Lecture 12: Overview

- NP-Completeness (Chapter 34)
NP Completeness

- Almost all algorithms we have covered so far have been polynomial-time algorithms.
- Worse-case running time is $O(n^k)$ for some constant $k$.
- Are all problems solvable in polynomial time?
Halting Problem

- Halting problem:
  - Can we write an algorithm that takes in a program and always determine whether the program will halt (i.e. complete its execution) or not?
  - The answer is no
- Why not?
Halting Problem

• Suppose that we had such an algorithm A
• Consider the program P:
  • Run A on itself (P)
  • If A says we halt, loop
  • If A say we loop, halt
• Halting problem can’t be solved
• Undecidable class
Categorizing Problems

- Like to categorize the difficulty of problems
- Difficulty is that different machine models may have change the complexity of the program
- Consider a machine that stores everything on a tape...memory accesses aren’t $O(1)$ anymore
- Turns out that all Turing machines are equivalent within a polynomial factor
Class P

- Create the first class of problems P (Polynomial)
- These problems can be solved in polynomial time
Class NP

- Non-deterministic polynomial
- Class of problems that we can verify the solution of in polynomial time
- Example: SAT Solver programming assignment
- If we give an assignment to all of the boolean variables that makes expression true, we can verify that it is satisfiable in polynomial time
- Easy solution in exponential time (iterate through possible solutions, see if the solution is correct)
Big Open Problem in Computer Science

- Does P=NP?
- If so, then a bunch of hard problems (factoring, etc) are actually easy.
  - Seems unlikely
  - Lots of people have been looking for good solutions to these problems
- But so far there is no proof that P≠NP
Decision vs. Optimization

- Talking about decision problems (is there a solution)
- Not talking about optimization problems (what is the best solution)
- Can convert optimization problems into decision problems (shortest path => is there a path of less than length l for some l)
NP-complete

- NP-complete problems are at least as hard as any other problem in NP
- How to prove:
  - Need to show that we can reduce any other problem in NP to this problem in polynomial time
  - or
  - Show that we can reduce another NP-complete problem to this problem in polynomial time
First NP-Complete Problem

- Circuit Satisfiability
- Given a logic circuit of AND, OR, and NOT gates, is there an input that causes the output to be true
- Same problem as the first homework assignment
Formalization

• Need to encode problem instance into a binary string using some type of encoding
• Algorithm that “solves” a problem actually takes an encoding of a problem instance as input.
• We state that an algorithm solves a concrete problem in $O(T(n))$ if when it is provided an instance $i$ of length $n = |i|$, the algorithm can produce a solution in $O(T(n))$ time.
• An algorithm is polynomial time solvable if there exists an algorithm to solve in $O(n^k)$ time
• Suppose we have an algorithm that operates on an integer k specified as k 1’s and is $\Theta(k)$
• With a normal encoding, $n=\log|k|$ and the algorithm is $O(2^n)$
• Normally rule out really bad encodings of the problem
Polynomial-Time Computable

• Say that a function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is polynomial computable is there exists a polynomial time algorithm that given an input $x$ in $\{0,1\}^*$ produces $f(x)$.

• Two encodings $e_1$ and $e_2$ are polynomial related if there exists two polynomial-time computable functions $f_{12}$ and $f_{21}$ such that $f_{12}(e_1(i))=e_2(i)$ and $f_{21}(e_2(i))=e_1(i)$
Formal-Language Framework

• An alphabet is a finite set of symbols
• A language \( L \) over an alphabet is any set of strings made up of symbols from the alphabet
• Example
  • If Alphabet = \{0,1\}, then the set \( L = \{10, 11, 101, 111, 1011, \ldots\} \) is the language of binary representations of prime numbers
• The empty string is denoted by \( \epsilon \), and the empty language by \( \emptyset \)
Operations on Language

- Set theoretic operations such as union and intersection
- Concatenation of two languages:
  \[ L = \{ x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2 \} \]
- Closure or Kleene star of a language \( L \) is
  \[ L^* = \{ \varepsilon \} \cup L \cup L^2 \cup L^3 \cup \ldots \]
Language Theory

- Set of instances for any decision problem Q is simply the set \( \{0, 1\}^* \).
- Q can be completely characterized by the set of instances for the problem that produce a 1 (yes) answer.
- Can view Q as a language L of \{0,1\} where \( L = \{x \in \{0, 1\}^* : Q(x) = 1\} \).
Language Theory

• Language L is decided by A if it accepts every string in the language and none of the strings not in the language.

• Language L is accepted in polynomial time by algorithm A if it is accepted by A and if in addition there is a constant k such that for any length n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

• Language L is decided in polynomial time by algorithm A if there is a constant k such that for any length n string $x \in \{0,1\}^*$, algorithm A correctly decides whether $x \in L$ in time $O(n^k)$. 
Language Theory

\[ P = \{ L \subseteq \{0, 1\}^*: \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time} \} \]