SIMS 255 Foundations of Software Design

Complexity and NP-completeness

Matt Welsh

November 29, 2001

mdw@cs.berkeley.edu
Outline

Complexity of algorithms

- Space and time complexity
- "Big O" notation
- Complexity hierarchies and algorithm examples

\( \mathcal{P} \) and \( \mathcal{NP} \)

- Decision problems
- Polynomial time decidability and computability
- The ultimate question: Does \( \mathcal{P} = \mathcal{NP} \)?
- \( \mathcal{NP} \)-completeness
- Examples of \( \mathcal{NP} \)-complete problems
Complexity of Algorithms

The kinds of questions we want to answer:

- Given an algorithm, how much time does it take to run?
- Given an algorithm, how much space does it use?

  ▶ Characterized by the size of the problem

A simple example: finding max in a sequence of numbers

- Algorithm: Scan the numbers from 1 to \( N \), find the maximum value
- The run time is “order \( N \)”

Another example: is a given number prime?

- Recall - prime number cannot be divided by any integer
- The “size of the problem” is \( N = \log_{10} x \) (number of digits in \( x \))
- Brute force algorithm: divide \( x \) by every number less than \( \sqrt{x} \)
- Run time: about \( 10^{N/2} \)
Big O Notation

This is a way of characterizing the run time (or space constraints) of a given algorithm.

We say a function \( f(n) \) is "\( O(g(n)) \)" when

\[
f(n) \leq Cg(n)
\]

for some constant of \( C \).

For example,

\[
f(n) = 859398n^5 + 29810n^3 + 10191032n
\]

is \( O(n^5) \), since as \( n \to \infty \), \( f(n) \) "looks like" \( n^5 \) regardless of those big constants!
Big O Examples

Linear: $O(n)$
- e.g., Finding max or min in a sequence of numbers

Polynomial: $O(n^p)$ for some integer $p$
- Classic "bubblesort" algorithm is $O(n^2)$

Logarithmic: $O(\log n)$
- "Quicksort" algorithm is $O(n \log n)$
- To sort 1 million numbers, quicksort takes 6 million steps, bubblesort takes a trillion!

Exponential: $O(B^n)$ for some constant $B > 1$
- Brute force factorization, lots of numerical problems

Factorial: $O(n!)$
- $n!$ defined as $n \times (n-1) \times (n-2) \times ... \times 1$
- e.g., Calculating the Fibonacci numbers recursively
  $(0, 1, 1, 2, 3, 5, 8, 13, ...)$
Comparison of complexity classes

- Linear
- Polynomial
- Exponential

Running time vs. Problem size graph.
Looking at the last slide, it seems that exponential run times are pretty bad!

- In fact, they are worse than almost anything else
- $O(n!)$ and $O(n^n)$ are even worse, but uncommon

We say that **tractable** problems are those that we can solve in practice

**Intractible** problems can be solved in theory, but not in practice

- Tractable problems have solutions that are *polynomial* or better
- Intractable problems have solutions that are *exponential* or worse

Some problems are flat-out **unsolvable**!

- This is not to say that they are “really hard”, but rather no computer could possibly solve them
- Next lecture!
A complexity class is a set of computational problems with the same bounds in time and space.

Say I give you a problem to solve -- what is your best hope for an efficient algorithm?

- If you can reduce the problem to another problem with known complexity, then you can answer straight away!

Example: Given a graph $G$, is it possible to color the nodes with just 3 colors, such that no two adjacent nodes have the same color?
Problem Reduction

It turns out we can reduce this problem to another one: boolean satisfiability

- Given a boolean expression of multiple variables, is there some assignment to the variables that makes the expression true?

\[(p \lor q \lor \bar{r} \lor s) \land (\bar{q} \lor s)\]

- What values should you assign to \(p, q, r,\) and \(s\) to make this statement true?

It turns out there is no known polynomial time solution to this problem!
A decision problem is one that seeks a yes-or-no answer.

Example: \( k \)-colorability
- Can this graph be colored with \( k \) colors?

Traveling Salesperson Problem
- Given a set of cities with distances between them, can someone travel to each city (without visiting a city more than once) in \( M \) miles or less?

Traveling salesperson problem A classical scheduling problem that has baffled linear programmers for 30 years, but which, in a more complex formulation, is solved daily by traveling salespersons.
**Optimization Problems**

Some problems are **optimization problems**

- What is the **fewest number** of colors that color this graph?
- What is the **shortest path** that one can take to visit all the cities?

Usually if we have a way to solve the decision problem, without much more work we can solve the corresponding optimization problem:

- Start with a graph $G$
- Find out if $G$ can be colored with 50 colors $\rightarrow$ yes or no
- If yes, then try 25 colors
- If yes, then try 12 colors, etc.

So, we mainly talk about decision problems

- Can easily derive the corresponding optimization problem
Polynomial-time decidability

We say a problem is **polynomial-time decidable** if:

- Given a problem $P$ and a proposed solution $s$
- There is a polynomial time algorithm that checks whether $s$ is a solution for $P$

**Example: Graph colorability**

- Given a graph $G$ and an assignment of colors to nodes
- Can easily check whether any two nodes have the same color
- Simple algorithm is $O(n)$, where $n$ is the number of nodes
The Complexity Class $\mathcal{NP}$

The set of problems for which the answer can be checked in polynomial time is called $\mathcal{NP}$

- $\mathcal{NP}$ stands for "nondeterministic polynomial time"

The name comes from a "nondeterministic Turing machine"

- Formal model of computing that allows the (theoretical) machine to perform an infinite number of operations simultaneously
- More about this next lecture!

For now, think of problems in $\mathcal{NP}$ as those that we have some way of quickly checking answers for

- But not necessarily a fast way to get the answer!
A problem is \textit{polynomial-time computable} if:

\begin{itemize}
  \item We can \textbf{find a solution} in polynomial time
  \item Note that this is quite different than \textit{checking a given solution}!
\end{itemize}

The set of problems that have this property is called $\mathcal{P}$

Generally speaking, problems in $\mathcal{P}$ are “good”

\begin{itemize}
  \item i.e., We have fast algorithms for them
  \item Even if a problem is $O(n^{1902892})$, it’s still better than $O(10^n)$!
  \item In practice, most problems in $\mathcal{P}$ are $O(n^k)$ for small $k$
\end{itemize}
**Does \( P = NP \)?**

All problems in \( P \) are also in \( NP \)
- But the converse is not known to be true

The most important open problem in Computer Science!
- In fact there is a $1,000,000 award for anyone who can solve it

We know that many problems don’t seem to have polynomial-time algorithms
- But, nobody has proven that these “hard” problems are not in \( P \)
- There may be some mysterious poly-time algorithm for one of those “hard” problems lurking out there...
Example: Factoring Large Numbers

Many modern systems rely on public key cryptography
- Popular implementation is RSA
- Used in all Web browsers for secure connections

Start with two (large) prime numbers, and multiply them
- Primes $P$ and $Q$, with product $PQ$
- Note that $P$ and $Q$ are the only two numbers that you can multiply to get $PQ$

We can make the product $PQ$ public
- Because it is very hard to factor the number into the “secrets” $P$ and $Q$!

Public key encryption idea:
- Bob publishes the number $PQ$ to the world
- Any one can use $PQ$ to encrypt a message for Bob
- Only Bob knows $P$ and $Q$ separately to decrypt the message
Factoring Large Numbers is Hard!

Factoring is in $\mathcal{NP}$

- It’s easy to check whether two factors $P$ and $Q$ multiply to get $PQ$
- But, the fastest algorithm we have for finding factors is still exponential:

$$O(e^{c \log n^{1/3} \log \log n^{2/3}})$$

Still, better factoring algorithms are always being developed...

- In 1977, Ron Rivest said that factoring a 125-digit number would take 40 quadrillion years
- In 1994, a 129-digit number was factored

Upshot: If $\mathcal{P} = \mathcal{NP}$, then all hard problems (or at least those in $\mathcal{NP}$) can be solved in polynomial time!

- See the movie “Sneakers”
NP-Completeness

The “hardest” problems in $\mathcal{NP}$ are called $\mathcal{NP}$-complete.

A problem is $\mathcal{NP}$-complete if:

- It is in $\mathcal{NP}$
- All other problems in $\mathcal{NP}$ can be reduced to it (in polynomial time)

Result: If we can find a mapping from any $\mathcal{NP}$-complete problem to any problem in $\mathcal{P}$, then all problems in $\mathcal{NP}$ are also in $\mathcal{P}$!
Examples of $\mathcal{NP}$-Complete Problems

Boolean satisfiability

- Given a boolean expression in a set of variables, what values of the variables makes the expression true?

Traveling Salesperson Problem

- Given a set of cities connected by roads, what is the path of minimum distance that visits all cities exactly once?

$k$-colorability

- Given a graph $G$, what assignment of $k$ colors to the nodes leaves no two adjacent nodes with the same color?

Partition problem

- Given a list of integers $x_1, x_2, \ldots$, does there exist a subset whose sum is exactly $\frac{1}{2} \sum x_i$?
The Deeper Meaning

$NP$-completeness is about the theoretical limits of computing

- If a problem is $NP$-complete, it is very unlikely that we will ever find a fast algorithm for it

Nobody knows whether $P = NP$

- Although many people have been working on it for years
- It’s impressive that we can’t even prove $P \neq NP$

This is not about Computer Scientists “not realizing” that there is a fast algorithm for an $NP$-complete problem

- Rather, this is a fundamental limit on what can and cannot be computed efficiently!
- Huge implications: If you know a problem is $NP$-complete, you might as well give up looking for a fast solution
Some hope for the future

Random algorithms and approximations

- Many $NP$-complete problems can be approximated by fast techniques
- For example, Monte Carlo methods use randomness to "guess" an answer to a problem
- Can often trust the answer with 99.99999% (or more) confidence

Quantum Computing

- Computers built using quantum particles can quickly compute many answers simultaneously
- It turns out that quantum computers can (theoretically) solve many problems efficiently, for which no previous fast algorithm was known
- For example, a Quantum Computer can factor numbers in polynomial time!
- But, QCs are very hard to build
Summary

Algorithm complexity and “Big O” notation
Comparing complexities: linear, polynomial, exponential
Tractable and intractable problems
Complexity classes and decision problems
Polynomial-time decidability (NP)
Polynomial-time computability (P)
The $P = NP$ problem and NP-completeness