Lecture 2 – Asymptotic Notation
Reading: KT Sections 2.1 and 2.2

Partial content of these slides have been obtained from the official lecture slides that accompany the textbook. A complete set of slides can be found at: http://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Algorithm efficiency

- What makes us say that an algorithm is efficient?
  - Real answer: when it’s better than its brute force counter-part

  Brute force: For many nontrivial problems, there is a natural brute-force search algorithm that checks every possible solution.
  - Typically takes $2^n$ time or worse for inputs of size $n$.
  - Unacceptable in practice.

Remember the interval scheduling problem from last time?

2. Algorithm Analysis

- computational tractability
- asymptotic order of growth
- survey of common running times

Polynomial time algorithms

We say that an algorithm is efficient if it has a polynomial running time.

Justification. It really works in practice!
- In practice, the poly-time algorithms that people develop have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
Worst case analysis

Worst case. Running time guarantee for any input of size $n$.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Exceptions. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.

Other types of analyses

Worst case. Running time guarantee for any input of size $n$.
Ex. Heapsort requires at most $2n \log_2 n$ compares to sort $n$ elements.

The way things grow

By the numbers

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n^t$</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>
</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>&lt; 1 sec</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>&lt; 1 sec</td>
<td>11 min</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>36 years</td>
<td>very long</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>10^12 years</td>
<td>very long</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>&lt; 1 sec</td>
<td>10 sec</td>
<td>12 days</td>
<td>367,200 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Agenda

2. Algorithm Analysis
- computational tractability
- asymptotic order of growth
- survey of common running times

Big-Oh notation

Upper bounds. \( T(n) \) is \( O(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that \( T(n) \leq c \cdot f(n) \) for all \( n \geq n_0 \).

- This means that \( T(n) \) grows no faster than \( f(n) \).
- For example, let’s consider \( 17n^2 \) and \( n^2 \)

Common mistakes

Equals sign. \( O(f(n)) \) is a set of functions, but computer scientists often write \( T(n) = O(f(n)) \) instead of \( T(n) \in O(f(n)) \).

Ex. Consider \( f(n) = 5n^3 \) and \( g(n) = 3n^2 \).
- We have \( f(n) = O(n^3) = g(n) \).
- Thus, \( f(n) = g(n) \). ❌

Big-Omega notation

Lower bounds. \( T(n) \) is \( \Omega(f(n)) \) if there exist constants \( c > 0 \) and \( n_0 \geq 0 \) such that \( T(n) \geq c \cdot f(n) \) for all \( n \geq n_0 \).

Typical usage. Any compare-based sorting algorithm requires \( \Omega(n \log n) \) compares in the worst case.

Meaningless statement. Any compare-based sorting algorithm requires at least \( O(n \log n) \) compares in the worst case.
Big-Theta notation

Tight bounds. \( T(n) \) is \( \Theta(f(n)) \) if there exist constants \( c_1 > 0, c_2 > 0, \) and \( n_0 \geq 0 \) such that \( c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n) \) for all \( n \geq n_0 \).

Ex. \( T(n) = 32n^2 + 17n + 1 \).

Typical usage. Mergesort makes \( \Theta(n \log n) \) compares to sort \( n \) elements.

Some properties to know...

Polynomials. Let \( T(n) = a_0 + a_1 n + \ldots + a_d n^d \) with \( a_d > 0 \). Then, \( T(n) = \Theta(n^d) \).

\[ \lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0 \]

Agenda

Can you think of a?

- Linear time algorithm
- Sublinear time algorithm
- Linearithmic time algorithm
- Quadratic time algorithm
- Cubic time algorithm