Lecture 1: Intro

Classify problems according to the computational resources required to solve them. In particular, we will consider:
- running time
- memory/space
- parallelism
- randomness
- rounds of interaction (2 parties communicating)

⇒ What can we compute with a limited set of resources?

⇒ Decidability

What is computable (ignoring restrictions on resources)?

For example, here are some of the big open questions in the field (and the conjectured answers to them).

Conjecture

No * \( P \neq \text{NP} \) : is finding a solution more difficult than recognizing/checking one.

No * \( P \neq \text{NC} \) : is every efficient algorithm parallelizable?

No * \( P \neq \text{L} \) : Can every efficient algorithm be turned into one that uses a very small amount of memory?

No * \( \text{EXP} \nsubseteq \text{P/poly} \) : Are there small (polysize) boolean circuits for all problems that can be computed in polynomial time?

Yes * \( P \subseteq \text{BPP} \) : Can every efficient randomized algorithm be converted to a deterministic one?

If any of these conjectures is wrong, it will have a big impact on computation.
We will start with the following groundwork:

Problems + Languages
Complexity Classes
Turing Machines
Reductions
Completeness

We need a formal definition of a "computational problem."

Informally we say:

- Given a graph \( G \) with vertices \( s \) and \( t \), find the shortest path from \( s \) to \( t \) in \( G \).
- Given matrices \( A \) and \( B \), compute \( AB \).
- Given an integer \( N \), find its prime factors.
- Given a Boolean formula, find a satisfying assignment.

How to encode the inputs to these problems?

\( \Sigma \) : finite alphabet \( \{0, 1\} \) or \( \{0, 1, 2, \ldots, q \} \)

Inputs are encoded in strings over the alphabet. This is done routinely in computer science. We will usually not worry too much about the details of the encoding task, but there needs to be an agreed upon interpretation of strings.

It is not necessary for every string to represent a valid problem instance. The specific encoding will effect finer grained analysis (e.g., adjacency matrix vs. adjacency list). We will mostly be concerned with higher level analysis.

Given an encoding a problem can be expressed formally as a function from strings to strings: \( f: \Sigma^* \rightarrow \Sigma^* \)
given $x$, compute $f(x)$.

For a decision problem, we have $f : \Sigma^* \rightarrow \{\text{yes}, \text{no}\}$.

given $x$, accept (yes) or reject (no).

We will work mostly with decision problems.
This simplification usually doesn't give up too much.

Given an algorithm to solve a decision problem, it can often be used to solve a related optimality/search problem without too much additional overhead.

Given $n+k$ : is there a factor of $n < k$? (Decision).

Can use binary search to solve:
Given $n$, find its prime factors.

Given a Boolean formula $\phi(x_1, \ldots, x_n)$ is it satisfiable (Decision)
Given $\phi(x_1, \ldots, x_n)$ find a satisfying assignment.
First see if $\phi(x_1, \ldots, x_n)$ is satisfiable: If NO → STOP.
Is $\phi(T, x_2, \ldots, x_n)$ satisfiable?
Yes: fix $x_1 \leftarrow T$
No: fix $x_1 \leftarrow F$
continue on to $x_2, x_3, \ldots, x_n$.

We will view decision problems as a language. (i.e. set).
This requires a fixed encoding of problem instances in $\Sigma^*$.

Language $L$ is subset of $\Sigma^*$ corresponding to "yes" instances.
e.g. \((n,k)\) s.t. \(n\) has a factor \(< k\),
  - all satisfiable boolean formulas.

Decision problem: Given \(x\), is \(x \in L\)?

Strings which are not valid encodings are treated as
no instances.

\[ \Sigma^+ \]

\[
\begin{array}{c}
\text{yes} \\
\text{no}
\end{array}
\]

Invalid inputs.

Complement of \(L\) \(\text{co-} L\)

\[ \Sigma^+ - \bar{\Sigma} \]

Invalid inputs.

So far, languages are just a set-theoretic definition.
No reference to computation yet.

Complexity class: Class of languages.
Typically, these are defined by a computational constraint.

\(P = \) set of all languages decidable in polynomial time.
\(NP = \) set of \(L\) where
\[ L = \{ x \mid \exists y \; |y| \leq |x|^k, \; (x,y) \in R \} \]
\(R\) is a language in \(P\).

Meaningful complexity classes will have certain properties:
* Capture genuine computational phenomenon (e.g., parallelism).
* Contain natural + relevant problems.
characterized by natural problems (complexity).
* robust under various models of computation.
* possibly closed under operations s.t. \( \times, \vee, \exists \)

To define a complexity class, we need a model of computation. We want this model to yield complexity classes that capture important aspects of computation.

We will use a Turing Machine for this:

```
ab|b|c|c|a|0|0|... one alphabet letter per cell.
```

Finite Control

read/write head

Turing Machine

\( Q \): finite set of states.
\( \Sigma \): finite alphabet including blank character \( \$ \)

\( q_0, \text{start} \), \text{start} \ text{special} \ text{states} \ text{in} \( Q \).
Transition function \( \delta \): \( Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, S\} \)

\[
\delta(q, a) = (q', b, R) \quad (- \text{ means stay in same place}).
\]

Start Configuration:

```
Encoding of input  infinite sequence \( \omega \)
0 1 0 1 1 0 ... \( \omega \)
```

\( q_0 \), start.
Sequence of steps specified by 8.
if \( q_{\text{acc}} \) or \( q_{\text{ rej}} \) are reached, then halt.
(no outgoing transitions from these states)

3 ways for a Turing Machine to Compute
in all input \( x \) written on tape.

1. Function Computation: output \( f(x) \) is left on tape when TM halts.
2. Language decision: TM halts in state \( q_{\text{acc}} \) if \( x \in L \). TM halts in state \( q_{\text{ rej}} \) if \( x \notin L \)
3. Language recognition: TM halts in state \( q_{\text{acc}} \) if \( x \in L \); may loop forever otherwise.

TM Example. Is \( x \) a palindrome? (\( x = x^R \)).

\( Q = \{ q_{\text{acc}}, q_{\text{ rej}} \} \)
\( \Sigma = \{ 0, 1, \_ \} \)

\[
\begin{array}{c}
\text{State} & \text{Input} & \text{Write} & \text{Move} \\
q_{\text{acc}}, 0 & \_ & q_0 & R \\
q_{\text{ rej}}, 1 & \_ & q_1 & R \\
q_{\text{ acc}}, \_ & \_ & q_{\text{ acc}} & \_ \\
q_0, 0 & 0 & q_0 & R \\
q_0, 1 & 1 & q_1 & R \\
q_0, \_ & \_ & q_1 & L \\
q_1, 0 & 0 & q_{\text{ rej}} & \_ \\
q_1, 1 & 1 & q_{\text{ rej}} & \_ \\
\end{array}
\]

Same for \( q_{\text{ rej}} \)
\( q_1, 1 \to \_ \_ \_ C. L \)

\( q_1, 0 \to \_ \_ \_ C. L \)

\( \_ \_ \_ 1 \to \_ \_ \_ q_{\text{ rej}} \)

\( \_ \_ \_ 0 \to \_ \_ \_ q_{\text{ rej}} \)
$q_0 \quad 0 \quad q_1 \quad L \quad q_2 \quad \Sigma \quad q_3$
$q_4 \quad 1 \quad q_5 \quad L \quad q_6 \quad \Sigma \quad \delta_{q_5}$
$q_7 \quad \Sigma \quad q_8 \quad \Sigma$

$T M$ is very robust to variation:

For example

\[
\begin{array}{c}
\text{finite \ terminal} \\
q \downarrow \\
\vdots \\
\end{array}
\]

Usually one read-only to hold input
one write-only to write output
$k-2$ read/write work tapes.

Can simulate a $k$-tape $T M$ by a single tape $T M$

\[
\begin{array}{c}
a \ b \ a \ b \ \ldots \\
| \\
\ \ \\
| \\
\ v \\
\ \\
\ b \ c \ d \ \ldots \\
\end{array}
\]

$M \quad k$-tape $T M \rightarrow M' \quad 1$-tape $T M$

One step of $M$ \rightarrow head scans the entire length of the tape remembering each symbol at each head. Then scans back to implement the step on each tape.
M takes $T$ steps on input $x$.
$M'$ takes $O(T^2)$ steps on input $x$.

If a # is hit then the contents of the tape are moved over to make room.

When $M$ halts: erase everything except the output string.

---

Turing Machines are clumsy as computation but they are a simple abstract model that allows us to formalize our intuitive notion of what it means to compute efficiently.

A TM $M$ computes a language $L$ in time $t(n)$ if

\[ \forall x \text{ on input } x \quad M \text{ "halts" (i.e. reaches } q_{acc} \text{ or } q_{ rej} \text{) in at most } t(|x|) \text{ steps.} \]

\[ x \in L \iff M \text{ accepts } x. \]

The "Extended" Church-Turing Thesis

Everything that can be computed in time $t(n)$ on a physical computing device, can be computed in time $[t(n)]^{O(1)}$ on a Turing machine.

\[ \text{A polynomial slow-down.} \]

Quantum computers have been the only real challenge to this belief.

All physically realizable models of computation can be simulated by a TM w/ polynomial overhead.
Our first concern is what can be computed by a TM w/o any restrictions on running time.

L is decided by TM $M$ if

\[ \forall x \; M \text{ halts on input } x \text{ after a finite number of steps.} \]
\[ x \in L \iff M \text{ accepts } x. \]

There are natural problems (languages) that can not be decided by any TM. \( \Rightarrow \) Undecidable.

\[ \text{Halt} = \exists \langle M, x \rangle : M \text{ halts on input } x \]

Requires an encoding of every Turing Machine $M$ for example, use $\langle M \rangle$.

\[ \langle M \rangle \text{ will refer to an encoding of } M. \]
\[ \langle x \rangle \text{ refers to an encoding of } x. \]

**Theorem**: Halt is Undecidable.

If a TM (program) that will always finish in a finite amount of time and determines whether $\langle M, x \rangle \in \text{Halt}.$

**Proof by contradiction**: 

Fill in an infinite chart.

Rows + columns indexed by all strings in lexicographic order:

\[
\begin{array}{cccccc}
\sigma_1 & \sigma_2 & \sigma_3 & \sigma_4 & \cdots \\
0 & 1 & 00 & 01 & 10 & 11 & \cdots
\end{array}
\]

\[
\begin{array}{cccccc}
\sigma_1 & \sigma_2 & \sigma_3 & \sigma_4 & \cdots & \sigma_i \\
0 & 1 & 00 & 01 & 10 & 11 & \cdots
\end{array}
\]

If $\sigma_i$ is a valid encoding of a TM $M$, fill in the row as follows:

If $M$ halts on $(\sigma_i)$ $\langle M, \sigma_i \rangle \in \text{Yes}$

\[ \therefore \langle M, \sigma_i \rangle \in \text{No} \]
If $S_k$ doesn't make sense as an encoding of a TM, then fill in the row with all "N".

Every TM $M$ is encoded by some string $S_k$.

Now suppose there is some TM $H$ that decides HALT.

On input $(S_k, S_j)$

$M$ outputs answer in square $(S_k, S_j)$ in a finite number of steps.

New $H'$: on input $S_j$ run according to $H$ on input $(S_j, S_j)$

if about to enter state $qace$ → go to an infinite loop.
if about to enter state $qej$ → go to $qace$ instead.

Consider diagonal $S$ chart.

Consider diagonal $S$ chart.

on input $S_j$ $H'$ does the opposite of what it says in square $(S_j, S_j)$.

But $H'$ is encoded by some string $S$.

If $H'$ halts on $S_j$ $(S, S) = Y$ but then $H'$ loops forever in $S$.

If $H'$ loops for even on $S$ $(S, S) = N$.

$H'$ accepts $S$.

This proof is an example of "diagonalization".

$\Rightarrow$ Profound implications for program testing & verification.
Back to complexity classes:

\[ \text{TIME}(f(n)) \text{ for } f : N \rightarrow N. \]

= \text{the set of all languages decided by a multi-tape TM in at most } f(n) \text{ steps}

\[(n \text{ is the \# characters in the input).}\]

\[ \text{SPACE}(f(n)) \text{ set of all languages decided by a multi-tape TM and touches at most } f(n) \text{ tape squares.} \]

\[ P = \bigcup_{k \geq 1} \text{TIME}(n^k) \]

\[ \text{if } L \in P \Rightarrow L \in \text{TIME}(n^k) \text{ for some fixed } k \]

\[ \text{independent of the input.} \]

Goal:
1. Find an algorithm that decides \( L \)
2. Prove that no algorithm does better (use of resources).

Good C ② Hopeless ②

Instead: relate difficulty of problems to each other by reductions.
Show that certain problems are the “hardest” in a complexity class. (complete)

Powerful + useful surrogate for ②

Reductions: relating problems to each other.
Two languages $L_1 \preceq L_2$ \( \iff \) $L_1$ "reduces to" $L_2$.

If $f$ is an efficient (for non-polynomial time) algorithm that computes $f$ s.t.

\[
\begin{align*}
& x \in L_1 \implies f(x) \in L_2 \\
& x \notin L_1 \implies f(x) \notin L_2
\end{align*}
\]

If $L_1 \preceq L_2$ and $L_2 \notin P \implies L_1 \notin P$. $L_1$ is as easy as $L_2$.

If $L_1 \preceq L_2$ and $L_1 \notin P \implies L_2 \notin P$. $L_2$ is as hard as $L_1$.

**Example:** 3SAT $\preceq$ Independent Set.

3-SAT = $\exists \varphi$ : $\varphi$ is a 3-CNF formula that has a satisfying assignment?

3-CNF = 3 Conjunctive Normal Form.

AND of $m$ clauses: each clause OR $q \leq 3$ literals

ex: $(x_1 \lor x_7 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land \cdots$

**Independent Set**

**Input:** Graph $G$, integer $k$

Is there an independent set of size $k$?

\( \iff \) set of vertices, no two of which are connected.
Reduction Sketch:

Input: \( \Phi = (x \lor y \lor z) \land (x \lor z \lor y) \ldots \)

Output: \( G_\Phi, \ k = m = \# \text{ clauses in } \Phi \).

\( G_\Phi \): triangle for each clause in \( \Phi \) (or edge, vertex if \(<3 \text{ literals}\))

... add edge between each \( y, y' \) pair.

Inued set of size \( m \Rightarrow \) set \( \Phi \text{-signs} \).

one variable per triangle.

Let each literal chosen be true.

No contradictions because \( \Phi \text{-signs} \) edges.

\( \Phi \text{-signs} \Rightarrow \) set of true literals.

Corresponds to subset of nodes - at least one per triangle.

Pick one vertex per triangle - black edges not violated.

Completeness complexity class \( C \).

Language \( L \) is \( \mathcal{C} \)-complete if

a) \( L \in \mathcal{C} \)

b) \( \forall \mathcal{L}' \subseteq \mathcal{C}, \mathcal{L}' \subseteq L \).

\( L \) is the hardest problem in complexity class \( \mathcal{C} \).

This allows us to reason about an entire class by thinking only about a single concrete problem.
How to reduce every language in $C \leq L$?
- Show $L \preceq L'$ for some $L'$ that is $C$-complete.
- Hard to find the first $C$-complete problem.

For example:

$$NP = \text{ set of languages } L \text{ s.t. } \exists k, R \in P \ni L = \{ x \mid \exists y \mid |y| = |x|^k, (x,y) \in R \}$$

Cook in 1971 showed that SAT (boolean satisfiability) is $NP$-complete.

Karp in 1972 many other important problems in CS are also $NP$-complete.

**Recap:** Here are the basic ideas we have so far:

- Formal definition of problems
  - Functions, decision
  - Language = set of strings.
- Complexity class = set of languages
- Efficient computation = efficient computation on a Turing Machine.
  - Single-tape, multi-tape.
  - Diagonalization technique
    - $HALT$ is undecidable.
- Time and Space classes.
- Reductions.
- C-completeness, C-hardness.