The main purpose of these notes is to act as a supplement to Book & Lecture Notes #14. There are some differences between 3rd Edition of the book and Lecture #14. I will highlight these as well.

So far, in this course, we have described many algorithms — most of these are polynomially bounded — the one exception being KNAPSACK whose algorithm complexity is $O(nW)$ where $W$ is knapsack capacity — we will discuss this later.

However, most problems from the "real world" seem to be much harder. This discussion is about how to show that some of these are harder. It is an extension of our discussion on lower bounds.

For the same "application" there are three types of problems that we encounter — these are related many times:

1) **Optimization Problem**: Ex: Find min $\sum_{e \in T} w(e)$
   - $T$: Spanning tree in $(G, w)$.

2) **Decision Problem**: Given $(G, w)$, is there a $\text{sp}$-tree $T$ such that $\sum_{e \in T} w(e) \leq k$? | Answer: Yes or No (or False)
(3) Search Problem: Given a yes answer to (2), produce such a sp. tree.
(If the answer is No, nothing to produce.)

In many cases (2) is central; if we have an Alg. that solves (2), we can use "Binary Search" on \( K \) for (1); By trying to check whether or not each edge is "essential", we solve (3). For example: Suppose answer to (2) for \( K^* \) is yes, we can remove an edge and do (2) again. If the answer is still yes, this edge is not essential & can be removed.

By repeating this a poly. # of times, on (2) we can answer (3). So we concentrate from now on (2).

Warning: [This has limitations when we try this in Game Theory]

Now comes some "abstract" stuff; we will try to give examples to "soften the blow."

See also CLRS (Ch34) and Lect #14.
Def 1. An "abstract" problem \( Q \) is a binary relation on a set \( I \) of instances of the problem and a set \( S \) of solutions. If \( s \in S \) is a solution to instance \( i \in I \), then \( r(i,s) = 1 \) (the relation holds for this pair).

Remark. There may be 0 or many solutions for a given instance.

Example: SHORTEST-PATH is a problem.
Instance: \((G, w), s, t\)
Solution: a path from \( s \) to \( t \) in \((G, w)\) of min. length.

Def 2. A decision problem is one when \( S = \{0, 1\} \cup \{\text{Yes, No}\} \cup \{\text{True, False}\} \)

i.e. \( |S| = 2 \).

Encodings: A set \( S \) of abstract objects is mapped by a mapping \( S \to \{0, 1\}^* \)

binary strings: \([A \text{ set of finite length strings each of whose elements are in } \{0, 1\}]\)

Def 3. A problem whose set of instances is a set of binary strings is called a concrete problem (as opposed to abstract problem).
Def: We say that an algorithm solves a concrete problem in $O(T(n))$ time, if when it is provided an input instance $i$ whose length $|i| = n$, the algorithm takes $O(T(n))$ time.

At most, if in the above $T(n) = O(n^k)$ (a fixed) then we say that the problem is polynomial-time solvable (poly time for short).

Complexity Classes:

- $P$: the class of concrete decision (set of sets) problems that are poly-time solvable.

Formal Language Approach

Comes from Automata Theory (a theory of computation) course at the U/G level.

We begin with a finite set $\Sigma$ called the alphabet. Mostly, we use $\Sigma = \{0,1\}$.

A language $L$ is a set of strings (of finite length) made up of symbols in $\Sigma$. $\epsilon$ represents empty string. $\phi$ represents empty language.
\[ \Sigma^* \text{ is the universal set which is the set of all strings (of finite length) over } \Sigma. \]

Hence \( L \subseteq \Sigma^* \).

Since \( L \) is a set, we can do set operations:

- **Union:** \( L_1 \cup L_2 = \{ x \in \Sigma^* : (x \in L_1) \text{ or } (x \in L_2) \text{ or both} \} \)
- **Intersection:** \( L_1 \cap L_2 = \{ x \in \Sigma^* : (x \in L_1) \text{ and } (x \in L_2) \} \)
- **Complement:** \( \overline{L} = \Sigma^* - L = \{ x \in \Sigma^* : x \notin L \} \)

Since elements of \( L \) are strings, we have one more operation: **Concatenation**

\[ L_1 \cdot L_2 = \{ (x, y) : x \in L_1 \text{ and } y \in L_2 \} \]

**Closure** given \( L \), \( L^* = \{ \epsilon \} \cup L \cup \{ L \} \cup \{ L \} \cup \{ L \} \cup \ldots \)

\[ \text{Kleene Star} \]

Where \( L^{(k)} = L \cdot L \cdot L \cdots L \) \( k \text{ times} \)

Now we turn to algorithms in this formal language approach.
Def: Algorithm $A$ accepts (rejects) a string $x \in \Sigma^*$ if given input $x$, the algorithm outputs $A(x) = \mathbb{1}(0)$.

Note: Here $\Sigma^* = \{0,1\}^*$; $\Sigma = \{0,1\}$.

[$1/0$ may be replaced by True/False; yes/no]

Def: The language accepted by an algorithm $A$ is the set $L = \{ x \in \Sigma^* : A(x) = \mathbb{1} \}$.

Such an algorithm that accepts $x \in L$ may not reject $x \notin L$. [This is because the algorithm may not stop on some strings] e.g. Halting Problem.

Def: A language $L$ is decided by an algorithm $A$ if and only if $x \in L \Rightarrow A(x) = \mathbb{1}$ & $x \notin L \Rightarrow A(x) = 0$.

(Such an algorithm must necessarily stop and output 0/1 on each string in $\Sigma^*$)

Def: A language $L$ is accepted in poly-time by an algorithm $A$ if for any input string $x \in L$ of length $n$, algorithm accepts $x$ in $O(n^k)$ time for fixed $k$.

$P = \{ L : \exists$ alg. $A$ that decides $L$ in poly-time $\}$

$\exists$ Need Proof

$\exists$ L: $\exists$ alg. $A$ that accepts $L$
The last remark in previous page needs proof and it is given below.

Poly time alg that decides \( L \) clearly accepts \( x \in L \). It is the converse that needs proof.

Suppose \( \exists \) alg. \( A \) that accepts \( x \in L \) in time \( \leq c \cdot n^k \) where \( n = \text{length of } x \).

Then, given any \( x \), run alg. \( A \) until one of the following happens.

(i) alg. stops in time \( \leq c \cdot n^k \), and if it does not put \( A(x) = 1 \Rightarrow x \in L \)

\[ A(x) = 0 \Rightarrow x \not\in L \] Hence \( A \) decides in this case.

(ii) alg. continues without any output beyond \( c \cdot n^k \) steps. In this case, our process stops the algorithm in a "force-quit" manner and outputs \( A(x) = 0 \Rightarrow x \not\in L \).

This modification produces an alg. \( A' \) that decides \( L \) in poly time.

Hence the result.
NP (Non-deterministic Polynomial)

Poly-time Verification:
The main idea here is: If I know the answer, how can I convince you of this answer with a small amount of additional information beyond the input? You can use a poly-time alg. to convince yourself with this added information from me (called a Verification Certificate) using a Verification algorithm, which uses two inputs: Original input & my Verification Certificate.

Def: A verification algorithm is a two input alg. A where one input is a binary string $x \in \Sigma^*$ (original input) and the other is a string $y$ (which may depend on $x$) such that $A(x,y) = 1$. The language verified by Alg. A is $L = \{ x \in \Sigma^* : \exists y \in \Sigma^* \text{ such that } A(x,y) = 1 \}$.

Hence A verifies each $x \in L$ for which $\exists$ a Certificate $y$ such that $A(x,y) = 1$. Also, if $x \notin L$ there should not exist $y$ such that $A(x,y) = 1$. 
Class \( \text{NP} \):

\( L \in \text{NP} \) if \( L \) can be verified in poly-time

\[ L = \left\{ x \in \Sigma^* : \exists y \in \Sigma^*, \, |y| = O(|x|^C) \text{ and } \right\} \]

\[ \text{if poly-time alg A such that } \]

\[ A(x, y) = 1 \]

\( C \) is a constant \( \implies \) Certificate should not be too big.

Alg. \( A \) is said to verify \( L \) in poly-time.

Lemma: \( L \in \text{P} \implies L \in \text{NP}. \iff \text{P} \subseteq \text{NP} \)

Note: For \( x \notin L \) we do not have to prove that \( x \notin L \). On the other hand, we should not “prove” \( x \in L \) if indeed \( x \notin L \). (False Certificate)

Open Question: \( I \Rightarrow \text{P} \neq \text{NP} \) (Most important question of our time, other than finding a vaccine or cure for COVID-19)

\[ 1. \quad L \in \text{P} \implies \overline{L} \in \text{NP} \quad \overline{L} \in \text{co-NP} \]
NP-Completeness and Reducibility

Def: Language \( L_1 \) is poly-time reducible to language \( L_2 \) (written: \( L_1 \leq_p L_2 \)) if \( \exists \) a poly-time \underline{computable function} (from automata theory)

\[ f : \sum^* \rightarrow \sum^* \text{ such that for all } x \in \sum^* \]
\[ [x \in L_1] \iff [f(x) \in L_2] \]

\( f \) is called \underline{reduction function} and the poly-time alg. that computes it is called the \underline{reduction algorithm}.

The idea here is if we want to know whether \( x \in L_1 \), we instead compute \( f(x) \) and check if \( f(x) \in L_2 \).

This is \underline{transforming} one problem to another for which we have (or don't have) an alg. This includes what we called \underline{formulation before any Flow}.
Lemma 4: \[ L_1 \leq_p L_2 \land L_2 \in P \Rightarrow L_1 \in P \]

Good part

A reduction: given \( x \rightarrow f(x) \) polytime, check \( x \in L_1 \) or not \( f(x) \in L_2 \) in polytime.

N.P. Completeness

A language \( L \) is NP-Complete (NPC for short) if
\[ L \in NP \]
\[ \forall L' \in NP, L \leq_p L' \]

[If we only prove (ii), \( L \) is said to be NP-hard]

Trying to show (ii) for each new problem \( \text{is TOO HARD} \): But we have a tool:

Lemma 5: \[ L_1 \leq_p L_2 \land L_2 \leq_p L_3 \Rightarrow (L_1 \leq_p L_3) \]

Proof: Exercise for you.

\[ \therefore \leq_p \text{ is transitive relation.} \]

Hence if we know \( L_1 \) is NP-Complete & \( L_1 \leq_p L_2 \) then \( L_2 \) is NP-Complete

Proof: Exercise for you.
But there are 2 issues now.

1) How was the very first NPC problem shown to be NPC?
   (We cannot use our trick here)
   This was done by S.A. Cook using Nondeterministic Machines and Circuit Satisfiability problem
   (A description is in your book & Lect #14)
   I will not repeat it here.

2) If I want to show $L \in NPC$, and there are 1000's of known NPC problems, which problem should I use in showing $\exists p \in NPC$?

There are no easy answers as we shall see in the next page.

There are a few problems which we will do for this purpose.
Here is an outline of what we do:

- **Circuit-SAT**
  - **SAT**
    - **3-SAT**
      - **CLIQUE**
        - **Vertex-Cover**
          - **HAM-Cycle**
            - **TSP**

We will do these for sure.

- **SUBSET-SVM**
  - **Lect #14**
    - Second Edition
      - Will not do