Define 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”?)

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)

Measuring efficiency

Time = # of instructions executed in a simple programming language
- only simple operations (+, *, -, =, if, call, ...)
- each operation takes one time step
- each memory access takes one time step
- no fancy stuff (add these two matrices, copy this long string, ...) built in; write it/charge for it as above
- No fixed bound on the memory size
We left out things but...

Things we’ve dropped
memory hierarchy
disk, caches, registers have many orders of magnitude differences in access time
not all instructions take the same time in practice
different computers have different primitive instructions

However,
the RAM model is useful for designing algorithms and measuring their efficiency
one can usually tune implementations so that the hierarchy etc. is not a huge factor

Pros and cons:

Best-case
unrealistic oversell

Average-case
over what probability distribution? (different people may have different “average” problems)
analysis often hard

Worst-case
a fast algorithm has a comforting guarantee
maybe too pessimistic

Complexity analysis

Problem size n
Worst-case complexity: max # steps algorithm takes on any input of size n
Best-case complexity: min # steps algorithm takes on any input of size n
Average-case complexity: avg # steps algorithm takes on inputs of size n

Why Worst-Case Analysis?

Appropriate for time-critical applications, e.g. avionics
Unlike Average-Case, no debate about what the right definition is
If worst >> average, then (a) alg is doing something pretty subtle, & (b) are hard instances really that rare?
Analysis often easier
Result is often representative of "typical" problem instances
Of course there are exceptions…
**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?

- 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 to do it today, how much longer will it take when my business problems are twice as large? (E.g. today: $cn^2$, next year: $c(2n)^2 = 4cn^2$; 4 x longer.)

**Complexity**

The complexity of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes, with each problem size $n$.

Mathematically,

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

that is $T$ is a function that maps positive integers (giving problem sizes) to positive real numbers (giving number of steps).
O-notation etc

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 \( \leq c \cdot g(n) \).
- \( f(n) \) is \( \Omega(g(n)) \) iff there is a constant \( c > 0 \) so that \( f(n) \) is eventually always \( \geq c \cdot g(n) \).
- \( f(n) \) is \( \Theta(g(n)) \) iff there are constants \( c_1, c_2 > 0 \) so that eventually always \( c_1 g(n) \leq f(n) \leq c_2 g(n) \).

Examples

\( 10n^2 - 16n + 100 \) 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 \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
- If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f = \Omega(h) \).
- If \( f = \Theta(g) \) and \( g = \Theta(h) \) then \( f = \Theta(h) \).

Additivity.
- If \( f = O(h) \) and \( g = O(h) \) then \( f + g = O(h) \).
- If \( f = \Omega(h) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
- If \( f = \Theta(h) \) and \( g = O(h) \) then \( f + g = \Theta(h) \).

Asymptotic Bounds for Some Common Functions

Polynomials:
- \( a_0 + a_1n + \ldots + a_d n^d \) is \( \Theta(n^d) \) if \( a_d > 0 \)

Logarithms:
- \( O(\log_a n) = O(\log_b n) \) for any constants \( a, b > 0 \)

Logarithms:
- For all \( x > 0 \), \( \log n = O(n^x) \)
“One-Way Equalities”

“2 + 2 is 4” vs 2 + 2 = 4 vs 4 = 2 + 2
“All dogs are mammals” vs “All mammals are dogs”
2n² + 5n is $O(n³)$ vs $2n² + 5n = O(n³)$ vs $O(n³) = 2n² + 5n$
OK to put big-$O$ in R.H.S. of equality, but not left.
[Better, but uncommon, notation: $T(n) \in O(f(n))$.]

Working with $O$-$\Omega$-$\Theta$ notation

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

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

$(n+a)^b \geq (n/2)^b$ for $n \geq 2|a|$ (even if $a < 0$
= $2^{-b}n^b$
= $c'n^b$ for $c' = 2^{-b}$
so $(n+a)^b$ is $\Omega(n^b)$

Big-Theta, etc. not always “nice”

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

$f(n) = \Theta(n^2)$ for any $a$.

Fortunately, such nasty cases are rare

$f(n \log n) = \Theta(n^a)$ for any $a$, either, but at least it’s simpler.
A Possible Misunderstanding?

We have looked at
type of complexity analysis
worst-, best-, average-case
types of function bounds
$O$, $\Omega$, $\Theta$

These two considerations are independent of each other
one can do any type of function bound with any type of
complexity analysis - measuring different things with
same yardstick

Insertion Sort:
$\Omega(n^2)$ (worst case)
$O(n)$ (best case)

Polynomial time

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

Asymptotic Bounds for Some
Common Functions

Exponentials.
For all $r > 1$ and all $d > 0$,

$$n^d = O(r^n).$$

Why It Matters

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log n$</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>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>&lt; 1 min</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>50</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
</tr>
<tr>
<td>100</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>11 min</td>
<td>36 years</td>
</tr>
<tr>
<td>1,000</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>12,692 years</td>
<td>1017 years</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>1 sec</td>
<td>1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</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>
</tr>
<tr>
<td>1,000,000</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>
</tr>
</tbody>
</table>

In cases where the running time exceeds 375 years, we simply record the algorithm as
taking a very long time.

Table 2.1 The running times (rounded up) of different algorithms on inputs of
increasing size, for a processor performing 1 million high-level instructions per second.
In cases where the running time exceeds 375 years, we simply record the algorithm as
taking a very long time.
Geek-speak Faux Pas du Jour

“Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.”
Statement doesn’t "type-check."
Use $\Omega$ for lower bounds.

Summary

Typical initial goal for algorithm analysis is to find a reasonably tight bound on worst case running time as a function of problem size i.e., $\Theta$ if possible

Use $\Omega$ for lower bounds.

i.e., $O$ or $\Theta$

usually upper bound

This is rarely the last word, but often helps separate good algorithms from blatantly poor ones - so you can concentrate on the good ones!