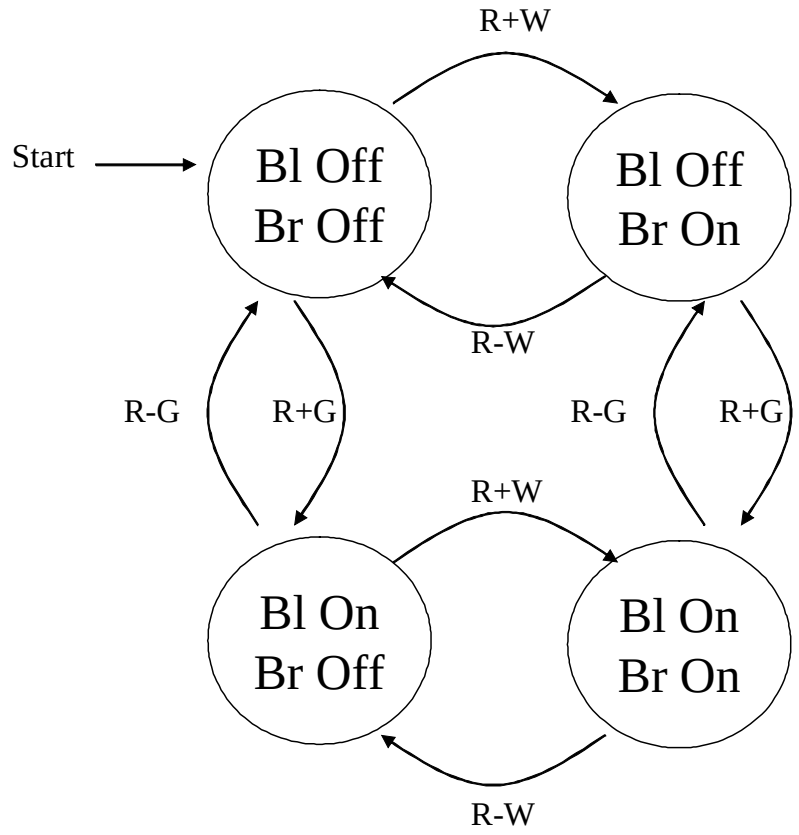
State number 8 represents a final state. If we ever reach this state, then it means that we have found an author/title pair. Naturally, this simple automaton will fail on certain types of inputs (e.g. the sentence "the fork by the spoon" inside a bulleted list).

However, this type of automata is very common for various forms of lexical analysis and pattern matching. Each state in the automata represents some knowledge of what inputs the machine has encountered.

As a final real-world example, consider a typical gas furnace. Most gas furnaces have three terminals that lead to the thermostat. There is a R, W, and G terminal as shown below:



The R terminal is the hot wire and completes a circuit. When R and G are connected, the blower turns on. When R and W are connected, the burner comes on. Any other state where R is not connected to either G or W results in no action. The job of the thermostat is to connect these terminals together automatically based on the temperature. We can model the results of making various connections in the automaton below.



Note that in this example, we've left out connections that have no effect (e.g., connecting W and G). Once the logic in the automata has been formalized, the model can be used to construct an actual circuit to control the furnace (i.e., a thermostat). The model can also help to identify states that may be dangerous or problematic. For example, if we stay in the state with the Burner On and the Blower Off, there is the possibility the burner will overheat the furnace without the flow of air. We might then try to avoid this state or add some additional states to prevent failure from occurring (e.g., a timeout or failsafe).

Languages and Grammars

Languages and grammars provide a different "view" of computing. We will see that often languages and grammars are identical to automata. Consider once again the automata that we described to check for authors and titles in a HTML document. Rather than view this automata as a set of states, we can view this as the problem of determining

all of the strings that make up valid author/title pairs. The set of all valid strings accepted by the automata makes up the Language for this particular problem.

Just like English, languages can be described by grammars. For example, below is a very simple grammar:

S→ Noun Verb-Phrase
Verb-Phrase → Verb Noun
Noun → { Kenrick, cows }
Verb → { loves, eats }

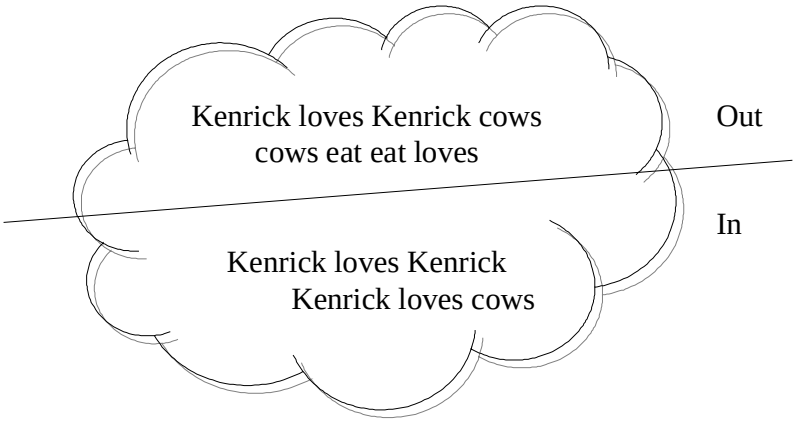
Using this simple grammar our language allows the following sentences:

Kenrick loves Kenrick
Kenrick loves cows
Kenrick eats Kenrick
Kenrick eats cows
Cows loves Kenrick
Cows loves cows
Cows eats Kenrick
Cows eats cows

The above sentences are "in" the language defined by the grammar. Sentences that are not in the language would be things like:

Kenrick loves cows and kenrick.
Cows eats love cows.
Kenrick loves chocolate.

The first two sentences are not possible to construct given the grammar. The last sentence uses a word (chocolate) that is not defined in the "alphabet" of the language. Later we'll see ways to go back and forth between a grammar-based definition for languages and an automata based definition. You can think of this as a game, given a sentence (we'll call this. a "string") determine if it is in or out:



Notice what the grammar has done for us. There is a big fuzzy set of all possible sentences we can make with the words in our alphabet. This may very well be an infinitely large set. The grammar is making a crude "cut" through this space, singling out certain sentences that are acceptable. When we look at this in a more formal light, we will examine these languages as patterns, subsets of the larger set, and in terms of properties of the language (what separates it from the rest of the set).

Perhaps a better known example for grammars are for "parsers", the component of a compiler that deals with the recursively nested features of typical programming languages. For example, a grammatical rule like $E \rightarrow E+E$ allows an expression to be formed by taking any other two expressions and placing a "+" in between. Using this rule we can recursively define complex expressions like "E1 = E2 + E3 + E4 + E5".

Before we leave languages, let's make a few definitions:

An **alphabet** is a finite, nonempty set of symbols. By convention we use the symbol $\boldsymbol{\Sigma}$ for an alphabet. In the above example, our alphabet consisted of words, but normally our alphabet will consist of individual characters.

Some sample alphabets:

1. $\Sigma = \{0,1\}$ the binary alphabet
2. $\Sigma = \{a,b, \ldots z\}$ the set of all lowercase letters

A **string** (or sometimes a **word**) is a finite sequence of symbols chosen from an alphabet. For example, 010101010 is a string chosen from the binary alphabet, as is the string 0000 or 1111.

The **empty string** is the string with zero occurrences of symbols. This string is denoted $\boldsymbol{\varepsilon}$ and may be chosen from any alphabet.

The **length** of a string indicates how many symbols are in that string. For example, the string 0101 using the binary alphabet has a length of 4. The standard notation for a string w is to use $\boldsymbol{|w|}$. For example, |0101| is 4.

Powers of an alphabet: If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

For example, given the alphabet $\Sigma = \{0,1,2\}$ then:

$\Sigma^0 = \{\varepsilon\}$
$\Sigma^1 = \{0,1,2\}$
$\Sigma^2 = \{00,01,02,10,11,12,20,21,22\}$
$\Sigma^3 = \{000,001,002,\ldots 222\}$

Note that Σ and Σ^1 are two different beasts. The first is the alphabet; its members are 0,1,2. The second is the set of strings whose members are the strings 0,1,2, each a string of length 1.

By convention, we will try to use lower-case letters at the beginning of the alphabet to denote symbols, and lower-case letters near the end of the alphabet to represent strings.

The **set of all strings** over an alphabet is denoted by **Σ^***. That is:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ...$$

Sometimes it is useful to exclude the empty string from the set of strings. The set of nonempty strings from the alphabet is denoted by **Σ^+**.

Finally, to **concatenate** strings, we will simply put them right next to one another. If x and y are strings, where x=001 and y=111. For any string w, the equation $\varepsilon w = w\varepsilon = w$.

Formal Definition of Languages

Given our definition for the alphabet, we can now formally define a language. A set of strings all of which are chosen from some Σ^* is called a language. If Σ is an alphabet and L is a subset of Σ^* then **L is a language over Σ.**

Note that a language need not include all strings in Σ^*.

Here are some abstract examples of languages:

1. The language of all strings consisting of n 0's followed by n 1's, for some n>=0: { ε, 01, 0011, 000111, …}
2. The set of binary numbers whose value is a prime: { 10, 11, 101, 111, …}
3. Ø is the empty language, which is a language over any alphabet.
4. {ε} is the language consisting of only the empty string. Note that this is not the same as example #3, the former has no strings and the latter has one string.

A **problem** is the question of deciding whether a given string is a member of some particular language. More colloquially, a problem is expressed as membership in the language.

Languages and problems are basically the same thing. When we care about the strings, we tend to think of it as a language. When we assign semantics to the strings, e.g. maybe the strings encode graphs, logical expressions, or integers, then we will tend to think of the set of strings as a solution to the problem.

Finally, a notation we will commonly use to define languages is by a "set-former":

{ w | something about w }

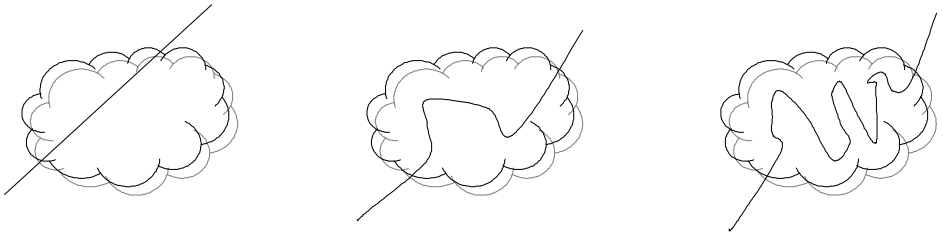
The expression is read "the set of words w such that (whatever is said about w to the right of the vertical bar)." For example:

1. {w | w consists of an equal number of 0's and 1's }.
2. {w | w is a binary integer that is prime }
3. { 0^n1^n | n >=1 }. This includes 01, 0011, 000111, etc. but not ε
4. { 0^n1 | n>=0 }. This includes 1, 01, 001, 0001, 00001, etc.

Turing Machines, More on Languages

As we will see, finite state automata and the corresponding languages and grammars are powerful but also limited in what they are able to compute. We will measure more closely the boundary of what is computable as the class progresses.

Finite state automata provide only a crude "cut" of Σ* to select the strings we will accept. Turing machines and more complex grammars provide for more sophisticated ways to define the language. One way this will be accomplished is there will no longer be a finite set of states, but an infinite number of possible states. The increase in complexity is depicted in the figure below.



Crude
e.g. Simple automata

Complex
e.g. Turing Machine

As we study Turing machines and other devices to describe these languages, we'll also look at the associated properties of these languages. A short taxonomy of these devices and their associated grammars is shown below.

Uncomputable	
Turing Machines	Phrase Structure
Linear bounded automata	Context-Sensitive
Pushdown automata	Context-Free
Finite state automata	Regular
Machines	Grammars/Languages

Complex

Crude

Complexity

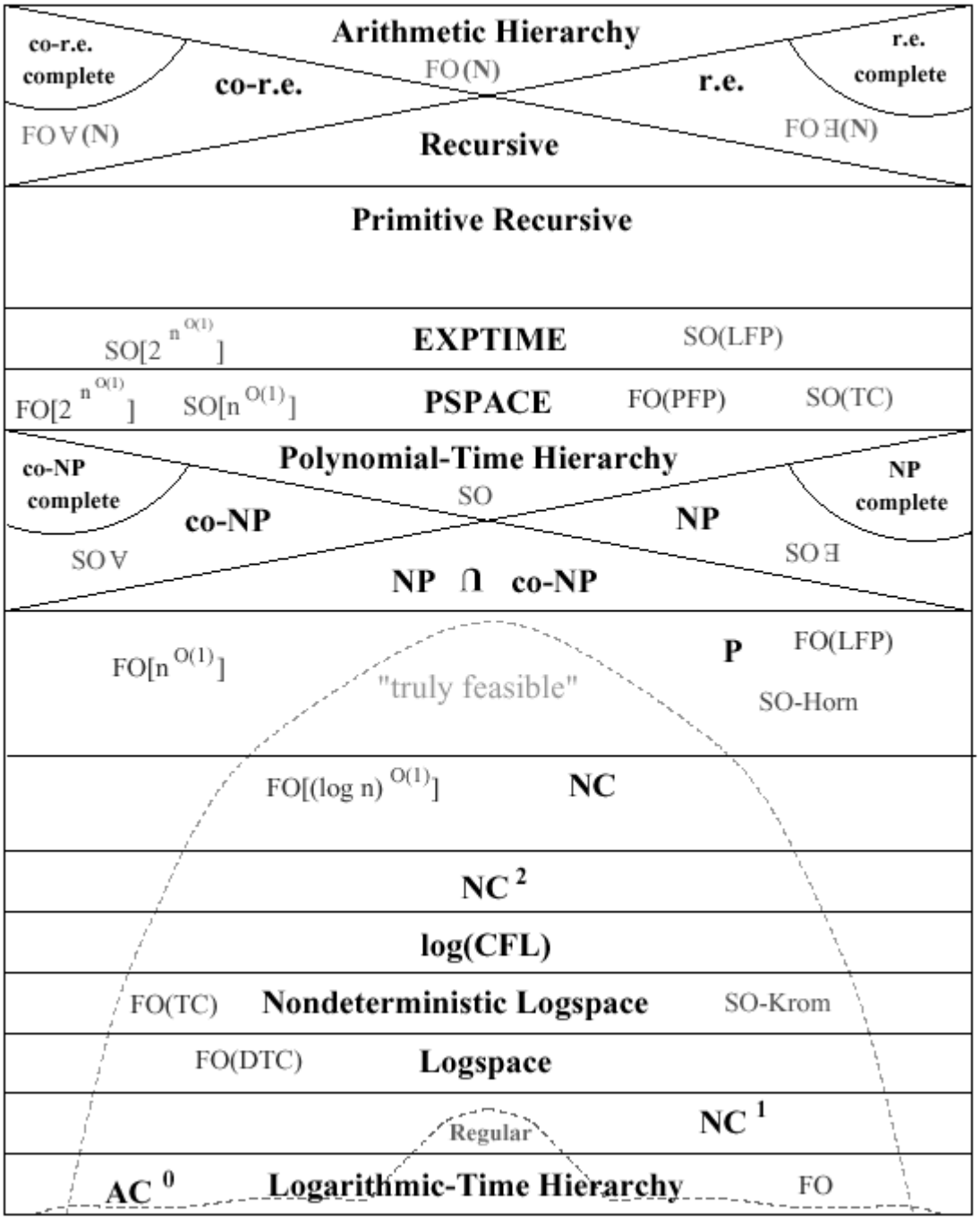
As the previous diagram implies, some problems are uncomputable. Complexity is the study of the limits of computation. There are two important issues:

1. Decidability. What can a computer do at all? The problems that can be solved by a computer in a realistic amount of time are called decidable. Some problems are undecidable, or only semi-decidable.

   For example, suppose you have a procedure for generating (enumerating) members of a set S, but no general procedure for deciding if something is not in S (later we'll see this has a relationship between recursively enumerable but not recursive sets). You want to know if x is an element of S: if it is in S you can just generate elements of S until you produce x (which you will do after a finite number of steps). But suppose it is not in S. You can never be sure that you won't generate it if S is not finite.

2. Intractability. What can a computer do efficiently? This studies the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input. Typically we will take all polynomial functions to be tractable, while functions that grow faster than polynomial intractable.

A diagram indicating some classes of complexity for different types of problems and machines are shown in the figure on the next page. We'll discuss only a few of the items on the diagram, in particular the relationship between problems that can be solved in polynomial time and those that may be solved in nondeterministic polynomial time.



Complexity Hierarchy

Introduction to Formal Proof

For the rest of this handout we will review methods to prove the truth of a statement. In this class we'll take a middle grojund between extremely formal proofs (very rigorous and exact) and "personal feeling" proofs (not very exact, but gets the general idea across). Both camps have their pro's and con's. Very complex programs are too difficult to analyze formally, and instead we must rely on software engineering testing procedures. However, with a tricky recursive function or iteration is unlikely to be correct unless the programmer can prove it is correct or at least understand the process of proving it correct.

Automata theory lends itself to both deductive and inductive proofs. We'll spend most of our time on inductive proofs, but let's look first at deductive proofs.

Deductive Proofs

A deductive proof leads us from a *hypothesis* H to a *conclusion* C given some *statements*. We'll use inductive proofs for recursive problems. For deductive proofs, think of Sherlock Holmes reasoning out a line of logic to prove that a particular person committed a crime. This is the type of reasoning that goes into a deductive proof.

Consider the following theorem:

If x >=4 then 2^x >= x^2

Here, H is x>=4 and C is $2^x >= x^2$.

Intuitively, it should not be difficult to convince yourself that this statement is true. Each time x increases by one, the left hand side doubles in size. However, the right side increases by the ratio $((x+1)/x)^2$. When x=4, this ratio is 1.56. As x increases and approaches infinity, the ratio $((x+1)/x)^2$ approaches 1. This means the ratio gets smaller as x increases. Consequently, 1.56 is the largest that the right hand side will increase. Since 1.56 < 2, the left side is increasing faster than the right side, therefore the conclusion must be true as long as we start with a value like x=4 where the inequality is already satisfied.

What we have just done is an informal but accurate proof of the theorem using deduction. We'll return to this and perform an inductive proof.

Basic Formal Logic

First, let's look at some more deductive proofs using formal logic. An "If H then C" statement is typically expressed as:

$H \Rightarrow C$ or H implies C

The logic truth table for implication is:

H	C	H$\Rightarrow$ C (i.e. $\neg H \vee C$)
F	F	T
F	T	T
T	F	F
T	T	T

Sometimes we will have If and Only If statements, e.g. "If and only if H then C" means that H$\Rightarrow$C and C$\Rightarrow$ H. Sometimes this will be written as H$\Leftrightarrow$C or "H iff C". The truth table is:

H	C	H$\Leftrightarrow$C (i.e. H equals C)
F	F	T
F	T	F
T	F	F
T	T	T

Modus Ponens

Going back to implication, let's say that we are given H$\Rightarrow$C . Further, suppose we are given that H is true. Then by the logic of the truth table, we know that C must be true. Therefore, we can conclude C. This type of reasoning is called **modus ponens** (Latin for ``method of affirming") and can be used to form chains of logic to reach a desired conclusion.

In other words, given:

H$\Rightarrow$C and
H

Then we can infer C.

For example, given: "If X and Y are siblings then X and Y are related" as a true assertion, and also given "X and Y are siblings" as a true assertion, then we can conclude "X and Y are related."

Modus Tollens

A slightly trickier but equally true form of reasoning is **modus tollens** (Latin for ``method of denying"). This reasons backwards across the implication. Cognitive psychologists have shown that under 60% of college students have a solid intuitive understanding of Modus Tollens versus almost 100% for Modus Ponens.

Modus Tollens states the following. If we are given:

$H \Rightarrow C$ and
$\neg C$

then we can infer $\neg H$.

For example, given: "If X and Y are siblings then X and Y are related" as a true assertion, and also given "X and Y are not related" as a true assertion, then we can conclude "X and Y are not siblings."

Quiz: Is the following reasoning valid? Given: "If X and Y are siblings then X and Y are related" as a true assertion, and also given "X and Y are not siblings" what can we conclude about the relation between X and Y?

Here is a deductive reasoning example of Sherlock Holmes in action using modus ponens:

> "And now we come to the great question as to the reason why. Robbery has not been the object of this murder, for nothing was taken. Was it politics, or was it a woman? That is the question confronting me. I was inclined from the first to the latter supposition. Political assassins are only too glad to do their work and fly. This murder had, on the contrary, been done most deliberately and the perpetrator had left his tracks all over the room, showing he had been there all the time." - A. Conan Doyle, A Study in Scarlet

We can break the story into the following propositions:

P1: It was robbery.
P2: Nothing was taken.
P3: It was politics.
P4: It was a woman.
P5: The assassin left immediately.
P6: The assassin left tracks all over the room.

Holmes makes the following propositions:

1. $P2 \Rightarrow \neg P1$	If nothing was taken, it was not robbery
2. P2	Nothing was taken
3. $P1 \vee (P3 \vee P4)$	It was robbery, politics, or a woman
4. $P3 \Rightarrow P5$	If it was politics the assassin will leave immediately
5. $P6 \Rightarrow \neg P5$	If one leaves tracks then one did not leave immediately
6. P6	One left tracks

A. Using modus ponens and modus tollens on the deductive trail, from #1 and #2 we can infer ¬P1.
B. From #5 and #6 we can infer ¬P5.
C. From ¬P5 and #4 we can conclude ¬P3.
D. #3 can be rewritten as $\neg P1 \Rightarrow (P3 \vee P4)$. From the result of A, we can infer P3 $\vee$ P4 using modus ponens.
E. P3 $\vee$ P4 can be rewritten as $\neg P3 \Rightarrow P4$. using the results of C, we can then infer P4.

This means the object of the murder is that it was a woman!

Here are a few more:

1. If Elvis is the king of rock and roll, then Elvis lives. Elvis is the king of rock and roll. Therefore Elvis is alive. Valid or invalid?

This argument is valid, in that the conclusion is established (by Modus ponens) if the premises are true. However, the first premise is not true (unless you live in Vegas). Therefore the conclusion is false.

2. If the stock market keeps going up, then I'm going to get rich. The stock market isn't going to keep going up. Therefore I'm not going to get rich. Valid or invalid?

This argument is invalid, specifically an inverse error. Its form is from ¬H and infer ¬C. This yields an inverse error.

3. If New York is a big city, then New York has lots of people. New York has lots of people. Therefore New York is a big city. Valid or invalid?

This argument is invalid, even though the conclusion is true. We are given $H \Rightarrow C$ and given C. This does not mean that $C \Rightarrow H$ so we can't infer H is true.

Proof by Contradiction

Suppose that we want to prove H and we know that C is true. Instead of proving H directly, we may instead show that assuming ¬H leads to a contradiction.

Here is a simple example to show that the butler is innocent: "Suppose the butler killed Col. Mustard. Then the butler had to be in the room at the time of the crime to shoot him. However, at that exact time, the butler was serving dinner to Miss Scarlet. Therefore the butler couldn't have been the one to shoot him."

Here is a more formal example:

A large sum of money has been stolen from the bank. The criminal(s) were seen driving away from the scene. From questioning criminals A, B, and C we know:

1. No one other than A, B, or C were involved in the robbery.
2. C never pulls a job without A
3. B does not know how to drive

Let A, B, and C represent the propositions that A, B, or C is guilty.

From the story we know that:

1. $A \vee B \vee C$	A, B, or C is guilty
2. $C \Rightarrow A$	If C is guilty, A is also guilty
3. $B \Rightarrow (A \vee C)$	If B is guilty, A or C is guilty

Is A innocent or guilty? Let's assume that A is innocent, i.e.:

A. $\neg A$
B. From $\neg A$ and #2 using modus tollens, we can infer $\neg C$
C. We thus have $\neg A \wedge \neg C$, which be De Morgan's Law is logically equivalent to $\neg(A \vee C)$
D. From $\neg(A \vee C)$ and #3 using modus tollens, we can infer $\neg B$
E. We now have $\neg A$ and $\neg B$ and $\neg C$ which contradicts assumption #1!

Since we have a logical contradiction, our assumption must be false. Therefore, A is guilty!

Proof by Contrapositive

Proof by contrapositive takes advantage of the logical equivalence between "H implies C" and "Not C implies Not H". For example, the assertion "If it is my car, then it is red" is equivalent to "If that car is not red, then it is not mine". So, to prove "If P, Then Q" by the method of contrapositive means to prove "If Not Q, Then Not P".

Example: Parity

Here is a simple example that illustrates the method. The proof will use the following definitions:

An integer x is called even (respectively odd) if there is another integer k for which $x = 2k$ (respectively $2k+1$).
Two integers are said to have the same parity if they are both odd or both even.

Theorem. If x and y are two integers for which $x+y$ is even, then x and y have the same parity.

Proof. The contrapositive version of this theorem is "If x and y are two integers with opposite parity, then their sum must be odd." So we assume x and y have opposite parity. Since one of these integers is even and the other odd, there is no loss of generality to suppose x is even and y is odd. Thus, there are integers k and m for which x = 2k and y = 2m+1. Then, we compute the sum x+y = 2k + 2m + 1 = 2(k+m) + 1, which is an odd integer by definition.

The difference between the Contrapositive method and the Contradiction method is subtle.

In contradiction, we assume the opposite of what we want to prove and try to find some sort of contradiction. In contrapositive, we assume ¬C and prove ¬H, given H⇒C. The method of Contrapositive has the advantage that your goal is clear: Prove Not H. In the method of Contradiction, your goal is to prove a contradiction, but it is not always clear what the contradiction is going to be at the start. Indeed, one may never be found (and will never be found if the hypothesis is false).

Inductive Proofs

Inductive proofs are essential when dealing with recursively defined objects. We can perform induction on integers, automata, and concepts like trees or graphs.

To make an inductive proof about a statement S(X) we need to prove two things:

1. Basis: Prove for one or several small values of X directly.
2. Inductive step: Assume S(Y) for Y "smaller than" X; then prove S(X) using that assumption.

Here is an example with integers that you may have seen before in a math class:

Theorem: For all n ≥ 0:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \text{(i)}$$

First prove the basis. We pick n=0. When n=0, there is a general principle that when the upper limit (0) of a sum is less than the lower limit (1) then the sum is over no terms and therefore the sum is 0. That is, $\sum_{i=1}^{0} i = 0$.

Next prove the induction. Assume n ≥ 0. We must prove that the theorem implies the same formula when n is larger. For integers, we will use n+1 as the next largest value. This means that the formula should hold with n +1 substituted for n:

$$\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$$
$$= \frac{n^2 + 3n + 2}{2} \qquad \text{(ii)}$$

How does this help us? (ii) should equal what we came up with for (i) if we just add on an extra n+1 term:

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i \right) + (n+1)$$

Expanding the right side of the equation:

$$\left(\sum_{i=1}^{n} i \right) + (n+1)$$
$$= \frac{n(n+1)}{2} + (n+1)$$
$$= \frac{n^2 + n}{2} + \frac{2n+2}{2}$$
$$= \frac{n^2 + 3n + 2}{2}$$

This is the same as what we computed from the inductive step in (ii). Therefore, the theorem is true.

Let's prove the example we looked at earlier with deduction:

Theorem: If x >=4 then $2^x >= x^2$

Basis: If x=4, then 2^x is 16 and x^2 is 16. Thus, the theorem holds.

Induction: Suppose for some x >=4 that $2^x >= x^2$. With this statement as the hypothesis, we need to prove the same statement, with x+1 in place of x:

$2^{(x+1)} >= (x+1)^2$ (i)

This is S(x+1). We should rewrite it to make use of S(x). In this case:

$2^{(x+1)} = 2 * 2^x$

and can conclude that:

$2^{(x+1)} = 2 * 2^x >= 2x^2$ (ii)

By substituting (ii) into (i) we need to show:

$2x^2 >= (x+1)^2$
$2x^2 >= x^2 + 2x + 1$
$x^2 >= 2x + 1$

Dividing both sides by x yields:

$x >= 2 + 1/x$

Since x >=4, we get some value >=4 on the left side. The right side will equal at most 2.25 and in fact gets smaller and approaches 2 as x increases. Consequently, we have proven the theorem to be true by induction.

Let's go through one more example. Consider a string of characters that consists entirely of left and right parentheses. We would like to make sure that there is a balanced number of parentheses. This is a simpler example of a larger problem where there may be additional text in the string.

Here are two ways that we can define balanced parentheses:

1. Grammatically (GB)
    a) The empty string ε is balanced.
    b) If w is balanced, then (w) is balanced.
    c) If w and x are balanced, then so is wx.
2. by Scanning (SB)
    d) w has an equal number of left and right parentheses
    e) Every prefix of w has at least as many left as right parentheses

Theorem: a string of parentheses w is GB if and only if it is SB.
i.e., GB $\Leftrightarrow$ SB.

To prove this, we must prove this both directions: GB $\Rightarrow$ SB and SB $\Rightarrow$ GB.
First let's perform induction on |w| assuming w is SB. Prove w is GB.

Basis: If w = ε then |w| = 0. By rule (a) w is GB.

Induction: Suppose the statement "SB implies GB" is true for strings shorter than w. We can split this up into two cases:

Case 1: w is not ε, but has no nonempty prefix with an equal number of (and).
Then w must begin with (and end with); i.e., w = (x).
x must be SB (why? Also x may be ε).
By the inductive hypothesis, x is GB.
By rule (b), (x) is GB; but (x) =w, so w is GB.

Case 2: w = xy, where x is the shortest, nonempty prefix of w with an equal number of (and), and $y \neq \varepsilon$.

x and y are both SB (why)?
By the inductive hypothesis, x and y are GB.
w is GB by rule (c).

Now let's perform induction on |w| assuming w is GB. Prove w is SB.

Basis: If $w = \varepsilon$ then $|w| = 0$. Both rules (d) and (e) hold. There is an equal number of left and right parentheses (zero in both cases) and every prefix (zero of them) has at least as many left as right parentheses.

Induction: Suppose the statement "GB implies SB" is true for strings shorter than w. We can split this up into two cases:

Case 1: w is not ε, but has no nonempty prefix with an equal number of (and). Then w must begin with (and end with); i.e., w = (x). w is GB because of rule (b); i.e. w = (x) and x is GB.

By the inductive hypothesis, x is SB.
Since x has equal numbers of ('s and)'s so does w=(x).
Since x has no prefix with more ('s than)'s so does (x).

Case 2: w = xy, where x is the shortest, nonempty prefix of w with an equal number of (and), and $y \neq \varepsilon$. w is GB by rule (c) and x and y are GB.

By the inductive hypothesis, x and y are SB.
xy has equal numbers of ('s and)'s because x and y both do.
If w had a prefix with more)'s than ('s, that prefix would either be a prefix of x (contradicting the fact that x has no such prefix) or it would be x followed by a prefix of y (contradicting the fact that y also has no such prefix). By contradiction, this means that w does not have a prefix with more)'s than ('s.