Lecture 4 – 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/

Agenda

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

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.

**Exceptions.** Some poly-time algorithms do have high constants and/or exponents, and/or are useless in practice.

Q. Which would you prefer $20n^{120}$ vs. $n^{1.02} \ln n$?
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 $2 n \log_2 n$ compares to sort $n$ elements.

**Probabilistic.** Expected running time of a randomized algorithm.
**Ex.** The expected number of compares to quicksort $n$ elements is $\sim 2n \ln n$.

**Amortized.** Worst-case running time for any sequence of $n$ operations.
**Ex.** Starting from an empty stack, any sequence of $n$ push and pop operations takes $O(n)$ primitive computational steps using a resizing array.

**Average-case.** Expected running time for a random input of size $n$.
**Ex.** The expected number of character compares performed by 3-way radix quicksort on $n$ uniformly random strings is $\sim 2n \ln n$.

**Also.** Smoothed analysis, competitive analysis, ...
The way things grow

By the numbers

<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>$n = 10$</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>4 sec</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>16 years</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

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 $10^{20}$ years, we simply record the algorithm as taking a very long time.
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 \( n^2 \)

Can you find \( c \) and \( n_0 \)?
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 $f(n) = O(n^2) = g(n)$.
- Thus, $f(n) = g(n)$. \(\times\)

**Domain.** The domain of $f(n)$ is typically the natural numbers \{0, 1, 2, ...\}.
- Sometimes we restrict to a subset of the natural numbers.
- Other times we extend to the reals.

**Non-negative functions.** When using big-Oh notation, we assume that the functions involved are (asymptotically) non-negative.

---

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.** $T(n) = 32n^2 + 17n + 1$.
- $T(n)$ is $\Theta(n^2)$.
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

**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, $T(n)$ is $\Theta(n^d)$.

**Pf.** $\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$

**Logarithms.** $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a$, $b > 0$.

**Logarithms and polynomials.** For every $d > 0$, $\log n$ is $O(n^d)$.

**Exponentials and polynomials.** For every $r > 1$ and every $d > 0$, $n^d$ is $O(r^n)$.

**Pf.** $\lim_{n \to \infty} \frac{n^d}{r^n} = 0$
Agenda

2. Algorithm Analysis

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

Can you think of a?

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