Algorithms and Computational Complexity: an Overview

Winter 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 x q, 1024 bits

In principle

there is an algorithm that given n will find p and q:
try all \(2^{512}\) possible p’s, but an astronomical number

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

algorithms or hardware?
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
algorithms: a definition

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
more precise problem definition

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.
nearest neighbor heuristic
an input where nn works badly

length ~ 84
an input where nn works badly

optimal soln for this example
length $\sim 64$
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: \( n! \sim \sqrt{2\pi n} \cdot (n/e)^n \sim 2^{O(n \log n)} \)
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
my plan

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,

$T: N^+ \rightarrow R^+$

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

Problem size

Time

T(n)

Problem size

Time
computational complexity: general goals

Characterize growth rate of worst-case run time as a function of problem size, up to a constant factor

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 \times\) longer.)
Problem size

Time

computational complexity

T(n)

2n log₂n

n log₂n
Given two functions $f$ and $g$: $\mathbb{N} \rightarrow \mathbb{R}$, $f(n)$ is $O(g(n))$ iff

$\exists$ constant $c > 0$ so that $f(n)$ is eventually always $\leq c \cdot g(n)$

Example:

$10n^2 - 16n + 100$ is $O(n^2)$ (and also $O(n^3)$…)

why?:

$10n^2 - 16n + 100 \leq 11n^2$ for all $n \geq 10$
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., $2^n$ vs $2^{n+1}$)
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>$n = 10$</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>$n = 30$</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>$n = 50$</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>$n = 100$</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>$n = 1,000$</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>$n = 10,000$</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>$n = 100,000$</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>$n = 1,000,000$</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 \backslash/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
algorithm design techniques
We will survey two:

Later: Dynamic programming
   Orderly solution of many smaller sub-problems, typically non-disjoint
   Can give exponential speedups compared to more brute-force approaches

Today: Divide & Conquer
   Reduce problem to one or more sub-problems of the same type, typically disjoint
   Often gives significant, usually polynomial, speedup
algorithm design techniques

Divide & Conquer

Reduce problem to one or more sub-problems of the same type
Each sub-problem’s size a fraction of the original
Sub-problem’s typically disjoint
Often gives significant, usually polynomial, speedup
Examples:
    Mergesort, Binary Search, Strassen’s Algorithm, Quicksort (roughly)
Suppose we've already invented DumbSort, taking time \( n^2 \)

Try **Just One Level** of divide & conquer:

- DumbSort(first \( n/2 \) elements)
- DumbSort(last \( n/2 \) elements)

Merge results

**Time:** \( 2 \left(\frac{n}{2}\right)^2 + n = \frac{n^2}{2} + n \ll n^2 \)

*Almost twice as fast!*
Moral 1: “two halves are better than a whole”

Two problems of half size are better than one full-size problem, even given $O(n)$ overhead of recombining, since the base algorithm has super-linear complexity.

Moral 2: “If a little's good, then more's better”

Two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing. In the limit: you’ve just rediscovered mergesort.
Mergesort: (recursively) sort 2 half-lists, then merge results.

\[ T(n) = 2T(n/2) + cn, \quad n \geq 2 \]

\[ T(1) = 0 \]

Solution: \( O(n \log n) \)
A Divide & Conquer Example: Closest Pair of Points
closest pair of points: non-geometric version

Given $n$ points and arbitrary distances between them, find the closest pair. (E.g., think of distance as airfare – definitely not Euclidean distance!)

Must look at all $n$ choose 2 pairwise distances, else any one you didn’t check might be the shortest.

Also true for Euclidean distance in 1-2 dimensions?
Given \( n \) points on the real line, find the closest pair

Closest pair is \textit{adjacent} in ordered list

Time \( O(n \log n) \) to sort, if needed

Plus \( O(n) \) to scan adjacent pairs

Key point: do not need to calc distances between all pairs: exploit geometry + ordering
closest pair of points. 2d, Euclidean distance: 1st try

Divide. Sub-divide region into 4 quadrants.
Divide. Sub-divide region into 4 quadrants. Obstacle. Impossible to ensure \( \frac{n}{4} \) points in each piece.
Algorithm.

Divide: draw vertical line $L$ with $\approx n/2$ points on each side.
Algorithm.

Divide: draw vertical line $L$ with $\approx n/2$ points on each side.
Conquer: find closest pair on each side, recursively.
Algorithm.

Divide: draw vertical line $L$ with $\approx n/2$ points on each side.
Conquer: find closest pair on each side, recursively.
Combine: find closest pair with one point in each side.
Return best of 3 solutions.
Find closest pair with one point in each side, assuming that distance < $\delta$. 

$\delta = \min(12, 21)$
Find closest pair with one point in each side, assuming that distance < $\delta$.

Observation: suffices to consider points within $\delta$ of line $L$.
Find closest pair with one point in each side, assuming that distance < \( \delta \).

Observation: suffices to consider points within \( \delta \) of line \( L \).

Almost the one-D problem again: Sort points in \( 2\delta \)-strip by their \( y \) coordinate.

\[ \delta = \min(12, 21) \]
Find closest pair with one point in each side, assuming that distance $< \delta$.

Observation: suffices to consider points within $\delta$ of line $L$.

Almost the one-D problem again: Sort points in $2\delta$-strip by their $y$ coordinate. Only check pts within 8 in sorted list!

$\delta = \min(12, 21)$
Def. Let $s_i$ be the point in the $2\delta$-strip, with the $i^{th}$ smallest $y$-coordinate.

Claim. If $|i - j| > 8$, then the distance between $s_i$ and $s_j$ is $> \delta$.

Pf: No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box; only 8 boxes within $\delta$.
Number of pairwise distance calculations:

\[ D(n) \leq \begin{cases} 
0 & \quad n = 1 \\
2D(n/2) + 7n & \quad n > 1 
\end{cases} \quad \Rightarrow \quad D(n) = O(n \log n) \]

(A mostly superfluous detail: straightforward implementation gives a running time that is a factor of log n larger, due to sorting in the various subproblems. Run time can be reduced to \( O(n \log n) \) also, roughly by the trick of sorting by \( x \) at the top level, and returning/merging \( y \)-sorted lists from the subcalls.

Regardless of this nuance, the big picture is the same: divide-and-conquer allows sharp speed gain over a naive \( n^2 \) method.)
Integer Multiplication
Add. Given two n-digit integers a and b, compute $a + b$.

$O(n)$ bit operations.
Add. Given two n-digit integers a and b, compute \( a + b \). 
\( \mathcal{O}(n) \) bit operations.

Multiply. Given two n-digit integers a and b, compute \( a \times b \). 
The “grade school” method: \( \Theta(n^2) \) bit operations.
To multiply two 2-digit integers:

Multiply four 1-digit integers.

Add, shift some 2-digit integers to obtain result.

\[
x = 10 \cdot x_1 + x_0 \\
y = 10 \cdot y_1 + y_0 \\
xy = (10 \cdot x_1 + x_0) (10 \cdot y_1 + y_0) \\
     = 100 \cdot x_1y_1 + 10 \cdot (x_1y_0 + x_0y_1) + x_0y_0
\]

Same idea works for *long* integers –
can split them into 4 half-sized ints
divide-and-conquer multiplication: warmup

To multiply two \( n \)-digit integers:

Multiply four \( n/2 \)-digit integers.

Add, shift some \( n/2 \)-digit integers to obtain result.

\[
\begin{align*}
  x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right) \left(2^{n/2} \cdot y_1 + y_0\right) \\
&= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0
\end{align*}
\]

\[
T(n) = 4T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^2)
\]

assumes \( n \) is a power of 2
key trick: 2 multiplies for the price of 1

\[
x = 2^{n/2} \cdot x_1 + x_0 \\
y = 2^{n/2} \cdot y_1 + y_0 \\
xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) \\
= 2^n \cdot x_1 y_1 + 2^{n/2} (x_1 y_0 + x_0 y_1) + x_0 y_0
\]

Well, ok, 4 for 3 is more accurate…

\[
\alpha = x_1 + x_0 \\
\beta = y_1 + y_0 \\
\alpha \beta = (x_1 + x_0)(y_1 + y_0) \\
= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
(x_1 y_0 + x_0 y_1) = \alpha \beta - x_1 y_1 - x_0 y_0
\]
Karatsuba multiplication

To multiply two n-digit integers:

Add two $\frac{1}{2}n$ digit integers.

Multiply three $\frac{1}{2}n$-digit integers.

Add, subtract, and shift $\frac{1}{2}n$-digit integers to obtain result.

\[
x = 2^{n/2} \cdot x_1 + x_0 \\
y = 2^{n/2} \cdot y_1 + y_0 \\
xy = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
      = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) + x_0 y_0
\]

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit ops.

\[
T(n) \leq 3 T(n/2) + O(n) \quad \text{recursive calls, add, subtract, shift}
\]

\[
\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})
\]
Naïve: $\Theta(n^2)$

Karatsuba: $\Theta(n^{1.59...})$

Amusing exercise: generalize Karatsuba to do 5 size $n/3$ subproblems $\rightarrow \Theta(n^{1.46...})$

Best known: $\Theta(n \log n \log\log n)$

"Fast Fourier Transform"

but mostly unused in practice (unless you need really big numbers - a billion digits of $\pi$, say)

High precision arithmetic IS important for crypto
Idea:

“Two halves are better than a whole”
if the base algorithm has super-linear complexity.

“If a little's good, then more's better”
repeat above, recursively

Applications: Many.
Binary Search, Merge Sort, (Quicksort), Closest points, Integer multiply,…