Algorithms and Computational Complexity: an Overview

Autumn 2011
Larry Ruzzo

Thanks to Paul Beame, James Lee, Kevin Wayne for some slides
Design of Algorithms – a taste

design methods

common or important types of problems

analysis of algorithms - efficiency
Complexity & intractability – a taste

- solving problems in principle is not enough
  - algorithms must be efficient

- some problems have no efficient solution

NP-complete problems
  - important & useful class of problems whose solutions (seemingly) cannot be found efficiently
Cryptography (e.g. RSA, SSL in browsers)

Secret: $p,q$ prime, say 512 bits each

Public: $n$ which equals $p \times q$, 1024 bits

In principle

*there is an algorithm* that given $n$ will find $p$ and $q$:

try all $2^{512} \approx 1.3 \times 10^{154}$ possible $p$’s (but that’s kinda big…

for comparison, the age of the universe is $\approx 5 \times 10^{29}$ picosec)

In practice

*no fast algorithm* known for this problem (on non-quantum computers)

security of RSA depends on this fact

(and research in “quantum computing” is strongly driven by the possibility of changing this)
Moore’s Law and the exponential improvements in hardware...

Ex: sparse linear equations over 25 years

10 orders of magnitude improvement!
25 years progress solving sparse linear systems

Hardware alone: 4 orders of magnitude

Source: Sandia, via M. Schultz
25 years progress solving sparse linear systems

Hardware alone: 4 orders of magnitude

Software alone: 6 orders of magnitude

Source: Sandia, via M. Schultz
The N-Body Problem:
in 30 years
$10^7$ hardware
$10^{10}$ software
Procedure to accomplish a task or solve a well-specified problem

Well-specified: know what all possible inputs look like and what output looks like given them

“accomplish” via simple, well-defined steps

Ex: sorting names (via comparison)

Ex: checking for primality (via +, -, *, /, ≤)
Printed circuit-board company has a robot arm that solders components to the board.

Time: proportional to total distance the arm must move from initial rest position around the board and back to the initial position.

For each board design, find best order to do the soldering.
printed circuit board
printed circuit board
Input: Given a set $S$ of $n$ points in the plane
Output: The shortest cycle tour that visits each point in the set $S$.

Better known as “TSP”

How might you solve it?
nearest neighbor heuristic

Start at some point $p_0$
Walk first to its nearest neighbor $p_1$
Walk to the nearest unvisited neighbor $p_2$, then nearest unvisited $p_3$, ... until all points have been visited
Then walk back to $p_0$

heuristic:
A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. May be good, but usually not guaranteed to give the best or fastest solution. (And often difficult to analyze precisely.)
nearest neighbor heuristic
an input where nn works badly

length \sim 84
an input where nn works badly

optimal soln for this example
length ~ 64
Repeatedly join the closest pair of points
(such that result can still be part of a single loop in the end. I.e., join endpoints, but not points in middle, of path segments already created.)

How does this work on our bad example?
a bad example for closest pair
a bad example for closest pair

\[ 6 + \sqrt{10} = 9.16 \]

vs

8
“Brute Force Search”:
For each of the $n! = n(n-1)(n-2)\ldots 1$ orderings of the points, check the length of the cycle;
Keep the best one
The two incorrect algorithms were greedy

- Often very natural & tempting ideas
- They make choices that look great “locally” (and never reconsider them)
- When greed works, the algorithms are typically efficient
- BUT: often does not work - you get boxed in

Our correct alg avoids this, but is incredibly slow

- 20! is so large that checking one billion per second would take 2.4 billion seconds (around 70 years!)
- And growing: \( n! \sim \sqrt{2 \pi n} \cdot (n/e)^n \sim 2^{O(n \log n)} \)
the morals of the story

Algorithms are important
   Many performance gains outstrip Moore’s law
Simple problems can be hard
   Factoring, TSP, *many* others
Simple ideas don’t always work
   Nearest neighbor, closest pair heuristics
Simple algorithms can be very slow
   Brute-force factoring, TSP
A point we hope to make: for some problems, even the *best* algorithms are slow
A brief overview of the theory of algorithms
Efficiency & asymptotic analysis
Some scattered examples of simple problems where clever algorithms help
A brief overview of the theory of intractability
Especially NP-complete problems

“Basics every educated CSE student should know”
The *complexity* of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes, with each problem size $n$.

Mathematically,

$$ T: \mathbb{N}^+ \to \mathbb{R}^+ $$

i.e., $T$ is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).
computational complexity: general goals

Asymptotic growth rate, i.e., characterize growth rate of worst-case run time as a function of problem size, up to a constant factor, e.g. \( T(n) = O(n^2) \)

Why not try to be more precise?

Average-case, e.g., is hard to define, analyze
Technological variations (computer, compiler, OS, …) easily 10x or more
Being more precise is a ton of work
A key question is “scale up”: if I can afford this today, how much longer will it take when my business is 2x larger? (E.g. today: \( cn^2 \), next year: \( c(2n)^2 = 4cn^2 \) : 4 x longer.) Big-O analysis is adequate to address this.
computational complexity

- $T(n)$
- $2n \log_2 n$
- $n \log_2 n$
For all $r > 1$ (no matter how small) and all $d > 0$, (no matter how large)
$n^d = O(r^n)$. 

In short, every exponential grows faster than every polynomial!
the complexity class P: polynomial time

P: Running time $O(n^d)$ for some constant $d$
(d is independent of the input size $n$)

**Nice scaling property:** there is a constant $c$ s.t. doubling $n$, time increases only by a factor of $c$.
(E.g., $c \sim 2^d$)

Contrast with exponential: For any constant $c$, there is a $d$ such that $n \rightarrow n+d$ increases time by a factor of more than $c$.
(E.g., $c = 100$ and $d = 7$ for $2^n$ vs $2^{n+7}$)
polynomial vs exponential growth
why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>n</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

not only get very big, but do so abruptly, which likely yields erratic performance on small instances
Next year's computer will be 2x faster. If I can solve problem of size $n_0$ today, how large a problem can I solve in the same time next year?

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Increase</th>
<th>E.g. $T=10^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(n)</td>
<td>$n_0 \rightarrow 2n_0$</td>
<td>$10^{12} \rightarrow 2 \times 10^{12}$</td>
</tr>
<tr>
<td>O($n^2$)</td>
<td>$n_0 \rightarrow \sqrt{2} n_0$</td>
<td>$10^6 \rightarrow 1.4 \times 10^6$</td>
</tr>
<tr>
<td>O($n^3$)</td>
<td>$n_0 \rightarrow 3\sqrt{2} n_0$</td>
<td>$10^4 \rightarrow 1.25 \times 10^4$</td>
</tr>
<tr>
<td>$2^n/10$</td>
<td>$n_0 \rightarrow n_0+10$</td>
<td>$400 \rightarrow 410$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$n_0 \rightarrow n_0+1$</td>
<td>$40 \rightarrow 41$</td>
</tr>
</tbody>
</table>
Typical initial goal for algorithm analysis is to find an

asymptotic
upper bound on
worst case running time
as a function of problem size

This is rarely the last word, but often helps separate good algorithms from blatantly poor ones - concentrate on the good ones!
why “polynomial”? 

Point is not that $n^{2000}$ is a nice time bound, or that the differences among $n$ and $2n$ and $n^2$ are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

“My problem is in P” is a starting point for a more detailed analysis

“My problem is not in P” may suggest that you need to shift to a more tractable variant, or otherwise readjust expectations