Complexity Theory

Classify problems according to the **computational resources** required
- running time
- storage space
- parallelism
- randomness
- rounds of interaction, communication, others…

**Attempt to answer:** what is **computationally feasible** with **limited resources**?
Complexity Theory

• Contrast with decidability: What is computable?
  – answer: some things are not

• We care about resources!
  – leads to many more subtle questions
  – fundamental open problems
The central questions

• Is finding a solution as easy as recognizing one?
  \( P = NP? \)
• Is every efficient sequential algorithm parallelizable?
  \( P = NC? \)
• Can every efficient algorithm be converted into one that uses a tiny amount of memory?
  \( P = L? \)
• Are there small Boolean circuits for all problems that require exponential running time?
  \( \text{EXP} \subset P/\text{poly}? \)
• Can every efficient randomized algorithm be converted into a deterministic algorithm one?
  \( P = \text{BPP}? \)
Central Questions

We think we know the answers to all of these questions …

… but no one has been able to prove that even a small part of this “world-view” is correct.

If we’re wrong on any one of these then computer science will change dramatically
Introduction

• You already know about two complexity classes
  – \( P \) = the set of problems decidable in \textit{polynomial time}
  – \( NP \) = the set of problems with witnesses that can be checked in polynomial time

... and notion of \textit{NP-completeness}

• Useful \textbf{tool}

• Deep \textbf{mathematical problem}: \( P = NP \)?

  Course should be \textbf{both} useful \textbf{and} mathematically interesting
A question

• Given: polynomial $f(x_1, x_2, \ldots, x_n)$ as arithmetic formula (fan-out 1):

  $- x_1 x_2 \ldots x_n$
  
  • multiplication (fan-in 2)
  • addition (fan-in 2)
  • negation (fan-in 1)

• Question: is $f$ identically zero?
A question

• **Given**: multivariate polynomial
  \[ f(x_1, x_2, \ldots, x_n) \]
  as an arithmetic formula.

• **Question**: is \( f \) identically zero?

• **Challenge**: devise a deterministic polytime algorithm for this problem.
A randomized algorithm

- **Given**: multivariate degree r poly. \( f(x_1, x_2, \ldots, x_d) \)
  note: \( r = \deg(f) \leq \text{size of formula} \)

- **Algorithm**:
  - pick small number of random points
  - if \( f \) is zero on all of these points, answer “yes”
  - otherwise answer “no”

(low-degree non-zero polynomial evaluates to zero on only a small fraction of its domain)

- **No efficient deterministic algorithm known**
Derandomization

• Here is a deterministic algorithm that works under the assumption that there exist hard problems, say SAT.

• solve SAT on all instances of length $\log n$

  1 1 0 0 1 1 1 0 0 1

• encode using error-correcting code (variant of a Reed-Muller code)

  1 1 0 0 1 1 1 0 0 1 1 1 0 0 1
Derandomization

- run randomized alg. using these strings in place of random evaluation points
  - if $f$ is zero on all of these points, answer “yes”
  - otherwise answer “no”
- This works. (proof in this course)
Derandomization

This technique works on any randomized algorithm.

Gives generic “derandomization” of randomized procedures.
A surprising fact

- Is finding a solution as easy as recognizing one?
  \[ P = NP? \] probably FALSE

- Is every sequential algorithm parallelizable?
  \[ P = NC? \] probably FALSE

- Can every efficient algorithm be converted into one that uses a tiny amount of memory?
  \[ P = L? \] probably FALSE

- Are there small Boolean circuits for all problems that require exponential running time?
  \[ \text{EXP} \subseteq P/\text{poly}? \] probably FALSE

- Can every randomized algorithm be converted into a deterministic algorithm one?
  \[ P = \text{BPP}? \] probably TRUE
Outline

Should be mostly review…

1. Problems and Languages
2. Complexity Classes
3. Turing Machines
4. Reductions
5. Completeness
Problems and Languages

• Need formal notion of “computational problem”. Examples:
  – Given graph G, vertices s, t, find the shortest path from s to t
  – Given matrices A and B, compute AB
  – Given an integer, find its prime factors
  – Given a Boolean formula, find a satisfying assignment
Problems and Languages

• One possibility: function from strings to strings

\[ \begin{align*}
  f &: \Sigma^* \rightarrow \Sigma^* \\
\end{align*} \]

• function problem:
  
  given \( x \), compute \( f(x) \)

• decision problem: \( f: \Sigma^* \rightarrow \{\text{yes, no}\} \)
  
  given \( x \), accept or reject
Problems and Languages

• simplification doesn’t give up much:
  – Given an integer $n$, find its prime factors
  – Given an integer $n$ and an integer $k$, is there a factor of $n$ that is $< k$?
  – Given a Boolean formula, find a satisfying assignment
  – Given a Boolean formula, is it satisfiable?

• solve function problem efficiently using related decision problem (how?)

• We will work mostly with decision problems
Problems and Languages

• decision problems: \( f: \Sigma^* \rightarrow \{\text{yes, no}\} \)
• equivalent notion: language \( L \subseteq \Sigma^* \)
  \( L = \) set of “yes” instances

• Examples:
  – set of strings encoding satisfiable formulas
  – set of strings that encode pairs \((n,k)\) for which \(n\) has factor < \(k\)

• decision problem associated with \(L\):
  – Given \(x\), is \(x\) in \(L\)?
Problems and Languages

An aside: two encoding issues

1. implicitly assume we’ve agreed on a way to encode inputs (and outputs) as strings
   - sometimes relevant in fine-grained analysis (e.g. adj. matrix vs. adj. list for graphs)
   - almost never an issue in this class
   - avoid silly encodings: e.g. unary
2. some strings not valid encodings of any input -- treat as “no”
Problems and Languages

2. some strings not valid encodings of any input -- treat as “no”

What we usually mean by co-L

invalid

Valid

L

“yes”

“no”

Σ*

Σ*

“yes”

“no”
Complexity Classes

• **complexity class** = class of languages
• set-theoretic definition – no reference to computation (!)
• example:
  – **TALLY** = languages in which every yes instance has form $0^n$
  – e.g. $L = \{ 0^n : n \text{ prime} \}$
Complexity Classes

- complexity classes you know:
  - $P = \text{the set of languages decidable in polynomial time}$
  - $NP = \text{the set of languages } L \text{ where }$
    \[
    L = \{ x : \exists \ y, |y| \leq |x|^k, (x, y) \in R \}
    \]
    and R is a language in $P$

- easy to define complexity classes…
Complexity Classes

• ...harder to define *meaningful* complexity classes:
  – **capture** genuine computational phenomenon (e.g. parallelism)
  – **contain** natural and relevant problems
  – ideally **characterized** by natural problems (completeness – more soon)
  – **robust** under variations in model of computation
  – possibly **closed** under operations such as AND, OR, COMPLEMENT...
Complexity Classes

• need a **model of computation** to define classes that capture important aspects of computation

• Our model of computation: Turing Machine

```
  a   b   a   b
```

read/write head

finite control

infinite tape
Turing Machines

• $Q$ finite set of states
• $\Sigma$ alphabet including blank: “_”
• $q_{\text{start}}$, $q_{\text{accept}}$, $q_{\text{reject}}$ in $Q$
• $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, -\}$ transition fn.
• input written on tape, head on 1st square, state $q_{\text{start}}$
• sequence of steps specified by $\delta$
• if reach $q_{\text{accept}}$ or $q_{\text{reject}}$ then halt
Turing Machines

• three notions of computation with Turing machines. In all, input x written on tape…

  – **function computation**: output f(x) is left on the tape when TM halts

  – **language decision**: TM halts in state q_{accept} if x ∈ L; TM halts in state q_{reject} if x ∉ L.

  – **language recognition**: TM halts in state q_{accept} if x ∈ L; may loop forever otherwise.
Example:

<table>
<thead>
<tr>
<th>q</th>
<th>σ</th>
<th>δ(q,σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>(start, 0, R)</td>
</tr>
<tr>
<td>start</td>
<td>1</td>
<td>(start, 1, R)</td>
</tr>
<tr>
<td>start</td>
<td>_</td>
<td>(t, _, L)</td>
</tr>
<tr>
<td>start</td>
<td>#</td>
<td>(start, #, R)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>q</th>
<th>σ</th>
<th>δ(q,σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>(accept, 1, -)</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
<td>(t, 0, L)</td>
</tr>
<tr>
<td>t</td>
<td>#</td>
<td>(accept, #, R)</td>
</tr>
</tbody>
</table>

```
# 0 1
# 0 1
# 0 1
# 0 1
# 0 1
# 0 0
# 1 0
```

start
start
start
start
 t
 t
 accept
Turing Machines

- **multi-tape** Turing Machine:

  \[
  \delta: Q \times \sum^k \rightarrow Q \times \sum^k \times \{L,R,-\}^k
  \]

  \[
  \begin{array}{cccccc}
  a & b & a & b & \ldots & \\
  a & a & & & \ldots & \\
  b & b & c & d & \ldots & \\
  \end{array}
  \]

  (input tape)

  \[
  \begin{array}{cccccc}
  \text{finite} & \text{control} & \\
  \text{k tapes} & \\
  \text{Usually:} & \\
  \cdot \text{read-only “input tape”} & \\
  \cdot \text{write-only “output tape”} & \\
  \cdot \text{k-2 read/write “work tapes”} & \\
  \end{array}
  \]
Multitape TMs

simulation of k-tape TM by single-tape TM:

- add new symbol $x$ for each old $x$
- marks location of “virtual heads”

(input tape)
Multitape TMs

Repeat: \( O(t(n)) \) times

- scan tape, remembering the symbols under each virtual head in the state

\( O(k \ t(n)) = O(t(n)) \)

- make changes to reflect 1 step of \( M \);
  if hit \# , shift to right to make room.

\( O(k \ t(n)) = O(t(n)) \)

when \( M \) halts, erase all but output string

\( O(k \ t(n)) = O(t(n)) \)
Extended Church-Turing Thesis

• the belief that TMs formalize our intuitive notion of an efficient algorithm is:

The “extended” Church-Turing Thesis

everything we can compute in time $t(n)$ on a physical computer can be computed on a Turing Machine in time $t^{O(1)}(n)$ (polynomial slowdown)

• quantum computers challenge this belief
Extended Church-Turing Thesis

• consequence of extended Church-Turing Thesis: all reasonable physically realizable models of computation can be *efficiently* simulated by a TM

• e.g. multi-tape vs. single tape TM
• e.g. RAM model
Turing Machines

• Amazing fact: there exist (natural) **undecidable** problems

\[
\text{HALT} = \{ (M, x) : M \text{ halts on input } x \}
\]

• Theorem: HALT is undecidable.
Turing Machines

• Proof:
  – Suppose TM H decides HALT
  – Define new TM H’: on input <M>
    • if H accepts (M, <M>) then loop
    • if H rejects (M, <M>) then halt
  – Consider H’ on input <H’>:
    • if it halts, then H rejects (H’, <H’>), which implies it cannot halt
    • if it loops, then H accepts (H’, <H’>) which implies it must halt
  – contradiction.
Diagonalization

The existence of $H$ which tells us yes/no for each box allows us to construct a TM $H'$ that cannot be in the table.
Turing Machines

• Back to complexity classes:
  – $\text{TIME}(f(n))$ = languages decidable by a multi-tape TM in at most $f(n)$ steps, where $n$ is the input length, and $f : \mathbb{N} \rightarrow \mathbb{N}$
  – $\text{SPACE}(f(n))$ = languages decidable by a multi-tape TM that touches at most $f(n)$ squares of its work tapes, where $n$ is the input length, and $f : \mathbb{N} \rightarrow \mathbb{N}$

Note: $\mathbb{P} = \bigcup_{k \geq 1} \text{TIME}(n^k)$
Interlude

• In an ideal world, given language \( L \)
  – state an algorithm deciding \( L \)
  – \textbf{prove} that no algorithm does better

• we are pretty good at part 1

• we are currently \textbf{completely helpless}
  when it comes to part 2, for most problems
  that we care about
Interlude

• in place of part 2 we can
  – relate the difficulty of problems to each other via reductions
  – prove that a problem is a “hardest” problem in a complexity class via completeness

• powerful, successful surrogate for lower bounds
Reductions

• **reductions** are the main tool for relating problems to each other

• given two languages $L_1$ and $L_2$ we say “$L_1$ reduces to $L_2$” and we write “$L_1 \leq L_2$” to mean:
  
  – there exists an efficient (for now, poly-time) algorithm that computes a function $f$ s.t.
    
    • $x \in L_1$ implies $f(x) \in L_2$
    • $x \notin L_1$ implies $f(x) \notin L_2$
Reductions

• positive use: given new problem $L_1$ reduce it to $L_2$ that we know to be in $P$. Conclude $L_1$ in $P$ (how?)
  – e.g. bipartite matching $\leq$ max flow
  – formalizes “$L_1$ as easy as $L_2$”
Reductions

- **negative use**: given new problem $L_2$, reduce $L_1$ (that we believe not to be in $\mathbb{P}$) to it. Conclude $L_2$ *not in* $\mathbb{P}$ if $L_1$ *not in* $\mathbb{P}$ (how?)
  - e.g. satisfiability $\leq$ graph 3-coloring
  - formalizes “$L_2$ as hard as $L_1$”
Reductions

• Example reduction:
  – 3SAT = \{ \varphi : \varphi \text{ is a 3-CNF Boolean formula that has a satisfying assignment} \}  
    (3-CNF = AND of OR of \leq 3 literals)
  – IS = \{ (G, k) | G \text{ is a graph with an independent set } V' \subseteq V \text{ of size } \geq k \}  
    (ind. set = set of vertices no two of which are connected by an edge)