CSE 421: Intro Algorithms

2: Analysis

Larry Ruzzo
Efficiency

Our correct TSP algorithm was incredibly slow
Basically slow no matter what computer you have
We want a general theory of “efficiency” that is

Simple
Objective
Relatively independent of changing technology
But still predictive – “theoretically bad” algorithms should be bad in practice and vice versa (usually)
Defining Efficiency

“Runs fast on typical real problem instances”

Pro:
  sensible, bottom-line-oriented

Con:
  moving target (diff computers, compilers, Moore’s law)
  highly subjective (how fast is “fast”? What’s “typical”? )
The time complexity of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes, with each problem size $n$.

Mathematically,

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

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

“Reals” so we can say, e.g., $\sqrt{n}$ instead of $\lfloor \sqrt{n} \rfloor$
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

![Graph showing computational complexity](image)

- Problem size
- Time

- $T(n)$
- $2n \log_2 n$
- $n \log_2 n$
Given two functions $f$ and $g: \mathbb{N} \to \mathbb{R}$

- $f(n)$ is $O(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c g(n)$

- $f(n)$ is $\Omega (g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c g(n)$

- $f(n)$ is $\Theta (g(n))$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(n) \leq f(n) \leq c_2 g(n)$
10n^2 - 16n + 100 is $O(n^2)$ also $O(n^3)$

$10n^2 - 16n + 100 \leq 11n^2$ for all $n \geq 10$

$10n^2 - 16n + 100$ is $\Omega(n^2)$ also $\Omega(n)$

$10n^2 - 16n + 100 \geq 9n^2$ for all $n \geq 16$

Therefore also $10n^2 - 16n + 100$ is $\Theta(n^2)$

$10n^2 - 16n + 100$ is not $O(n)$ also not $\Omega(n^3)$
Properties

Transitivity.

If $f = \mathcal{O}(g)$ and $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$.

If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.

If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivitiy.

If $f = \mathcal{O}(h)$ and $g = \mathcal{O}(h)$ then $f + g = \mathcal{O}(h)$.

If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.

If $f = \Theta(h)$ and $g = \mathcal{O}(h)$ then $f + g = \Theta(h)$. 
Working with $O$-$Ω$-$Θ$ notation

Claim: For any $a$, and any $b>0$, $(n+a)^b$ is $Θ(n^b)$

\[
(n+a)^b \leq (2n)^b \quad \text{for } n \geq |a|
= 2^b n^b
= cn^b \quad \text{for } c = 2^b
\]
so $(n+a)^b$ is $O(n^b)$

\[
(n+a)^b \geq (n/2)^b \quad \text{for } n \geq 2|a| \text{ (even if } a < 0)
= 2^{-b} n^b
= c'n \quad \text{for } c' = 2^{-b}
\]
so $(n+a)^b$ is $Ω (n^b)$
Working with $O$-$\Omega$-$\Theta$ notation

Claim: For any $a, b > 1$ \( \log_a n \) is $\Theta(\log_b n)$

\[
\log_a b = x \text{ means } a^x = b \\
\quad a^{\log_a b} = b \\
(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n \\
(\log_a b)(\log_b n) = \log_a n \\
c \log_b n = \log_a n \text{ for the constant } c = \log_a b \\
\text{So:} \\
\log_b n = \Theta(\log_a n) = \Theta(\log n)
Asymptotic Bounds for Some Common Functions

Polynomials:
\[ a_0 + a_1 n + \ldots + a_d n^d \text{ is } \Theta(n^d) \text{ if } a_d > 0 \]

Logarithms:
\[ O(\log_a n) = O(\log_b n) \text{ for any constants } a, b > 0 \]
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!
Logarithms:
For all $x > 0$, \((\text{no matter how small})\) \(\log n = O(n^x)\)

log grows slower than every polynomial
Domination

\( f(n) \) is \( o(g(n)) \) iff \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)
that is \( g(n) \) dominates \( f(n) \)

If \( a \leq b \) then \( n^a \) is \( \Theta(n^b) \)

If \( a < b \) then \( n^a \) is \( o(n^b) \)

Note:
if \( f(n) \) is \( \Theta(g(n)) \) then it cannot be \( o(g(n)) \)
Working with little-o

\[ n^2 = o(n^3) \text{ [Use algebra]:} \]

\[ \lim_{n \to \infty} \frac{n^2}{n^3} = \lim_{n \to \infty} \frac{1}{n} = 0 \]

\[ n^3 = o(e^n) \text{ [Use L’Hospital’s rule 3 times]:} \]

\[ \lim_{n \to \infty} \frac{n^3}{e^n} = \lim_{n \to \infty} \frac{3n^2}{e^n} = \lim_{n \to \infty} \frac{6n}{e^n} = \lim_{n \to \infty} \frac{6}{e^n} = 0 \]
For all \( r > 1 \) (no matter how small) and all \( d > 0 \), (no matter how large)

\[ n^d = O(r^n) \]

\[ n^d = o(r^n), \text{ even} \]

In short, every exponential grows faster than every polynomial!
Big-Theta, etc. not always “nice”

\[
f(n) = \begin{cases} 
  n^2, & n \text{ even} \\
  n, & n \text{ odd}
\end{cases}
\]

\[f(n) \neq \Theta(n^a)\] for any \(a\).

Fortunately, such nasty cases are rare

\[f(n \log n) \neq \Theta(n^a)\] for any \(a\), either, but at least it’s simpler.
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 \( n \to n+d \) 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>$n = 10$</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>$n_0 \to 2n_0$</td>
<td>$10^{12} \to 2 \times 10^{12}$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$n_0 \to \sqrt{2} \ n_0$</td>
<td>$10^6 \to 1.4 \times 10^6$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$n_0 \to 3\sqrt{2} \ n_0$</td>
<td>$10^4 \to 1.25 \times 10^4$</td>
</tr>
<tr>
<td>$2^{n/10}$</td>
<td>$n_0 \to n_0+10$</td>
<td>$400 \to 410$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$n_0 \to n_0+1$</td>
<td>$40 \to 41$</td>
</tr>
</tbody>
</table>
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
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!