Efficiency

• Our correct TSP algorithm was incredibly slow
• Basically slow no matter what computer you have
• We would like a general theory of “efficiency” that is
  – Simple
  – Relatively independent of changing technology
  – But still useful for prediction - “theoretically bad” algorithms should be bad in practice and vice versa (usually)
Measuring efficiency: The RAM model

• RAM = Random Access Machine

• Time \approx \# \text{ of instructions executed in an ideal assembly language}
  – each simple operation (+, *, -, =, if, call) takes one time step
  – each memory access takes one time step

• No 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
- 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
Complexity analysis

• Problem size $n$
  
  – **Worst-case complexity**: $\text{max} \ # \ steps$ algorithm takes on any input of size $n$
  
  – **Best-case complexity**: $\text{min} \ # \ steps$ algorithm takes on any input of size $n$
  
  – **Average-case complexity**: $\text{avg} \ # \ steps$ algorithm takes on inputs of size $n$
Pros and cons:

• Best-case
  – unrealistic overselling
  – can “cheat”: tune algorithm for one easy input

• Worst-case
  – a fast algorithm has a comforting guarantee
  – no way to cheat by hard-coding special cases
  – maybe too pessimistic

• Average-case
  – over what probability distribution? (different people may have different “average” problems)
  – analysis hard
Why Worst-Case Analysis?

• Appropriate for time-critical applications, e.g. avionics
• Unlike Average-Case, no debate about what the right definition is
• Analysis often easier
• Result is often representative of "typical" problem instances
• Of course there are exceptions…
General Goals

• Characterize *growth rate* of 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 best/worst/average-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 size to positive real numbers giving number of steps.
Complexity

Time

Problem size

$T(n)$
Complexity

Time

Problem size

T(n)

2n \log_2 n

n \log_2 n
O-notation etc

• Given two functions \( f \) and \( g: \mathbb{N} \rightarrow \mathbb{R} \)
  
  – \( f(n) \) 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 is are constants \( c_1 \) and \( c_2 > 0 \) 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:
  \( a_0 + a_1 n + \ldots + a_d n^d \) 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
• “Every dog is a mammal” vs “Every mammal is a dog”
• $2n^2 + 5n$ is $O(n^3)$ vs $2n^2 + 5n = O(n^3)$ vs $O(n^3) = 2n^2 + 5n$ FALSE
• OK to put big-O in R.H.S. of equality, but not left. Better notation: $T(n) \in O(f(n))$. 
Working with $\mathcal{O}$-$\Omega$-$\Theta$ notation

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

- $(n+a)^b \leq (2n)^b$ for $n \geq |a|$  
  $= 2^b n^b$  
  $= cn^b$ 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 $a < 0$)  
  $= 2^{-b} n^b$  
  $= c'n$ for $c' = 2^{-b}$  
  so $(n+a)^b$ is $\Omega(n^b)$
Working with $\Theta$ notation

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

\[
\log_a b = x \text{ means } a^x = b
\]

\[
a^{\log_a b} = b
\]

\[
(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n
\]

\[
(\log_a b)(\log_b n) = \log_a n
\]

\[
c \log_b n = \log_a n \text{ for the constant } c = \log_a b
\]

So:

\[
\log_b n = \Theta(\log_a n) = \Theta(\log n)
\]
Domination

• $f(n)$ is $o(g(n))$ iff $\lim_{n \to \infty} f(n)/g(n) = 0$
  – that is $g(n)$ dominates $f(n)$

• If $\alpha \leq \beta$ then $n^\alpha$ is $O(n^\beta)$

• If $\alpha < \beta$ then $n^\alpha$ is $o(n^\beta)$

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

• $n^2 = o(n^3)$ [Use algebra]:

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

• $n^3 = o(e^n)$ [Use L’Hospital’s rule 3 times]:

$$\lim_{n \to \infty} \frac{n^3}{e^n} = \lim_{n \to \infty} \frac{3n^2}{e^n} = \lim_{n \to \infty} \frac{6n}{e^n} = \lim_{n \to \infty} \frac{6}{e^n} = 0$$
Big-Theta, etc. not always “nice”

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

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

Fortunately, such nasty cases are rare

\[ f(n \log n) \neq \Theta(n^a) \text{ for any } a, \text{ either, but at least it’s simpler.} \]
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

Insertion Sort:

$\Omega(n^2)$ (worst case)

$O(n)$ (best case)
Asymptotic Bounds for Some Common Functions

• Exponentials. For all $r > 1$ and all $d > 0$, $n^d = O(r^n)$.

Every exponential grows faster than every polynomial.
Polynomial time

• Running time is $O(n^d)$ for some constant $d$ independent of the input size $n$. 
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>
"Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons."

- Statement doesn't "type-check."
- Use $\Omega$ for lower bounds.