LE/EECS 4115 - Computational Complexity
Winter 2023
Recitation Set (Sample Version)

S.Toyonaga

April 19, 2023

Contents

1 Week 1: Course Introduction

Readings: Chapters 0 and 3

Exercises:

- $0.1 - 0.8$
- $3.1, 3.2, 3.7, 3.8, 3.10 - 3.13$

1.1 Introduction to Computational Complexity

Computational complexity has the goal of **putting theoretical boundaries over the complexity of problems**.

Given a problem, we want to answer the question of how hard it is:

- We need to define what complexity means in this context. What does it mean for a problem to be difficult, or hard to solve?
- Our definition of hardness must be measurable.

In general, this course will focus on analyzing:

1. **Time Complexity:** How much longer will this take?
    - We care about time efficiency, which matters in two ways:
        - Having a good estimate (# Operations) about how long it will take to run a program on a given input n.
        - If we know the theoretical boundaries, we can know how much our program can be improved. It provides good perspectives on practical solutions.
2. **Space Complexity:** How much space will it need?
    - We can raise the issues about time complexity for space complexity.
    - Knowing how much memory we will need when running a program can be vital to make sure that we have enough resources, and that we are not using too many.

In conclusion, we should always be asking, **"Where is the limit?"**

1.2 Mathematical Prerequisites

1.2.1 Sets

- A set is a group of objects represented as a unit. They may contain any type of object, including numbers, symbols, and even other sets.
- The objects in a set are called its elements or members.
- The order of describing a set does not matter, nor does the repetition of its members.

  Note: If we do want to take the number of occurrences of members into account, we call the group a multiset instead of a set.

Important Set Terminology:

1. Empty Set: The set with zero members, written by $\emptyset, \Phi,$ *or* $\{\}$
2. Singleton Set: The set with one member.
3. Unordered Pair: The set with two members
4. Set-Builder Notation: Used when we want to describe a set containing elements according to some rule. We typically write such a rule in the form of $\{x|\ \textit{rule about}\ x\}$
5. Set Union: If we have two sets A and B, the union, written $A \cup B$ is the set we get by combining all the elements in A and B into a single set.

   $A \cup B = \{x \in \Omega | x \in A \vee x \in B\}$
6. Set Intersection: The intersection of A and B, written $A \cap B$, is the set of elements that are both in A and B.

   $A \cap B = \{x \in \Omega | x \in A \wedge x \in B\}$
7. Complement: The complement of A, written $\bar{A}$, is the set of all elements under consideration that are not in A.

   $\bar{A} = \{x \in \Omega | x \notin A\}$

1.2.2 Sequences

- A sequence of objects is a list of objects in some order. We usually designate a sequence by writing the list within parentheses.
  - Order does not matter in a set, but in a sequence, it does.
  - Repetition does not matter in a set, but in a sequence, it does.
- Finite sequences are often called tuples. A sequence of k elements is a k-tuple.
- A 2-tuple is called an ordered pair.
- Sets and sequences may appear as elements of other sets and sequences. One such example is a Power Set.

1.2.3 Cartesian Product / Cross Product

- If A and B are two sets, the Cartesian Product or Cross 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 element is a member of B
- Assuming that A and B are non-empty, then $|A \times B| = |A| \times |B|$
- Note: If we have the Cartesian Product of a set with itself, we use the shorthand: $\overbrace{A \times A \times \cdots A}^{k} = A^k$

Example: If $A = \{1, 2\}$ and $B = \{x, y, z\}$, then what is the value of $A \times B$?

$A \times B = \{(1, x), (1, y), (1, z), (2, x), (2, y), (2, z)\}$

Example: The set $N \times N$ consists of all ordered pairs of the natural numbers.

We may also write it as: $\{(i, j)|i, j \geq 1\}$

1.2.4 Strings and Languages

- We define an alphabet to be any non-empty finite set.
- The members of the alphabet are the symbols of the alphabet.
- Σ and Γ are used to designate alphabets.

  i.e.,

  - $\Sigma = \{0, 1\}$
  - $\Gamma = \{0, 1, x, y, z\}$
- A string over an alphabet is a finite sequence of symbols 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 0 is called the empty string and is written ε
- If w has length n, we can write $w = w_1w_2...w_n$ where each $w_i \in \Sigma$
- The reverse string 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 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. That is, the string which can be represented by $x_1...x_my_1...y_n$
- To concatenate a string with itself many times, we use the superscript notation $x^k \equiv \overbrace{xx\cdots x}^{k}$

Shortlex Order / String Ordering

- The lexicographical ordering of strings is the same as the familiar dictionary ordering.

  Note: We'll occasionally use a modified lexicographic order called shortlex order / string order. This is identical to lexicographical ordering, however, shorter strings precede longer ones.

Example: The string ordering of all strings over the alphabet $\{0, 1\}$ is: $(\varepsilon, 0, 1, 00, 01, 10, 11, 000, \cdots)$

String Prefix

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

String Suffix

- A suffix is a substring that appears at the end of the word.

Language

- A language is a subset of strings of the alphabet.
- In this course, we will address many problems by proposing languages.

1.2.5 Functions

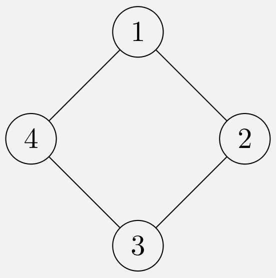
- A function is an object that sets up an input-output relationship. It takes an input and produces an output.
- In every function, the same input always produces the same output.
- A function f from the non-empty sets A to B is the assignment of exactly one element of B to each element of A
- If f is a function from A to B, we write $f : A \rightarrow B$
- If f is a function from A to B, then:
  - A = Domain
  - B = Co-Domain / Range
- If $f(a) = b$, we say that b is the image of a, and a is the pre-image of b.

1.2.6 Graphs

- If V is the set of nodes of G and E is the set of edges, we say that $G = (V, E)$
- If the graph is directed, the elements of E are tuples.
- **Important Terminology and Notation:**
  1. Subgraph: A graph G is a subgraph of H if 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.

2. Path: A sequence of nodes connected by edges.
3. Simple Path: A path that does not repeat any nodes.
4. Connected Graph: A graph is connected if every two nodes have a path between them.
5. Simple Cycle: A cycle that contains at least three nodes and repeats only the last node.
6. Degree: The number of edges at a particular vertex.
7. Loop: An edge such that its origin and destination are the same node.
8. Tree: A graph is a tree if it is connected and has no simple cycles.
9. Root: A specially designated node in a tree. The nodes of degree 1 in a tree, other than the root are called the leaves of the tree.

Example: Given the following graph, provide its formal description.

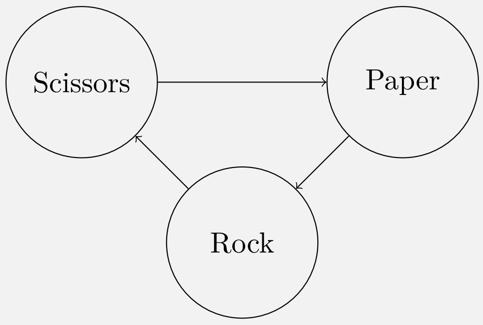


A formal description of the graph above is
$G = (\{1,2,3,4\}, \{(1,2),(2,3),(3,4),(4,1)\})$

1.2.7 Directed Graph

- A directed graph has arrows which represent the direction of the edge. We represent the edge from nodes i to j as an ordered pair.
- A path in which all the arrows point in the same direction as it steps is called a direct path.
- A direct path is strongly connected if a directed graph connects every two nodes.
- A direct graph is a handy way of depicting binary relations. If R is a binary relation whose domain is $D \times D$, a labelled graph $G = (D, E)$ represents R, where $E = \{(x,y)|xRy\}$

Example: Draw a directed graph for a relation, "Beats" in a game of Rock, Paper, Scissors



$G = (\{Scissors, Paper, Rock\}, \{$
$(Scissors, Paper),$
$(Paper, Rock),$
$(Rock, Scissors)$
$\})$

1.2.8 Boolean Logic

- A mathematical system built around the two values TRUE and FALSE.
- TRUE and FALSE are called the boolean values and are often represented by values of 1's and 0's.
  - TRUE is represented by 1
  - FALSE is represented by 0
- We can manipulate boolean values with boolean operations.
  - Boolean operations combine simple statements into more complex boolean expressions

P	$\neg P$
1	0
0	1

P	Q	$P \wedge Q$
1	1	1
1	0	0
0	1	0
0	0	0

P	Q	$P \vee Q$
1	1	1
1	0	1
0	1	1
0	0	0

P	Q	$P \oplus Q$
1	1	0
1	0	1
0	1	1
0	0	0

P	Q	$P \implies Q$
1	1	1
1	0	0
0	1	1
0	0	1

P	Q	$P \equiv Q$
1	1	1
1	0	0
0	1	0
0	0	1

1.2.9 Logical Equivalences

There are rules that help us manipulate the above logical operations. Some examples have been provided below:

1.2.10 Logical Equivalences (Formulas)

1. Distributive Laws

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$
$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$

2. De Morgan's Laws

$$\neg(p \wedge q) \equiv \neg p \vee \neg q$$
$$\neg(p \vee q) \equiv \neg p \wedge \neg q$$

1.2.11 Theorems and Proofs

Important Terminology

1. Definition: Describes the objects and notions that we use.
2. Mathematical Statements: A statement that expresses that some object has a certain property. Do note that statements may be true or false.
3. Proof: A convincing logical argument that a statement is true. An argument that is airtight; has proof beyond any doubt.
4. Counterexample: Used to show that a statement is not true.
5. Theorem: A mathematical statement proved true.
6. Lemma: A proved statement that assists in another proof.
7. Corollary: A statement that is proved to be true from another written proof.

1.2.12 How to Write Good Proofs

You will build your proof ideas, and like a toolbox, coming up with the first idea for a proof is hard. Once some tools have been used (learned) once, it is not as hard to use it again.

A good proof should be understood by someone if they are reading it for the first time, as long as they understand the statement.

- Explanations should be spelled out, in words. Formulas are not enough.
- Explain the method used to prove the statement, ensuring that no steps are skipped or assumed.
- Very elegant proofs are often written from the end to the beginning. Figure out the steps, and then write the complete proof.
- Practice, practice, practice...

1.3 Turing Machines

1.3.1 Formal Definition

A Turing Machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are all finite sets and:

1. Q is the set of states
2. Σ is the input alphabet, not containing the beginning symbol (\$) and blank symbol ($\sqcup$)
3. Γ is the tape alphabet, where \$, $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$

1.3.2 Configurations

- As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location.
- A setting of these three elements is called a configuration of the Turing machine.
- A configuration of a Turing machine describes its situation at some point during the computation.
- For a state q and two strings u and v over the tape alphabet Γ, we write uqv for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v.

Additional Review: Configuration Yields

- We say that configuration C_1 yields configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step.
- If $\delta(q_i, b) = (q_j, c, L)$, then uaq_ibv yields uq_jacv
- If $\delta(q_i, b) = (q_j, c, R)$, then uaq_ibv yields $uacq_jv$

Special cases occur when the head is at one of the ends of the configuration.

1. Left-Hand End
   - $q_i b v$ yields $q_j c v$
2. Right-Hand End
   - $u a q_i$ is equivalent to $u a q_i \sqcup$ because we assume that blanks follow the part of the tape represented in the configuration.

1.3.3 Steps

- A step of a Turing machine M is a relation between two configurations. For example, we can write $w_1 z q x w_2 \vdash w_1 z p y w_2$ which means that we read x, wrote y, went from state q to p, and lastly, did not move the Turing machine's head.

1.3.4 Computation

- Initially, M receives its input $w = w_1 w_2 \cdots w_n \in \Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank.
- The first blank appearing on the tape marks the end of the input.
- Once M has started, the computation proceeds according to the transition function.
- If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L.
- The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.

Note: We can also say that a computation of M is a (possibly infinite) sequence of configurations.
If $C_0 \vdash C_1 \vdash \cdots \vdash C_i$, we say that $C_0 \vdash^* C_i$.

1.3.5 Accepted Languages

The language accepted by the Turing machine M is:
$L(M) = \{w \in \Sigma^* | q_0\$w \vdash^* w_1 q_{accept} w_2$, for some $w_1, w_2 \in \Gamma^*\}$

This is to simply say that the language that a Turing machine accepts is the strings that put it into an accept state.

1.3.6 Case Study: $\mathbf{L} = \{w\#w | w \in \{0,1\}^*\}$

- We want M_1 to accept if its input is a member of B and to reject otherwise. That is, whether the input comprises two identical strings separated by a # symbol.
- The input is too long for you to remember it all, but you are allowed to move back and forth over the input and make marks on it.

Strategy

- We make multiple passes over the input string with the read-write head:
  - On each pass, it matches one of the characters on each side of the # symbol.
  - To keep track of which symbols have been checked already, M_1 crosses off each symbol as it is examined.
  - If it crosses off all the symbols, that means that everything matches successfully, and M_1 goes into an accept state.
  - If it discovers a mismatch, it enters a reject state.

Given this strategy, we can develop the following Turing machine below:

To simplify the figure, we don't show the reject state, or the transitions going to it.

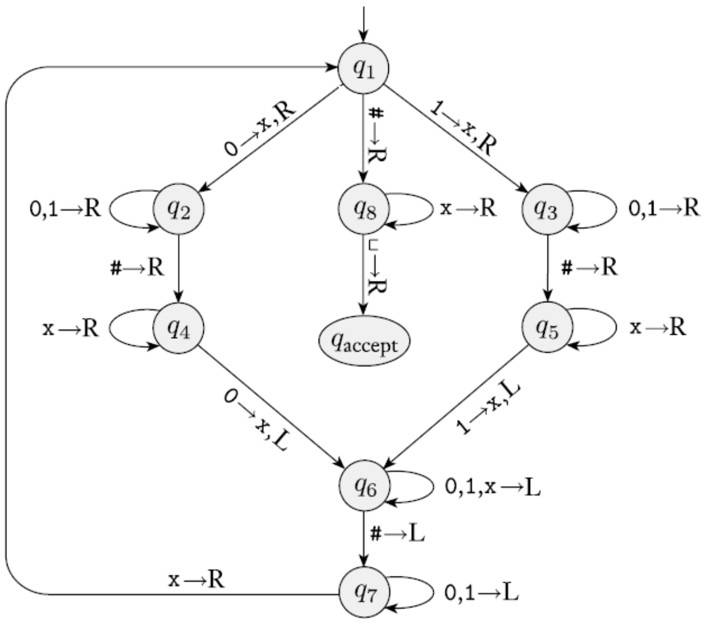
Those transitions occur implicitly whenever a state lacks an outgoing transition for a symbol in our tape alphabet.

Note: What is important to know is how the Turing Machines function. You will not be asked to design one in the assignments or the evaluations.

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*."

The following figure contains several nonconsecutive snapshots of M_1's tape after it is started on input `011000#011000`.



1.4 Multi-Tape Turing Machines

- Like an ordinary Turing Machine, but with several tapes. Each tape has its own head for reading and writing.
- Initially, the input appears on tape 1, and the others start out blank.
- The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously.

Transition Function

- $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, N\}^k$ where k is the number of tapes.
- $\delta(q_i, a_1, \cdots, a_k) = (q_j, b_1, \cdots, b_k, L, R, \cdots, L)$ means that if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left or right, or to stay put, as specified.

Notice that for our previous language, L $= \{w\#w | w \in \{0,1\}^*\}$, we can design a more optimal multi-tape Turing machine.

1. Scan across the input on the first tape, copying the content onto the second tape.
2. After reading the #, start comparing the read symbols to those written on tape 2.
3. If the contents are identical and we reach the point of reading a blank symbol, accept the input. Else, reject.

1.5 Nondeterministic Turing Machines

- At any point in a computation, the machine may proceed according to several possibilities.
- $\delta : Q \times \Gamma \rightarrow \varphi(Q \times \Gamma \times \{L, R, N\})$
- The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the accept state, the machine accepts its input.

Do note that the computation of a nondeterminsitic Turing machine is a tree, where every branch corresponds to a different computation possibility of the machine.

If some branch (Any of them) leads to the accepting state, the machine accepts its input.

1.6 Closing Remarks

- Single Tape TM $\equiv$ Multi-Tape TM $\equiv$ Nondeterministic TM
- Equivalent models recognize the same class of languages.
- To show that models are equivalent, we simply need to show that one can simulate the other.

1.7 Additional Notes: Nondeterminism Solving the Four Colour Theorem

"In mathematics, the four color theorem, or the four color map theorem, states that no more than four colors are required to color the regions of any map so that no two adjacent regions have the same color. Adjacent means that two regions share a common boundary curve segment, not merely a corner where three or more regions meet" (Wikipedia)
In class, we discussed that we can use nondeterminism to solve this problem more efficiently than by using a deterministic approach.

2 Week 2: Time Complexity

2.1 Introduction

- **Recall:** The goal of the course is to classify problems according to how difficult they are to solve.
  1. Uncomputable: No computer could solve them.
  2. Undecidable: While computable, they might have infinite computations for words outside of the language. That is, a bad input can put the machine into an infinite loop.
  3. Decidable: For these problems, there exists a Turing machine solving the problem. It will always halt, meaning that for every possible input, it will eventually stop and say yes or no.

2.2 Decidable Problems

- Do note that it is possible that a decidable problem is not solvable in practice, because it takes too much time to get to the stopping point.
- If any problem takes more than 10^{10} years to solve, we can forget about it. We can also say that 10^{18} seconds is our limit for computability.

2.2.1 Example: A Decidable but Impractical Problem

- If we would want a perfect chess solver, one that would take into account every single possible game outcome before making one move, we would have to check **all** possible chess games before the first move.
- $\#(Possibilities) = 10^{120}$
- Even if we could calculate one game per microsecond, it would take 10^{90} years to calculate the first move.

2.2.2 Example: Graham's Number

- $3 \uparrow 3 \equiv 3^3$
- $3 \uparrow\uparrow 3 \equiv 3 \uparrow (3 \uparrow 3) \equiv 3^{3^3}$
- $3 \uparrow\uparrow\uparrow 3 \equiv 3 \uparrow\uparrow 3 \uparrow\uparrow 3 \equiv 3 \uparrow\uparrow 3^{3^3}$

2.2.3 Conclusion

In conclusion, all we need to know is that it is not enough for problems to be decidable.

We need them to be decidable in a "reasonable" amount of time.

2.3 Measuring Time to Solve a Problem with Steps

Measure how much time it takes for the Turing Machine M_1 to decide the language $\mathrm{L} = \{0^k 1^k | k \geq 0\}$

Algorithm:

1. Cross off first read 0.
2. Move rightwards until the first 1 is read and cross it off.
3. Move left to first uncrossed 0
4. Repeat steps 1-3.
5. If all 0's and all 1's have been crossed out and we read a blank, accept.

Steps Analysis

- $0^k 1^k$ will take $2k$ steps to cross a pair of 0 and 1 off at a time.
- $\#(\mathit{Steps}) = k(2k) + 2k + 2k + 1 = 2k^2 + 4k + 1$
  - We multiply $2k$ by k, because for each iteration of crossing off, we must do this k times.
  - The first $2k$ resembles the number of steps we have to take to move from the right end of the tape to the left end to check that no more 0's remain.
  - The second $2k$ resembles the number of steps we have to take to move from the left end of the tape to the empty symbol to check that no more 1's remain.
  - The cost of 1 resembles the step to read the blank symbol (thus accepting.)

- **Note:** Recognizing that $w \notin L$ is generally faster, because we break the pattern before, or very close to the accept state.

2.4 Running Time

- We compute the running time of an algorithm as a function of the length of the string representing the input
- The running time will be the number of steps that the algorithm takes in order to fully solve the input.
- In **worst-case analysis**, we consider the longest running time of all inputs of a particular length.
- In **average-case analysis**, we consider the average of all the running times of inputs of a particular length.

Turing Machines: Running Time / Time Complexity
Let M be a deterministic Turing machine that halts on all inputs. The running time or time complexity of M is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

- We do not measure time complexity with a clock (in seconds), because of hardware complexities. Also, we have a Turing machine.
- We use a Turing machine to measure time because it works very well, and has nicely defined concepts (i.e., steps). It's unchanging.

2.5 Asymptotic Analysis ($Scale$:↑↓)

- The exact running time of an algorithm is often hard to measure. We tend to estimate it in the form of asymptotic analysis
- We only consider the highest contributors (order term) to the running time.

2.5.1 Big-O Notation

Let R^+ be the set of nonnegative real numbers:

Let f and g be functions $f, g : N \rightarrow R^+$. Say that $f(n) = O(g(n))$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$, $f(n) \leq c \times g(n)$. When $f(n) = O(g(n))$, we say that $g(n)$ is an upper bound for $f(n)$, or more precisely, that $g(n)$ is an asymptotic upper bound for $f(n)$, to emphasize that we are suppressing constant factors.

- Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor.

- The expression $f(n) = 2^{O(logn)}$ occurs in some analyses. Notice that $2^{O(\log_2 n)} = 2^{c \times \log_2 n} = 2^{\log_2(n^c)} = n^c = n^{O(1)}$

2.5.2 Polynomial Functions

We talk about polynomial functions if $f(n) = n^{O(1)}$

2.5.3 Exponential Functions

We talk about exponential functions if $2^{(n^\delta)}$ when $\delta \in R > 0$.

2.5.4 Little-O Notation

To say that one function is asymptotically less than another, we use small-o notation.
Let f and g be functions $f, g : N \rightarrow R^+$. Say that $f(n) = o(g(n))$ if $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = 0$.
In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < c \times g(n)$ for all $n \geq n_0$.

2.5.5 Big-O and Little-o Concluding Remarks

- Big-O says that one function is asymptotically no more ($\leq$) than another function.

- Little-o says that one function is less than ($<$) another function.

- $f(n)$ is never $o(f(n))$. However, $f(n)$ is $O(f(n))$

2.6 Time Complexity Classes

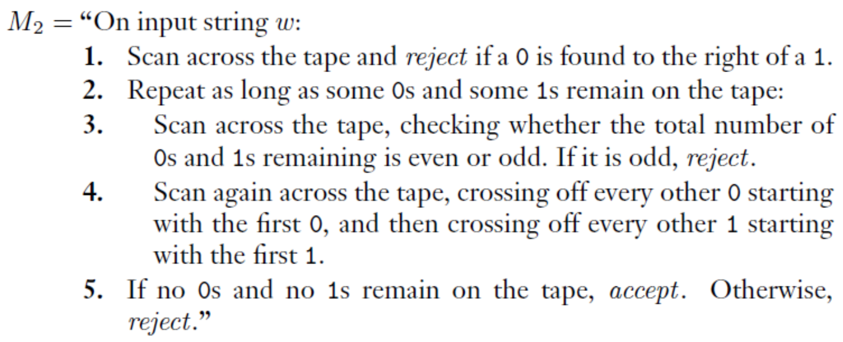
We define the following notation for classifying languages according to their time requirements.

Let $t : N \to R^+$ be a function. The time complexity class, $TIME(t(n))$ is the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Recall the language $L = \{0^k1^k | k \geq 0\}$. It is actually possible to decide L asymptotically quicker than before.

That is, it follows that L is in the complexity class $TIME(t(n))$ for $t(n) = o(n^2)$.

Consider the following algorithm:



```
$M_2$ = "On input string $w$:
    1. Scan across the tape and reject if a 0 is found to the right of a 1.
    2. Repeat as long as some 0s and some 1s remain on the tape:
    3.     Scan across the tape, checking whether the total number of
           0s and 1s remaining is even or odd. If it is odd, reject.
    4.     Scan again across the tape, crossing off every other 0 starting
           with the first 0, and then crossing off every other 1 starting
           with the first 1.
    5. If no 0s and no 1s remain on the tape, accept. Otherwise,
       reject."
```

To analyze the running time of M_2, we first obsserve that every stage takes $O(n)$ time. We then determine the number of times that each is executed.

- Stages 1 and 5 are executed once, taking a total of $O(n)$ time.
- Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + \log_2(n)$ iterations of the repeat loop occur before all get crossed off.

Thus, the total running time is $O(n) + O(n \log_2(n)) = O(n \log_2(n))$

<u>Note:</u> This result cannot be further improved on single-tape Turing machines. In fact, any language that can be decided in $o(n \log_2(n))$ time on a single-tape Turing machine is regular.

A regular language is one that is recognized by a finite automaton.

2.7 Multitape Turing Machines

We can decide the language A in $O(n)$ time if the Turing machine has a second tape.

The following two-tape TM M_3 decides A in linear time. It simply copies the 0s to its second tape and then matches them against the 1s.

M_3 = "On input string w:

1. Scan across tape 1 and reject if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, reject .
4. If all the 0s have now been crossed off, accept . If any 0s remain, reject ."

Each of the four stages uses $O(n)$ steps, so the total running time is $O(n)$ and thus is linear. Note that this running time is the best possible case, because n steps are necessary just to read the input.

Closing Notes:

- The complexity of A depends on the model of computation selected.
- In complexity theory, the choice of model affects the time complexity of languages.
- In complexity theory, we classify computational problems according to their time complexity.
- If our classification system isn't very sensitive to relatively small differences in complexity, the choice of deterministic model isn't crucial.

2.8 Complexity Relationships Between Models

2.8.1 Single and Multi-Tape Turing Machines

Theorem: Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

The idea behind the proof of this theorem is quite simple. We analyze the simulation of a multi-tape Turing machine on a single-tape Turing machine to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

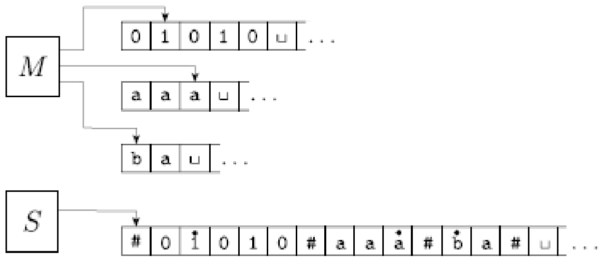


FIGURE 3.14
Representing three tapes with one

Let M be a k-tape TM that runs in $t(n)$ time. We construct a single-tape TM S that runs in $O(t^2(n))$ time.
Let Machine S operate by simulating M.

- **Configuration:** Initially, S puts its tape into the format that represents all the tapes of M and then simulates M's steps.
- To simulate one step, S scans all the information stored on its tape to determine the symbols under M's tape heads. Then it makes another pass over its tape to update the tape contents and head positions.

For each step of M, machine, S makes two passes over the active portion of its tape. The first obtains the information necessary to determine the next move and the second carries it out. A scan of the active portion of S's tape uses $O(t(n))$ steps.

To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts.

The initial stage, where S puts its tape into the proper format uses $O(n)$ steps. Afterwards, S simulates each of the $t(n)$ steps of M, using $O(t(n))$ steps, so this part of the simulation uses $t(n) \times O(t(n)) = O(t^2(n))$ steps
Therefore, the entire simulation of M uses $O(n) + O(t^2(n))$ steps.

We have assumed that $t(n) \geq n$ because M could not even read the entire input in less time. Thus, the running time of S is $O(t^2(n))$ and the proof is complete.

Note: This does not guarantee the best possible running time. Rather, it is an upper bound that we work with.

2.8.2 Deterministic and Nondeterministic Turing Machines

We show that any language that is decidable on such a machine is decidable on a deterministic single-tape Turing machine that requires significantly more time.

Recall that a nondeterministic Turing machine is a decider if all its computation branches halt on all inputs.

Let N be a nondeterministic Turing machine that is a decider. The running time of N is the function $f : N \to N$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

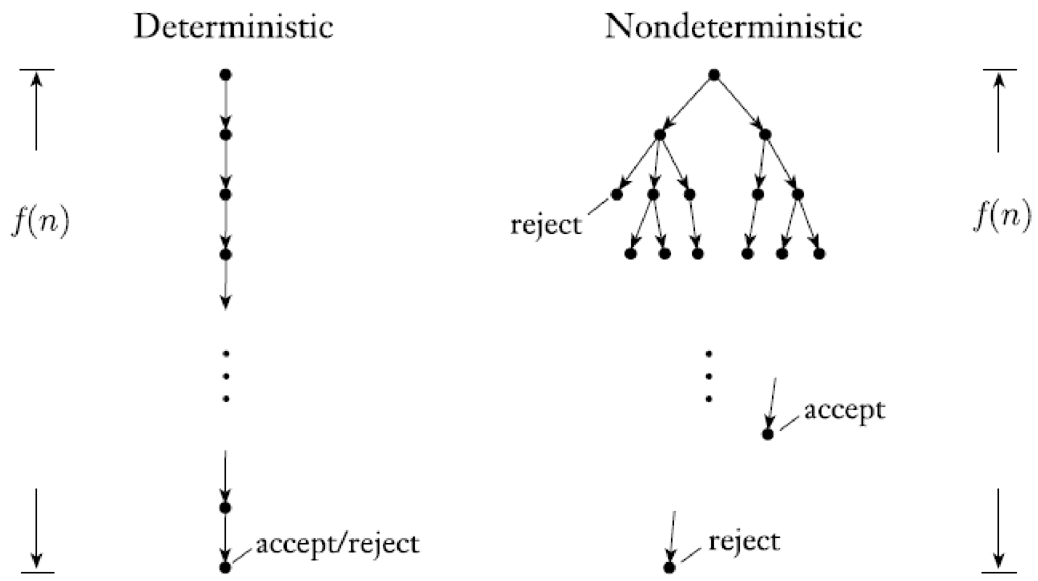


FIGURE 7.10
Measuring deterministic and nondeterministic time

Now that we have a definition for the running time of a nondeterministic Turing machine, and we know that for every nondeterministic Turing machine, an equivalent deterministic Turing machine exists. Again, we analyze the simulation a nondeterministic Turing machine using a deterministic Turing machine to prove the following theorem.

Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(n)}$ time deterministic single-tape Turing machine.

Proof:

Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N as in the proof of Theorem 3.16 by searching N's nondeterministic computation tree.

- On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$.
- Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function.
- The total number of leaves in the tree is at most $b^{t(n)}$
- The total number of nodes in the tree is less than twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$.

The time it takes to start from the root and travel down to a node is $O(t(n))$. Therefore, the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$

As described, TM D has three tapes. Converting to a single-tape TM at most squares the running time by Theorem 7.8.

Thus, the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$

3 Week 3: P and NP Classes

Recall from the previous class that:

- There is at most a square or polynomial difference between the time complexity of problems measured on deterministic single-tape and multitape Turing machines.
- There is at most an exponential difference between the time complexity of problems on deterministic and nondeterministic Turing machines.

3.1 Prerequisites

3.1.1 Instances

- We will consider an instance to be a number in binary. That is, more or less, $\log_2(n)$ in digits.
- Given n vertices and m edges of a graph, the instance size would be $m \times \log_2(n)$

3.1.2 Encoding

- A reasonable method is one that allows for polynomial time encoding and decoding of objects into natural internal representations or into other reasonable encodings.
- One reasonable encoding of a graph is a list of its nodes and edges.
- Another is the adjacency matrix, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not.
- When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation. In reasonable graph representations, the size of the representation is a polynomial in the number of nodes.

3.2 Introduction

- Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large.
- Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms are rarely useful.
- Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called brute-force search. Sometimes brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility.
- All reasonable deterministic computational models are polynomially equivalent. That is, any of them can simulate another with only a polynomial increase in running time.
- Reasonable $\rightarrow$ A notion broad enough to include models that closely approximate running times on actual computers.
- From here on, we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time.

3.3 Polynomial Time

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.

In other words, $P = \cup_k TIME(n^k)$.

The class P plays a central role in our theory and is important because:

- P is invariant (unchanging) for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine.
  - Mathematically robust; unaffected by particulars of the model of computation that we are using.
- P roughly corresponds to the class of problems that are realistically solvable in practice.
  - Relevant from a practical standpoint
  - For n^k, whether it is practical depends on k and on the application.

3.4 Examples of Problems in P

- When we present a polynomial time algorithm, we give a high-level description of it without reference to features of a particular computational model.
- We continue to describe algorithms with numbered stages
- We must be sensitive to the number of Turing machine steps required to implement each stage, as well as to the total number of stages that the algorithm uses.

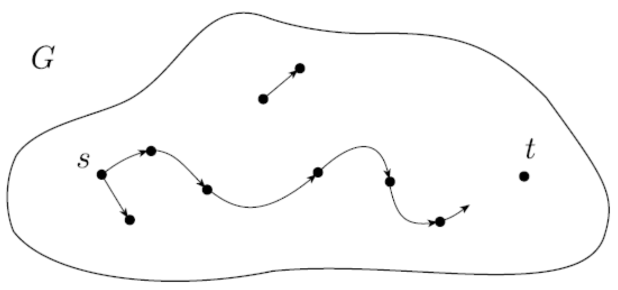
Proving that an Algorithm is in Polynomial Time:

1. Give a polynomial upper bound (in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n.
2. Examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. This will demonstrate that it runs for a polynomial number of stages, each of which can be done in polynomial time, and thus, the composition of polynomials is a polynomial.

3.4.1 Exercise 1: Path Problem

The $PATH$ problem is to determine whether a directed path exists from s to t.

$PATH = \{\langle G, s, t\rangle | G$ is a directed graph that has a directed path from s to t $\}$.



<u>Proof Idea:</u>

- We must present a polynomial time algorithm that decides PATH. One way is to use a graph-searching method such as breadth-first search (BFS).
- We successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time of this strategy by a polynomial is easy.

PROOF A polynomial time algorithm M for $PATH$ operates as follows.

```
M = "On input ⟨G, s, t⟩, where G is a directed graph with nodes s and t:
    1. Place a mark on node s.
    2. Repeat the following until no additional nodes are marked:
    3.    Scan all the edges of G. If an edge (a, b) is found going from
          a marked node a to an unmarked node b, mark node b.
    4. If t is marked, accept. Otherwise, reject."
```

Analysis: Proving a Polynomial Running Time

Polynomial Upper-Bound on the Number of Stages

- Stages 1 and 4 only run once.
- Stage 3 runs at most m times, because each time except the last, it marks an additional node in G.

Thus, the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.

Analysis of the Stages

- Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model.
- Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which is also easily implemented in polynomial time.

Hence, M is a polynomial time algorithm for $PATH$.

3.4.2 Exercise 2: Relatively Prime

Let $RELPRIME$ be the problem of testing whether two numbers are relatively prime.

$RELPRIME = \{\langle x, y\rangle |$ x and y are relatively prime.$\}$

We say that two numbers are relatively prime if 1 is the largest integer that evenly divides them both.

Solution: The Euclidean Algorithm

Assume $x > y$:

1. Divide x by y and get the remainder r.

   $$y = x$$
   $$r = y$$

2. Repeat until $y = 0$

- An instance of $RELPRIME$ has a length of $\log_2(x) + \log_2(y)$.
- $POLY(\log x + \log y) = O((\log x + \log y)^k)$ for some k.

3.5 Class NP

3.5.1 Introduction

Attempts to avoid brute force in certain other problems haven't been successful. Polynomial time algorithms that solve them aren't known to exist. We don't know whether these algorithms are undiscovered, or whether they simply cannot be solved in polynomial time.

3.5.2 Example: Hamiltonian Path

- A Hamiltonian path in a directed graph G is a directed path that goes through each node **exactly once.**
- We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes.
- We can easily obtain an exponential time algorithm for the $HAMPATH$ problem.
- No one knows whether $HAMPATH$ is solvable in polynomial time.

$HAMPATH = \{\langle G, s, t\rangle|$ G is a directed graph with a Hamiltonian path from s to t.$\}$

3.6 Polynomial Verifiability

- The $HAMPATH$ problem has a feature called polynomial verifiability.
- Verifying the existence of a Hamiltonian path may be much easier than determining its existence.
- Some problems may not be polynomially verifiable. For example, take $\overline{HAMPATH}$. Even if we could determine that a graph didn't have a Hamiltonian path, we don't know of a way for someone else to verify its non-existence without using the same exponential time algorithm for making the determination.

A verifier for a language A is an algorithm V where:
$A = \{w |$ V accepts $\langle w, c\rangle$ for some string $c\}$

We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w.

A language A is polynomially verifiable if it has a polynomial time verifier.

- The verifier uses additional information represented by the symbol c to verify that a string w is a member of A.
- The certificate has a polynomial length (in the length of w) because that is all the verifier can access in its time bound.

For the $HAMPATH$ problem, a certificate for a string $\langle G, s, t\rangle \in HAMPATH$ is a Hamiltonian path from s to t. The verifier can check in polynomial time that the input is in the language when it is given the certificate.

NP is the class of languages that have polynomial time verifiers.

3.7 Alternative NP Definitions

Recall: NP stands for nondeterministic polynomial time.

Definition 1 NP is the class of languages that have polynomial time verifiers.

Definition 2 NP is the class of languages that can be solved in polynomial time using a nondeterministic Turing machine.

We can connect the ideas of Definition 1 and 2 by proving the theorem: A language is in NP if and only if it is decided by some nondeterministic polynomial time turing machine.

Proof Idea: We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa.

- The NTM simulates the verifier by guessing the certificate.
- The verifier simulates the NTM by using the accepting branch as the certificate.

Direction 1:
N = "On input w of length n

1. Nondeterministically select string c of length at most n^k.
2. Run the polynomial time verifier V on input $\langle w, c\rangle$
3. If the polynomial time verifier v accepts, accept. Otherwise, reject."

Direction 2:
V = "On input $\langle w, c\rangle$, where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step.
2. If this branch of N's computation accepts, accept. Otherwise, reject.

3.8 Examples of Problems in NP

3.8.1 Satisfiability Problem

Prerequisite Knowledge

- Boolean variables are variables that take on the values true or false.
- Boolean Operators $(\wedge, \vee, \neg)$
- A Boolean formula is an expression involving boolean variables and operations.
- A boolean formula is satisfiable if for some assignment of true and false to the variables, it makes the formula evaluate to true.

The satisfiability problem can be defined as:
$SAT = \{\phi |\ \phi$ is a satisfiable Boolean formula.$\}$

Special Forms:

- A clause is a set of literals connected by $\vee$
- A boolean formula is in conjunctive normal form (cnf-form) if it is compromised by one or more clauses, connected by $\wedge$s.
- A formula is 3-cnf if every clause has 3 literals, Based on this, we can define the variation of the SAT problem, being:

$$3SAT = \{\phi | \ \phi \text{ is a satisfiable 3cnf-formula } \}$$

Do note that we can put forms into 3cnf as follows:

$$\begin{aligned} & x \vee y \vee z \vee t \\ & = (x \vee y \vee a) \wedge (\overline{a} \vee z \vee t) \end{aligned}$$

We can prove that $SAT \in NP$ by providing a verifier with a certificate! We can also give a nondeterminstic polynomial time Turing machine.

3.9 NP-Completeness

Theorem: $SAT \in P$ if and only if $P = NP$

NP Completeness means that if an NP-complete problem has a polynomial algorithm, then all NP problems are in P.

3.9.1 Polynomial Time Reducibility

In order to prove the former theorem, we need to have a way to relate SAT to all other NP problems. We make comparisons by using Polynomial Time Reducibility.

What we do is reduce a problem A to a problem B. This means that we change any input to A into an input for B in such a way that a solution for B can be used to solve A.

A language A is polynomial time many-one reducible to a language B. We write $A \leq_p B$, if a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every input w, $w \in A \leq f(w) \in B$.

The function f is called the polynomial time reduction of A to B.

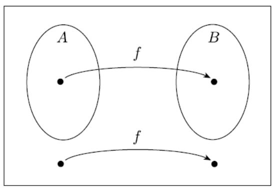


FIGURE 7.30
Polynomial time function f reducing A to B

3.9.2 Example: Clique

A k-clique is a clique that contains k nodes. The problem is defined as follows:

$CLIQUE = \{(G, k) |\ G$ is an undirected graph with a k-clique $\}$

Theorem: 3SAT is polynomial time reducible to CLIQUE.

Proof: We start with a satisfiability formula with k clauses:
$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$

This formula contains n variables $x_1, \cdots, x_n$. They may appear in positive and negative literals. For example:
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$

We transform this into a graph:

- Every literal is a node
- The edges connect all nodes in different clauses except if they have contradictory labels, x_1 and $\overline{x_1}$, this means that they cannot be true at the same time.

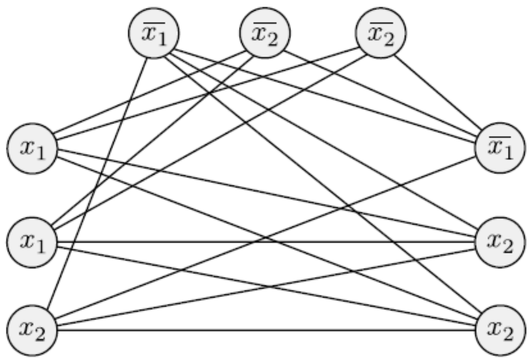


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

4 Week 4: NP Completeness

4.1 Definition of NP Completeness

A language is **NP-complete** if it satisfies two conditions:

1. B is in NP
2. Every A in NP is polynomial time reducible to B

Polynomial Time Reductions Compose....

That is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C.

For access to the remaining notes, please contact me over LinkedIn or GitHub.