CS334: Theory of Computation Notes

Steven DeFalco

Fall 2023

Contents

1 Introduction

1.1 Automata, Computability, and Complexity

The central question of ***complexity theory*** is *what makes some problems computationally hard and others easy?*; the answer is unknown. Cryptography is unique in that is specifically requires computational problems that are hard, rather than easy. In ***complexirty theory***, the objective is to classify problems as easy ones and hard ones; whereas in ***computability*** theory, the classification of probelms is by those that are solvable and those that are not.
Automata theory deals with the definitions and properties of mathematical models of computations.

1.2 Mathematical Notions and Terminology

1.2.1 Sets

A ***set*** is a group of objects represented as a unit. Sets may contain any type of object, including numbers, symbols, and even othber sets. The objects in a set are called its *elements* or *members*. One way to describe a set is to list the set's elements inside braces. Thus the set

$$S = \{7, 21, 57\}$$

contains the elements 7, 21, and 59. The symbols $\in$ and $\notin$ denote set membership and nonmembership. We say that A is a ***subset*** of B, written $A \subseteq B$, if every member of A is also a member of B. We say that A is a ***proper subset*** of B, written $A \subset B$, if A is a subset of B and not equal to B.

The order of describing a set doesn't matter, nor does repetition of its members. If we do want to take the number of occurrences of members into account, we call the group a ***multiset*** instead of a set. An ***infinite set*** contains infinitely many elements.

We write the set of ***natural numbers*** N as

$$\{1, 2, 3, \ldots\}$$

. The set of ***integers*** Z is written as

$$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$$

The set with zero members is called the ***empty set*** and is written $\emptyset$. A set with one member is sometimes called a ***singleton set*** and a set with two members is called an ***unordered pair***.

When we want to describe a set containing elements according to some rule, we write $\{n|\text{rule about}n\}$.

If we have two sets A and B, the ***union*** of A and B, written $A \cup B$, is the set we get by combining all the leements in A and B into a single set. The ***intersection*** of A and B, written $A \cap B$, is the set of elements that are both A and B. The ***complement*** of A, written $\overline{A}$, is the set of elelements under consideration that are *not* in A.

1.2.2 Sequences and Tuples

A ***sequence*** of objects is a list of these objects in some order. We usually designate a sequence by writing the list within parentheses. For example, the sequence 7,21,57 would be written

$$(7, 21, 57)$$

. The order does matter in a set.

Finite sequences often are called ***tuples***. A sequence with k elements is a ***k-tuple***. A 2-tuple is also called an ***ordered pair***.

Sets and sequences may appear as elements of other sets and sequences. For example, the ***power set*** of A is the set of all subsets of A. If A is the set 0,1, the power set of A is the set $\{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.

If A and B are two sets, the ***Cartesian product*** or defcross product of A and B, written $A \times B$, is the set of all ordered pairs wherein the first element is a member of A and the second elemetn is a member of B.

1.2.3 Functions and Relations

A ***function*** is an object that sets up an input-output relationship. A function takes an input and produces and output. In every function, the same input always produces the same output.

A function is also called a ***mapping***, and, if $f(a) = b$, we say that f maps a to b.

The set of possible inputs to the function is called its ***domain***. The outputs of a function come from a set called the ***range***. The notation for saying that f is a function with domain D and range R is

$$f : D \to R$$

When the domain of a function f is $A_1 \times \cdots \times A_k$ for some sets $A_1, \ldots, A_k$, the input to f is a k-tuple and we call the a_i the ***arguments*** to f. A function with k arguments is called a ***k-ary function***, and k is called the ***arity*** of the

function. If k is 1, f has a single argument and f called a ***unary function***. If k is 2, f is a ***binaryh function***. Certain familiar binary functions are written in a special ***infix notation***, with the symbol for the function placed between its two arguments, rather than in ***prefix notation***, with the symbol preceding.

A ***predicate*** or ***property*** is a function whose range is $\{\text{TRUE}, \text{FALSE}\}$. For example, let *even* be a property that is TRUE if its input is an even number and FALSE if its input is an odd number.

A property whose domain is a set of k-tuples $A \times \cdots \times A$ is called a ***relation***, a ***k-ary relation***, or a ***k-ary relation on A***.

A special type of binaryh relation, called an ***equivalence relation***, captures the notion of two objects being equal in some feature. A binary relation R is an equivalence relation if R satisfies three conditions:

1. R is ***reflexive*** if for every x, xRx;
2. R is ***symmetric*** if for every x and y, xRy implies yRx; and
3. R is ***transitive*** if for every x,y, and z, xRy and yRz implies xRz.

1.2.4 Graphs

An ***undirected graph***, or simply a ***graph***, is a set of points with lines connecting some of the points. The points are called ***nodes*** or ***vertices***, and the lines are called ***edges***.

The number of edges at a particular node is the ***degree*** of that node. No more thatn one edge is allowed between any two nodes. We may allow an edge from a node to itself, called a ***self-loop***.

We say that graph G is a ***subgraph*** of graph H is the nodes of G are a subset of the nodes of H, and the edges of G are the edges of H on the corresponding nodes.

A ***path*** in a graph is a seqeunce of nodes connected by edges. A ***simple path*** is a path that doesn't repeat any nodes. A graph is ***connected*** if every two nodes have a path between them. A path is a ***cycle*** if it starts and ends in the same node. A ***simple cycle*** is one that contains at least three nodes and repeates only the first and last nodes. A graph is a ***tree*** if it is connected and has no simple cycles. A tree may contain a specially designated node called the ***root***. The nodes of degree 1 in a tree, other than the root, are called the ***leaves*** of the tree.

A ***directed graph*** has arrows instead of lines. The number of arrows pointing from a particular node is the ***outdegree*** of that node, and the number of arrows

pointing to a particular node is the ***indegree***.

A path in which all the arrows point in the same direction as its steps is called a ***directed graph***. A directed graph is ***strongly connected*** if a directed path connects every two nodes.

1.2.5 Strings and Languages

Strings of characters are fundamental building blocks in computer science. The alphabet over which the strings are defined may vary with the application. For our purposes, we define an ***alphabet*** to be any nonemtpy finite set. The members of the alphabet are ***symbols*** of the alphabet. We generally use capital Greek letter Σ and Γ to dersignate alphabets.

A **string over an alphabet** is a finite sequence of syumbols from that alphabet, usually written next to one another and not separated by commas. If w is a string over Σ, the ***length*** of w, written $|w|$ is the number of symbols that it contains. The string of length zero is called the ***empty string*** and is written ϵ. The ***reverse*** of w, written w^R, is the string obtained by writing w in the opposite order. String z is a ***substring*** of w if z appears consecutively within w.

If we have a string x of length m and string y of length n, the ***concatenation*** of x and y, written xy, is the string obtained by appending y to the end of x, as in $x_1 \cdots x_m y_1 \cdots y_n$.

The ***lexicographic order*** of strings is the same as the familiar dictionary order. We'll occasionally use a modified lexicographic order, called ***shortlex order*** or simply ***string order***, that is identical to lexicographic order, except that shorter strings precede longer strings. Thus the string ordering of all strings over the alphabet $\{0, 1\}$ is

$$(\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots)$$

Say that string x is a ***prefix*** of string y if a string z exists where $xz = y$, and that x is a ***proper prefix*** of y if in addition $x \neq y$. A ***language*** is a set of strings. A language is a ***prefix-free*** if no member is a proper prefix of another member.

1.2.6 Boolean Logic

Boolean logic is a mathematical system built around the two values TRUE and FALSE. The values TRUE and FALSE are called the ***Boolean values*** and are often represented by the values 1 and 0.

We can manipulate Boolean values with the ***boolean operations***. The simplest boolean operation is the ***negation*** or ***NOT*** operation, designated with the symbol $\neg$. The negation of a Boolean value is the opposite value. We designate the ***conjunction*** or ***AND*** operation with the symbol $\wedge$. The conjunction

of two Boolean values is 1 if both of those values are 1. The ***disjunction*** or ***OR*** operation is designated with the symbol $\vee$. The disjunction of two Boolean values is 1 if either of those values is 1.

The ***exclusive or*** or ***XOR*** operation is designated by the $\oplus$ symbol and is 1 if either but not both of its two operands is 1. The ***equality*** operation, written $\leftrightarrow$, is 1 if both if its operandws have the same value. Finally, the ***implication*** operation is designated by the symbol $\rightarrow$ and is 0 if its first operand is 1 and its second operand is 0; otherwise, $\rightarrow$ is 1.

The ***distributive law*** for AND and OR comes in handy when we manipulate Boolean expression; it comes in two forms:

- $P \wedge (Q \vee R)$ equals $(P \wedge Q) \vee (P \wedge R)$, and its dual
- $P \vee (Q \wedge R)$ equals $(P \vee Q) \wedge (P \vee R)$

1.3 Definitions, Theorems, and Proofs

Definitions describe the objects and notions that we use. When defining some object, we must make clear what constitutes that object and what does not.

After we have defined various objects and notions, we usually make ***mathematical statements*** about them. Typically, a statement expresses that some object has a certain property.

A ***proof*** is a convincing logical argument that a statement is true.

A ***theorem*** is a mathematical statement proved true. Occasionally, we prove statements that are interesting only because they assist in the proof of another, more significant statement. Such statements are called ***lemmas***. Occasionally a theorem or its proof may allow us to conclude easily that other, related statements are true. These statements are called ***corollaries*** of the theorem.

1.3.1 Finding Proofs

The only way to determine the truth or falsity of a mathematical statement is with a mathematical proof. Experimenting with examples is especially helpful. Thus if the statement says that all objects of a certain type have a particular property, pick a few objects of that type and observe that they actually do have that property. After doing so, try to find an object that fails to have the property, called a ***counterexample***. If the statement actually is true, you will not be able to find a counterexample.

1.4 Types of Proofs

Several types of arguments arise frequently in mathematical proofs. Here, we describe a few that often occur in the theory of computation.

1.4.1 Proof by Construction

Many theorems state that a particular type of object exists. One way to prove such a theorem is by demonstrating how to construct the object. This technique is a ***proof by construction***.

1.4.2 Proof by Contradiction

In one common form of argument for proving a theorem, we assume that the theorem is false and then show that this assumption leads to an obviously false consequence, called a contradiction.

1.4.3 Proof by Induction

Proof by induction is an advanced method used to show that all elements of an infinite set have a specified property. For example we may use a proof by induction to show that an arithmetic expression computes a desired quantity for every assignment to its variables, or that a program works correctly at all steps or for all inputs.

Every proof by induction consists of two parts, the ***basis*** and the ***induction step***. Each part is an individual proof on its own. In the induction step, the assumption that $P(i)$ is true is called the ***induction hypothesis***.

Basis: Prove that $P(i)$ is true. **Induction step:** For each $i \geq 1$, assume that $P(i)$ is true and use this assumption to show that $P(i+1)$ is true.

2 Regular Languages

Real computers are quite complicated: too much so to allow us to set up a manageable mathematical theory of them directly. Instead, we use an idealized computer called a ***computational model***. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. Thus we will use several different computation models, depending on the features we want to focus on. We begin with the simplest model, called the ***finite state machine*** or ***finite automaton***.

2.1 Finite Automatata

Finite automata are good models for computers with an extremely limited amount of memory. Finite automatat and their probabilistic counterpart ***Markov***

chains are useful tools when we are attempting to recognize patterns in data.

Finite automata can be represented by a ***state diagram***. The ***start state*** is indicated by the arrow point at it from nowhere. The ***accept state*** is the one with a double circle. The arrows going from one state to another are called ***transitions***. When this such automation receives an input, it processes that and produces an output; the output is either ***accept*** or ***reject***.

2.2 Formal Definition of Finite Automaton

A finite automaton has several parts. It has a set of states and rules for going from one state to another, depending on the input symbol. It has an input alphabet that indicates the allowed input symbols. It has a tart state and a set of accept states. The formal definition says that a finite automaton is a list of those five objects: set of states, input alphabet, rules for moving, start state, and accept states. In mathematical language, a list of five elements is often called a 5-tuple. Hence we define a finite automaton to be a 5-tuple consisting of these five parts.

We use something called a ***transition function***, frequently denoted δ, to define the rules for moving. If the finite automaton has an arrow from a state x to a state y labeled with the input symbol 1, that means that if the automaton is in state x when it reads a 1, it then moves to state y. We can indicate the same thing with the transition function by saying that $\delta(x, 1) = y$. This notation is a kind of mathematical shorthand.

Definition 1 *A **finite automaton** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where*

1. *Q is a finite set called the **states**,*
2. *Σ is a finite set called the **alphabet**,*
3. *δ: $Q \times \Sigma \to Q$ is the **transition function**,*
4. *$q_0 \in Q$ is the **start state**, and*
5. *$F \subseteq Q$ is the **set of accept states***

If A is the set of all strings that machine M accepts, we say that A is the ***language of machine M*** and write $L(M) = A$. We say that ***M recognizes A*** or that ***M accepts A***. Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language-namely, the empty language $\emptyset$.

2.3 Formal Definition of Computation

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w | M \text{ accepts } w\}$.

Definition 2 *A language is called a* ***regular language*** *if some finite automaton recognizes it.*

2.4 Designing Finite Automata

Try putting yourself in the place of the machine you are trying to design and then see how you would go about performing the machine's task. Suppose that you are given some language and want to design a finite automaton that recognizes it. Pretending to be the automaton, you receive an inpuit string and must determine whether it is a member of the language the automaton is supposed to recognize. First, in order to make these decisions, you have to figure out what you need to remember about the string as you are reading it. For many languages, you don't need to remember the entire input. You need to remember only certain crucial information. Exactly which information is crucial depends on the particular language considered.

Once you have determined the necessary information to remember about the string as it is being read, you represent this information as a finite lit of possibilities. Then you assign a state to each of the possibilities. Next, you assign the transitions by seeing how to go from one possibility to another upon reading a symbol. Next, you set the start state to be the stae corresponding to the possibility associated with having seen 0 symbols so far. Last, set the accept states to be those corresponding to the possibilities where you want to accept the input string.

2.5 The Regular Operations

In the theory of computation, the objects are languages and the tools include operations specifically designed for manipulating them. We define three operatioons on languages, called the ***regular operations***, and use them to study properties of the regular languages.

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- ***Union***: $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- ***Concatenation***: $A \circ B = \{xy | x \in A \text{ and } y \in B\}$
- ***Star***: $A* = \{x_1 x_2 \dots x_k | k \geq 0 \text{ and each } x_i \in A\}$

The union operation takes all the strings in both A and B and lumps them together into one language. The concatenation operation attaches a string from A in front of a string from B in all possible ways to get the strings in the new language. The star operation is ***unary operation*** instead of a ***binary operation***. It works by attaching any number of strings in A together to get a string in the new language.

2.6 Nondeterminism