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?

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?
EXP ⊆ P/poly?
- Can every efficient randomized algorithm be converted into a deterministic algorithm one?
P = 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 polynomial time
  - NP = the set of problems with witnesses that can be checked in polynomial time
- … and notion of NP-completeness
- Useful tool
- Deep mathematical problem: P = NP?
  Course should be both useful and mathematically interesting
A question

• Given: polynomial \( f(x_1, x_2, \ldots, x_n) \) as arithmetic formula (fan-out 1):
  - multiplication (fan-in 2)
  - addition (fan-in 2)
  - negation (fan-in 1)

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

CS151 Lecture 1

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 poly-time algorithm for this problem.

CS151 Lecture 1

A randomized algorithm

• Given: multivariate degree \( r \) poly. \( f(x_1, x_2, \ldots, x_n) \)
  - note: \( r = \deg(f) \leq \) 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

CS151 Lecture 1

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 \( \log n \)

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

CS151 Lecture 1

Derandomization

This technique works on any randomized algorithm.

Gives generic “derandomization” of randomized procedures.

CS151 Lecture 1
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? \( EXP \subseteq P/poly? \) probably FALSE
- Can every randomized algorithm be converted into a deterministic algorithm one? \( P = BPP? \) 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

- 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?
  - can solve function problem efficiently using related decision problem (how?)
  - We will work mostly with decision problems
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"

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

• easy to define complexity classes…
Complexity Classes

- need a **model of computation** to define classes that capture important aspects of computation
- Our model of computation: **Turing Machine**

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

Example

<table>
<thead>
<tr>
<th>( q )</th>
<th>( a )</th>
<th>Action</th>
<th>( q )</th>
<th>( a )</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>#</td>
<td>start, #, R</td>
<td>#</td>
<td>#</td>
<td>accept, #, R</td>
</tr>
<tr>
<td>start</td>
<td>0</td>
<td>start, 0, R</td>
<td>#</td>
<td>#</td>
<td>accept, #, R</td>
</tr>
<tr>
<td>start</td>
<td>1</td>
<td>(start, 1, R)</td>
<td>#</td>
<td>#</td>
<td>accept, #, R</td>
</tr>
<tr>
<td>start</td>
<td>#</td>
<td>(start, #, L)</td>
<td>#</td>
<td>#</td>
<td>accept, #, R</td>
</tr>
<tr>
<td>start</td>
<td>#</td>
<td>(start, #, R)</td>
<td>#</td>
<td>#</td>
<td>accept, #, R</td>
</tr>
</tbody>
</table>

Multitape TMs

- **multi-tape** Turing Machine:
- \( \delta : Q \times \Sigma^k \rightarrow Q \times \Sigma^k \times \{(L, R, -) \}^k \) transition fn.
- \( k \) read/write "work tapes"
- Usually:
  - read-only "input tape"
  - write-only "output tape"
- \( k \geq 2 \)

Simulation of \( k \)-tape TM by single-tape TM:

- add new symbol \( x \) for each old \( x \)
- marks location of "virtual heads"
Multitape TMs
- Repeat: $O(t(n))$ times
  - scan tape, remembering the symbols under each virtual head in the state
    $O(k \cdot t(n)) = O(t(n))$
  - make changes to reflect 1 step of M; if hit #, shift to right to make room.
    $O(k \cdot t(n)) = O(t(n))$
  - when M halts, erase all but output string
    $O(k \cdot t(n)) = O(t(n))$

Repeat $O(t(n))$ times

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 $O(t(n))$ (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: $\text{HALT}$ is undecidable.

Proof:
- Suppose TM $H$ decides $\text{HALT}$
- 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.