Running Times and Asymptotic Notation
As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time?

- Charles Babbage

Computational Tractability
Computational Tractability

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that tries every possible solution.
- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $a > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $a N^d$ steps.

**Def.** An algorithm is poly-time if the above scaling property holds.

\[ \text{choose } C = 2^d \]
Worst Case Analysis
Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.
Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size $N$.
- Generally captures efficiency in practice.
Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
Worst Case Analysis

**Worst case running time.** Obtain bound on *largest possible* running time of algorithm on any input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.
Worst Case Analysis

**Worst case running time.** Obtain bound on largest possible running time of algorithm on any input of a given size $N$.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

**Average case running time.** Obtain bound on running time of algorithm on random input as a function of input size $N$. 
Worst Case Analysis

**Worst case running time.** Obtain bound on largest possible running time of algorithm on any input of a given size $N$.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

**Average case running time.** Obtain bound on running time of algorithm on random input as a function of input size $N$.
- Hard (or impossible) to accurately model real instances by random distributions.
Worst Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on any input of a given size N.
- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size N.
- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst Case Polynomial-Time
Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.
Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.
Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!
Worst Case Polynomial-Time

**Def.** An algorithm is **efficient** if its running time is polynomial.

Justification: It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.

\[ simplex \text{ method} \]
\[ Unix \text{ grep} \]
\[ Primality testing \]
Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
Worst Case Polynomial-Time

Def. An algorithm is **efficient** if its running time is polynomial.

**Justification:** It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop *almost always* have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
Worst Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.
Worst Case Polynomial-Time

Def. An algorithm is **efficient** if its running time is polynomial.

**Justification:** *It really works in practice!*
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always 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 are useless in practice.

- Primality testing
- simplex method
- Unix grep
Worst Case Polynomial-Time

**Def.** An algorithm is **efficient** if its running time is polynomial.

**Justification:** It really works in practice!
- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop **almost always** 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 are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.
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>

Note: age of Universe $\sim 10^{10}$ years...
Computer Science Approach to problem solving

- If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem???
Are there some problems that cannot be solved at all? and, are there problems that cannot be solved efficiently??
Computer Science Approach to problem solving

If my boss / supervisor / teacher formulates a problem to be solved urgently, can I write a program to efficiently solve this problem ???
Asymptotic order of Growth and Notation
Asymptotic order of Growth and Notation
Asymptotic order of Growth and Notation

*Upper bounds.* $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

**Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

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

**Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

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

**Lower bounds.** \(T(n) = \Omega(f(n))\) if there exist constants \(c > 0\) and \(n_0 \geq 0\) such that for all \(n \geq n_0\) we have \(T(n) \geq c \cdot f(n)\).
Asymptotic order of Growth and Notation

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

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

**Tight bounds.** T(n) is \( \Theta(f(n)) \) if T(n) is both \( O(f(n)) \) and \( \Omega(f(n)) \).
Asymptotic order of Growth and Notation

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

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

**Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$. 
Asymptotic order of Growth and Notation

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

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

**Tight bounds.** \( T(n) \) is \( \Theta(f(n)) \) if \( T(n) \) is both \( O(f(n)) \) and \( \Omega(f(n)) \).

**Ex:** \( T(n) = 32n^2 + 17n + 32 \).
Asymptotic order of Growth and Notation

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

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

**Tight bounds.** \( T(n) \) is \( \Theta(f(n)) \) if \( T(n) \) is both \( O(f(n)) \) and \( \Omega(f(n)) \).

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

- \( T(n) \) is \( O(n^2) \), \( O(n^3) \), \( \Omega(n^2) \), \( \Omega(n) \), and \( \Theta(n^2) \) .
Asymptotic order of Growth and Notation

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

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

**Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

**Ex:** $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $O(n^2), O(n^3), \Omega(n^2), \Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n), \Omega(n^3), \Theta(n)$, or $\Theta(n^3)$.
Asymptotic order of Growth and Notation
Asymptotic order of Growth and Notation

**Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

**Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

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

**Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
Asymptotic order of Growth and Notation

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

**Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$. 
Asymptotic order of Growth and Notation

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

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

**Ex:** \( T(n) = 32n^2 + 17n + 32 \).
Asymptotic order of Growth and Notation

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

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

**Ex:**  $T(n) = 32n^2 + 17n + 32$. 
Asymptotic order of Growth and Notation

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

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

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

- \( T(n) \) is \( O(n^2) \) since there exists \( c = 81 \) and \( n_0 = 1 \) such that for all \( n \geq 1 \) we have \( T(n) \leq 32n^2 + 17n^2 + 32n^2 = 81n^2 \).
Asymptotic order of Growth and Notation

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

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

**Ex:** $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$ since there exists $c = 81$ and $n_0 = 1$ such that for all $n \geq 1$ we have $T(n) \leq 32n^2 + 17n^2 + 32n^2 = 81n^2$.
- $T(n)$ is $\Omega(n^2)$ since there exists $c = 1$ and $n_0 = 0$ such that for all $n \geq 0$ we have $T(n) \geq n^2$. 
Asymptotic order of Growth and Notation

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

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

**Ex:** $T(n) = 32n^2 + 17n + 32$.

- $T(n)$ is $O(n^2)$ since there exists $c = 81$ and $n_0 = 1$ such that for all $n \geq 1$ we have $T(n) \leq 32n^2 + 17n^2 + 32n^2 = 81n^2$.

- $T(n)$ is $\Omega(n^2)$ since there exists $c = 1$ and $n_0 = 0$ such that for all $n \geq 0$ we have $T(n) \geq n^2$.

- $T(n)$ is not $O(n)$ since for all $c > 0$ and $n_0 \geq 0$ there exists $n = \lceil c + 1/c + n_0 \rceil$ such that $T(n) > 32(c+1/c+n_0)^2 + 17(c+1/c+n_0) + 32 \geq c^2 + c\cdot n_0 + 32 \geq cn$. 
Asymptotic Notation
Asymptotic Notation

Frequent Abuse of notation. $T(n) = O(f(n))$. 
Asymptotic Notation

**Frequent Abuse of notation.** \( T(n) = O(f(n)) \).

- Not transitive:
Asymptotic Notation

Frequent Abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3 \); \( g(n) = 3n^2 \)
Asymptotic Notation

Frequent Abuse of notation.  \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
Asymptotic Notation

Frequent Abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3)$ and $g(n) = O(n^3)$
  - but $f(n) \neq g(n)$ and $f(n) \neq O(g(n))$. 
Frequent Abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
  - but \( f(n) \neq g(n) \) and \( f(n) \neq O(g(n)) \).

- Better notations: \( T(n) \in O(f(n)) \), \( T(n) \) is \( O(f(n)) \).
Asymptotic Notation

Frequent Abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
  - but \( f(n) \neq g(n) \) and \( f(n) \neq O(g(n)) \).

- Better notations: \( T(n) \in O(f(n)) \), \( T(n) \) is \( O(f(n)) \).
Asymptotic Notation

Frequent Abuse of notation. \( T(n) = O(f(n)) \).

- Not transitive:
  - \( f(n) = 5n^3 \); \( g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
  - but \( f(n) \neq g(n) \) and \( f(n) \neq O(g(n)) \).
- Better notations: \( T(n) \in O(f(n)), T(n) \) is \( O(f(n)) \).

Meaningless statement. "Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons."
Asymptotic Notation

**Frequent Abuse of notation.** \( T(n) = O(f(n)) \).
- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
  - but \( f(n) \neq g(n) \) and \( f(n) \neq O(g(n)) \).
- Better notations: \( T(n) \in O(f(n)), \ T(n) \textup{ is } O(f(n)) \).

**Meaningless statement.** "Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons."
- Statement doesn't "type-check"."
Asymptotic Notation

Frequent Abuse of notation. \( T(n) = O(f(n)). \)

- Not transitive:
  - \( f(n) = 5n^3; \ g(n) = 3n^2 \)
  - \( f(n) = O(n^3) \) and \( g(n) = O(n^3) \)
  - but \( f(n) \neq g(n) \) and \( f(n) \neq O(g(n)). \)
- Better notations: \( T(n) \in O(f(n)), \ T(n) \text{ is } O(f(n)). \)

Meaningless statement. "Any comparison-based sorting algorithm requires at least \( O(n \log n) \) comparisons."

- Statement doesn't "type-check".
- The constant function \( f(n)=1 \) is \( O(n \log n). \)
Asymptotic Notation

Frequent Abuse of notation. $T(n) = O(f(n))$.

- Not transitive:
  - $f(n) = 5n^3; \ g(n) = 3n^2$
  - $f(n) = O(n^3)$ and $g(n) = O(n^3)$
  - but $f(n) \neq g(n)$ and $f(n) \neq O(g(n))$.
- Better notations: $T(n) \in O(f(n)), T(n) \text{ is } O(f(n))$.

Meaningless statement. "Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons."

- Statement doesn't "type-check".
- The constant function $f(n)=1$ is $O(n \log n)$.
- Use $\Omega$ for lower bounds.
Asymptotic Notation
Asymptotic Notation

Transitivity.
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$. 
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$.
- If $f$ is $\Omega(g)$ and $g$ is $\Omega(h)$ then $f$ is $\Omega(h)$. 
Asymptotic Notation

**Transitivity.**
- If \( f \) is \( O(g) \) and \( g \) is \( O(h) \) then \( f \) is \( O(h) \).
- If \( f \) is \( \Omega(g) \) and \( g \) is \( \Omega(h) \) then \( f \) is \( \Omega(h) \).
- If \( f \) is \( \Theta(g) \) and \( g \) is \( \Theta(h) \) then \( f \) is \( \Theta(h) \).
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$.
- If $f$ is $\Omega(g)$ and $g$ is $\Omega(h)$ then $f$ is $\Omega(h)$.
- If $f$ is $\Theta(g)$ and $g$ is $\Theta(h)$ then $f$ is $\Theta(h)$.
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$.
- If $f$ is $\Omega(g)$ and $g$ is $\Omega(h)$ then $f$ is $\Omega(h)$.
- If $f$ is $\Theta(g)$ and $g$ is $\Theta(h)$ then $f$ is $\Theta(h)$.
Asymptotic Notation

Transitivity.
- If \( f \) is \( \mathcal{O}(g) \) and \( g \) is \( \mathcal{O}(h) \) then \( f \) is \( \mathcal{O}(h) \).
- If \( f \) is \( \Omega(g) \) and \( g \) is \( \Omega(h) \) then \( f \) is \( \Omega(h) \).
- If \( f \) is \( \Theta(g) \) and \( g \) is \( \Theta(h) \) then \( f \) is \( \Theta(h) \).

Additivity.
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$.
- If $f$ is $\Omega(g)$ and $g$ is $\Omega(h)$ then $f$ is $\Omega(h)$.
- If $f$ is $\Theta(g)$ and $g$ is $\Theta(h)$ then $f$ is $\Theta(h)$.

Additivity.
- If $f$ is $O(h)$ and $g$ is $O(h)$ then $f + g$ is $O(h)$. 
Asymptotic Notation

Transitivity.
- If $f$ is $O(g)$ and $g$ is $O(h)$ then $f$ is $O(h)$.
- If $f$ is $\Omega(g)$ and $g$ is $\Omega(h)$ then $f$ is $\Omega(h)$.
- If $f$ is $\Theta(g)$ and $g$ is $\Theta(h)$ then $f$ is $\Theta(h)$.

Additivity.
- If $f$ is $O(h)$ and $g$ is $O(h)$ then $f + g$ is $O(h)$.
- If $f$ is $\Omega(h)$ and $g$ is $\Omega(h)$ then $f + g$ is $\Omega(h)$.
Asymptotic Notation

Transitivity.
- If \( f \) is \( O(g) \) and \( g \) is \( O(h) \) then \( f \) is \( O(h) \).
- If \( f \) is \( \Omega(g) \) and \( g \) is \( \Omega(h) \) then \( f \) is \( \Omega(h) \).
- If \( f \) is \( \Theta(g) \) and \( g \) is \( \Theta(h) \) then \( f \) is \( \Theta(h) \).

Additivity.
- If \( f \) is \( O(h) \) and \( g \) is \( O(h) \) then \( f + g \) is \( O(h) \).
- If \( f \) is \( \Omega(h) \) and \( g \) is \( \Omega(h) \) then \( f + g \) is \( \Omega(h) \).
- If \( f \) is \( \Theta(h) \) and \( g \) is \( O(h) \) then \( f + g \) is \( \Theta(h) \).
Frequently Used Functions

\[ \text{can avoid specifying the base} \]

\[ \text{log grows slower than every polynomial} \]

\[ \text{every exponential grows faster than every polynomial} \]
Frequently Used Functions

**Polynomials.** $a_0 + a_1n + \ldots + a_dn^d$ is $\Theta(n^d)$ if $a_d > 0$.

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

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

also can avoid specifying the base

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

log grows slower than every polynomial

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

every exponential grows faster than every polynomial
Asymptotic Notation

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as $n$ goes to infinity. Essentially, if the ratio of functions $f(n)$ and $g(n)$ converges to a positive constant as $n$ goes to infinity, then $f(n)$ is $\Theta(g(n))$. 
Chapter 2 Basics of Algorithm Analysis

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as $n$ goes to infinity. Essentially, if the ratio of functions $f(n)$ and $g(n)$ converges to a positive constant as $n$ goes to infinity, then $f(n)$ is $\Theta(g(n))$.

\[(2.1) \quad \text{Let } f \text{ and } g \text{ be two functions that}
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)}
\]

exists and is equal to some number $c > 0$. Then $f(n)$ is $\Theta(g(n))$.\]
Chapter 2 Basics of Algorithm Analysis

Precisely up to constant factors. And as the definition of \( \Theta(-) \) shows, one can obtain such bounds by closing the gap between an upper bound and a lower bound. For example, sometimes you will read a (slightly informally phrased) sentence such as "An upper bound of \( O(n^3) \) has been shown on the worst-case running time of the algorithm, but there is no example known on which the algorithm runs for more than \( \Theta(n^2) \) steps." This is implicitly an invitation to search for an asymptotically tight bound on the algorithm's worst-case running time.

Sometimes one can also obtain an asymptotically tight bound directly by computing a limit as \( n \) goes to infinity. Essentially, if the ratio of functions \( f(n) \) and \( g(n) \) converges to a positive constant as \( n \) goes to infinity, then \( f(n) \) is \( \Theta(g(n)) \).

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} \]

exists and is equal to some number \( c > 0 \). Then \( f(n) \) is \( \Theta(g(n)) \).

**Proof.** We will use the fact that the limit exists and is positive to show that \( f(n) \) is \( O(g(n)) \) and \( f(n) \) is \( \Omega(g(n)) \), as required by the definition of \( \Theta(\cdot) \).

Since

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0, \]

it follows from the definition of a limit that there is some \( n_0 \) beyond which the ratio is always between \( \frac{1}{2}c \) and \( 2c \). Thus, \( f(n) \leq 2cg(n) \) for all \( n \geq n_0 \), which implies that \( f(n) \) is \( O(g(n)) \); and \( f(n) \geq \frac{1}{2}cg(n) \) for all \( n \geq n_0 \), which implies that \( f(n) \) is \( \Omega(g(n)) \).
Winter 2016
COMP-250: Introduction to Computer Science
Lecture 9, February 9, 2016